Emerging Issues in Computer Science

Week 8: Emerging Trends In Software Engineering

Lecturer: Ikwap Flavia Agatha

Lecture learning outcome

At the end of this lecture, You will be able to

- Understand Software Engineering
- Understand Storage Models
- Understand Software Project management
- Understand Software metrics
- Understand Software Requirement Specification (SRS)
- Understand Software Quality Standards
- Understand Trends in software engineering

Software Engineering

- Software Engineering provides a standard procedure to design and develop a software.
- **What is Software Engineering?**
- The term software engineering is the product of two words, software, and engineering.
- The software is a collection of integrated programs.
- Software subsists of carefully-organized instructions and code written by developers on any of various particular computer languages.
- Computer programs and related documentation such as requirements, design models and user manuals.

What is Software Engineering?

- Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.
- Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

Why Software Engineering is required

- **Managing Large Systems:** As software projects grow in size and complexity, structured engineering practices are necessary to handle development efficiently.
- **Scalability:** Proper engineering enables existing software to be expanded or modified without starting from scratch.
- **Cost Control:** Using disciplined software engineering processes helps reduce development costs and avoid expensive errors or delays.
- **Handling Software Changes**: Since software must evolve with user needs and environments, software engineering ensures updates and modifications can be made systematically.
- **Improved Quality**: A well-defined development process leads to more reliable, maintainable, and high-quality software products.

Need for Software Engineering

- **Growing User Demands and Changing Environments**
  Rapidly evolving requirements and operational contexts necessitate a systematic approach to development.
- **Large-Scale Development:** Just like building a complex structure requires architecture and planning, large programs need structured processes.
- **Adaptability Challenges:** Without a sound engineering foundation, modifying or scaling software becomes inefficient and costly.
- **High Development Costs:** Unlike hardware, software costs remain high unless efficient engineering methods are used.
- **Continuous Evolution:** Software must adapt to changing client needs, requiring frequent updates and enhancements, which can only be handled with proper engineering.
- **Assured Quality:** Structured methods ensure that the final product meets quality standards and user expectations.

Models in software Engineering
Prototype Model in Software Engineering

- The Prototype Model involves creating a simplified version of the software (a prototype) before full-scale development begins. This mock-up helps understand system functionality when client requirements are unclear.

Key Steps:

- **Requirement Gathering:** Collect user needs and expectations.
- **Quick Design:** Develop a rough design based on initial requirements.
- **Build Prototype:** Create a working model reflecting key features.
- **User Evaluation:** Gather user feedback on the prototype.
- **Prototype Refinement:** Make improvements based on user input.
- **Develop Final Product:** Build the actual software using the refined model.

Advantages

- Reduces the risk of misinterpreted requirements.
- Ideal for projects with changing or unclear needs.
- Improves communication with stakeholders through early visuals.
- Supports early marketing and reduces long-term maintenance.
- Allows earlier detection and correction of errors.

Disadvantages:

- Risk of poor-quality prototype becoming the final product.
- Requires frequent and close collaboration with users.
- May lead to cost and time overruns.
- Risk of neglecting formal design and analysis phases.

Disadvantages

- Tools needed can be costly and specialized.
- Time-intensive development approach.

Types of Prototyping:

- **Throwaway Prototyping:** Quickly built and discarded after gathering requirements.
- **Evolutionary Prototyping:** Refined continuously into the final product.
- **Incremental Prototyping:** Several small prototypes combined into a complete system.
- **Extreme Prototyping:** Often used in web development, involving layered prototyping.

Application of Prototyping

Game Development, Web Development, User Interface Design, Application Development

Evolutionary Process Model

- This model is similar to iterative development but focuses on gradual evolution. Example Application include: In a database app: One cycle builds the GUI, Another handles file storage, A third supports queries, and the last handles data updates.

Benefits:

- Reduces project risk through early issue detection.
- Supports cost-effective experimentation.
- Allows early feedback from marketing and users.
- Improves alignment with user and market needs.
- Enhances visibility of project progress.
- Boosts team motivation and productivity.
- Helps meet time-to-market goals with early versions.
- Modifications during development.

Waterfall Model
Phases of the Waterfall Model

- **Requirement Analysis:** The primary goal is to gather and document the customer's requirements in detail. Which outlines *what* the system should do, not *how* it will do it.
- **Design Phase:** Converts the SRS into a structured design, including both high-level architecture and detailed component specifications.
- **Implementation & Unit Testing:** Developers write the actual code based on the design. Each module is coded and individually tested to verify functionality before integration.
- **Integration & System Testing:** The objective is to ensure that modules interact properly and the entire system meets the user’s needs.
- **Operation & Maintenance:** The software is monitored and maintained to fix bugs, improve performance, and adapt to new requirements.

Who Uses the Waterfall Model?

- The Waterfall Model is utilized by project managers and teams across industries like software development, construction, IT, and manufacturing when a structured, step-by-step approach is needed. Each phase must be completed before the next begins, making it suitable for projects where tasks follow a logical sequence.

Advantages of the Waterfall Model

- Simple to use and manage with minimal resources.
- Clearly defined and fixed requirements simplify planning.
- Well-structured phases make tracking progress straightforward.
- Predictable delivery timelines and cost estimation are possible.
- Transparent process improves client understanding and control.

Disadvantages of the Waterfall Model

- High risk in large or complex projects.
- Poor adaptability to changes during development.
- Difficult to revisit earlier phases once development progresses.
- Late testing phase can delay the discovery of issues or risks.

Applications of the Waterfall Model

- Large-scale software projects needing strict processes.
- Safety-critical systems (e.g., in aerospace or healthcare).
- Government and defense projects that require formal documentation and compliance.
- Projects with fixed or well-defined requirements.
- Environments with little or no expected change during development.

RAD Model

- Rapid Application Development (RAD) is a software development approach that emphasizes quick development cycles**,** frequent user feedback, and the use of prototypes to speed up the creation of software while maintaining quality.

Relevance in Modern Development

- RAD aligns well with modern software development due to:
- **Faster Development**: Shortens delivery time using iterative prototyping and component-based development.
- **High Flexibility**: Allows constant user feedback and updates during development.
- **User-Centric Design**: Actively involves end users, leading to more relevant and accepted products.
- **Risk Reduction**: Breaks projects into manageable units, making it easier to identify and resolve problems early.
- **Cost Efficiency**: Limits expensive late-stage revisions by integrating feedback throughout the cycle.

Principles of RAD

- **User Involvement**: Continuous input from end users ensures that the final product aligns with their expectations and reduces the gap between developers and users.
- **Iterative Development**: Projects are divided into cycles (iterations), each involving planning, coding, and testing. This allows regular refinement based on feedback and testing.
- **Prototyping**: Working models of the system are built quickly to visualize the product early. Users can interact with these to provide feedback, improving the end result.
- **Time-Boxing**: Each development phase is assigned a fixed duration. Teams prioritize essential features within this time, ensuring deadlines are met and scope creep is minimized.
- **Reuse of Components**: RAD encourages the use of pre-existing software components or modules, which speeds up development and enhances reliability.

Spiral Model in Software Development

- The Spiral Model blends the structured nature of the Waterfall model with the flexibility of iterative development. It is especially well-suited for large-scale or high-risk projects and prioritizes risk management throughout the development lifecycle.

Key Concepts of the Spiral Model

- Visualized as a spiral with loops, where each loop represents a development phase.
- Each cycle consists of four main stages:
  - **Objective Setting**: Define goals, explore solutions, and identify constraints.
  - **Risk Assessment**: Evaluate potential risks and decide on mitigation strategies.
  - **Development & Validation**: Create prototypes, simulate solutions, and validate through testing.
  - **Planning**: Decide whether to proceed with the next iteration and prepare plans for it.
- This **risk-driven approach** allows for continuous refinement and helps developers deal with uncertainty more effectively by building prototypes at every stage.

Steps in the Spiral Model

- Gather detailed requirements from users and stakeholders.
- Create an initial prototype representing the new system.
- Assess risks and build improved prototypes in iterative steps.
- Terminate the project if risks are too high or unmanageable.
- Repeat prototype cycles until client approval is achieved.
- Final system is developed based on the most refined prototype.
- Thorough system testing is conducted.
- Ongoing maintenance ensures long-term reliability.

Advantages

- Flexible to Change: Easily adapts to evolving requirements.
- Risk Management: Constant risk evaluation reduces project failure chances.
- Client Engagement: Regular feedback loops improve satisfaction and product accuracy.
- Cost Estimation: Incremental planning can simplify budget forecasting.

Disadvantages

- High Cost: Not cost-effective for smaller or low-budget projects.
- Complex Implementation: Requires specialized skills in risk analysis and project documentation.
- Time Uncertainty: Unclear timeline due to undefined number of iterations.
- Management Challenges: Requires strict process discipline and coordination.

Spiral Model

- The Spiral Model is a software development approach that merges the structured stages of the Waterfall model with the flexibility of iterative development. Its main focus is risk management, making it suitable for large, complex, and high-risk projects. Developers progressively build and refine a system through cycles, each represented as a loop in the spiral, allowing for continuous improvements and risk handling.

Steps in the Spiral Model

- Requirement Analysis – Gather system requirements through stakeholder interviews and analysis.
- Initial Prototype Development – Create a basic model that reflects the initial design.
- Refinement Cycle – Evaluate, redesign, and test enhanced versions of the prototype in multiple cycles.
- Risk Evaluation – Terminate if risks (e.g., cost, feasibility) are too high.
- Client Review – Incorporate feedback and repeat prototype improvements until satisfaction is achieved.

- Final Development – Build the actual product using the validated prototype as a base.
- Testing & Quality Assurance – Conduct extensive tests before release.
- Ongoing Maintenance – Ensure stability, minimize downtime, and fix issues post-deployment.

Real-World Applications

- The spiral model is used across industries that benefit from continuous feedback and testing:
- Software: Especially in mobile apps where frequent updates and debugging are common.
- Gaming: Game mechanics and visuals are tested and refined before release.
- E-commerce: Enhancing features based on market trends and user needs.
- Healthcare: Ensures compliance with laws and security standards in medical software.
- Aerospace: Iterative testing and prototyping for satellites and exploration equipment.

Advantages

- High Flexibility: Easily adapts to changes in requirements throughout the project.
- Effective Risk Management: Built-in risk analysis at every stage reduces chances of failure.
- Customer Feedback Integration: Clients can evaluate and suggest changes during development.
- Improved Cost Prediction: Prototypes help estimate cost more accurately in phases.
- Disadvantages
- High Cost: Resource-intensive, making it unsuitable for small-scale projects.
- Dependence on Risk Assessment: Success depends heavily on skilled risk management.
- Complex Process: Requires careful documentation and strict adherence to protocols.
- Uncertain Timelines: Difficult to estimate duration and budget due to undefined iteration count.

Incremental Model in Software Engineering

- The **Incremental Model** divides the software development process into smaller, independent segments or modules. Each segment goes through the full software development lifecycle — from **requirements gathering** to **design, implementation, testing**, and **integration**. Each new increment enhances the functionality of the previously delivered software, continuing until the final system is complete.

Phases of the Incremental Model

- **Requirement Analysis**: Understand and define system requirements, critical for guiding the development of each increment.
- **Design & Development**: Plan and create the architecture and structure of the module.
- **Testing**: Validate both existing and newly added functions after each increment.
- **Implementation**: Final code is written, functionality is tested, and the working product is expanded.

When to Use

- When requirements are clear but will evolve over time.
- For long-term projects.
- When a quick product release is desired.
- When development teams have limited experience.
- When early delivery of high-priority features is needed.

Advantages

- Easier to test and identify errors early.
- More flexible to changes.
- Risk is managed incrementally.
- Useful functionality is delivered early to clients.

Disadvantages

- Requires careful planning and defined interfaces.
- Higher total cost due to repeated phases.
- Needs well-structured and modularized systems.

Agile Model in Software Engineering

- The Agile Model is an iterative, flexible software development methodology that delivers working software in short cycles (iterations), typically 1–4 weeks. Agile focuses on customer collaboration, adaptive planning, and continuous improvement, making it well-suited for dynamic environments with changing requirements.

Phases of Agile Development

- Requirements Gathering: Identify project goals, timeline, and feasibility.
- Design: Define user workflows and high-level architecture with stakeholder input.

Phases of Agile Development

- Construction/Iteration: Develop basic features, aiming for a minimal viable product.
- Testing: Continuous testing ensures functionality and quality.
- Deployment: Release the product for end-user use.
- Feedback: Gather user input to refine the product in the next cycle.

When to Use Agile

- When frequent changes in requirements are expected.
- For small teams with skilled developers.
- When close and regular client interaction is feasible.
- For small to mid-sized projects needing fast delivery.

Advantages

- Fast and frequent delivery.
- Strong customer collaboration.
- Adaptive to change.
- Improves team productivity and reduces delivery time.

Disadvantages

- Minimal documentation can cause confusion.
- Maintenance can be hard once original developers move on.
- Informal processes may not suit all organizations

Iterative Model in Software Engineering

- The **Iterative Model** is a development approach where software is built and refined through repeated cycles (iterations). Initially, a basic version of the software is developed based on partial requirements. With each iteration, the product is evaluated and enhanced based on feedback or new requirements, eventually resulting in a final, polished version. Each iteration is time-bound and includes development, testing, and review.
- This model allows revisiting and modifying earlier stages, promoting continuous improvement until the software meets all objectives.

Phases of the Iterative Model

- **Requirement Gathering & Analysis**
  Stakeholders provide initial requirements. Analysts evaluate feasibility and budget compliance. If viable, the team moves forward.
- **Design**: The architecture of the system is created using design tools like Data Flow Diagrams, UML diagrams, etc.
- **Implementation**: The design is translated into working code, producing the first software version.

Phases of the Iterative Model

- **Testing**: The developed version undergoes testing using techniques such as black-box, white-box, and gray-box methods to identify bugs.
- **Deployment**: Once tested, the software is released into its intended environment.
- **Review**: Feedback is gathered from users and stakeholders. If any flaws or improvement areas are found, the cycle starts over from the requirement phase.
- **Maintenance**: Post-deployment, the software may need updates, bug fixes, or new features — all handled in this phase.

When to Use the Iterative Model

- When requirements are well-understood but expected to evolve.
- For large and complex software applications.
- When future changes and enhancements are likely.

Advantages

- Easier to test and debug in smaller parts.
- Supports parallel development of components.
- Accommodates evolving project requirements.
- Risk management is continuous throughout iterations.
- Less focus on exhaustive documentation, more on product improvement.

Disadvantages

- Not ideal for small or simple projects.
- May require more development resources.
- Constant changes in design can lead to inefficiencies.
- Evolving requirements can increase costs.
- Uncertain project timelines due to ongoing changes.

What is Software Project Management?

- Software project management involves the structured process and expertise required to plan, oversee, and guide software development projects. It is a specialized area focused on organizing, executing, tracking, and controlling the progress of software projects. This process ensures that necessary resources are efficiently managed, scheduled, and used to build software that meets specified requirements.

  Both developers and clients must understand the estimated duration, timeline, and cost associated with the project for effective management.

Key Requirements for Software Project Management

There are three core elements essential to managing software projects effectively:

- **Time**
- **Cost**
- **Quality**

These three aspects—often referred to as the "triple constraint"—must be balanced to ensure project success. Delivering a high-quality software product within budget and on schedule is vital. However, any disruption in one area can negatively affect the other two. Both internal and external factors may influence these constraints.

Role of a Project Manager

- The project manager is the individual accountable for overseeing the entire lifecycle of the project—from planning and design to execution, monitoring, control, and final delivery. This role is critical to the project's success.
  A project manager is tasked with making key decisions that help reduce uncertainty and manage risks. Every decision should positively impact the outcome of the project.

Key Responsibilities of a Project Manager:

- Identifying and managing risks and issues.
- Building the project team and assigning roles and responsibilities.
- Planning and sequencing project activities.
- Tracking progress and providing updates.
- Adjusting the project plan as needed to address challenges.

Software Metrics in Software Engineering

- Software metrics are quantitative measures used to assess various attributes of software.

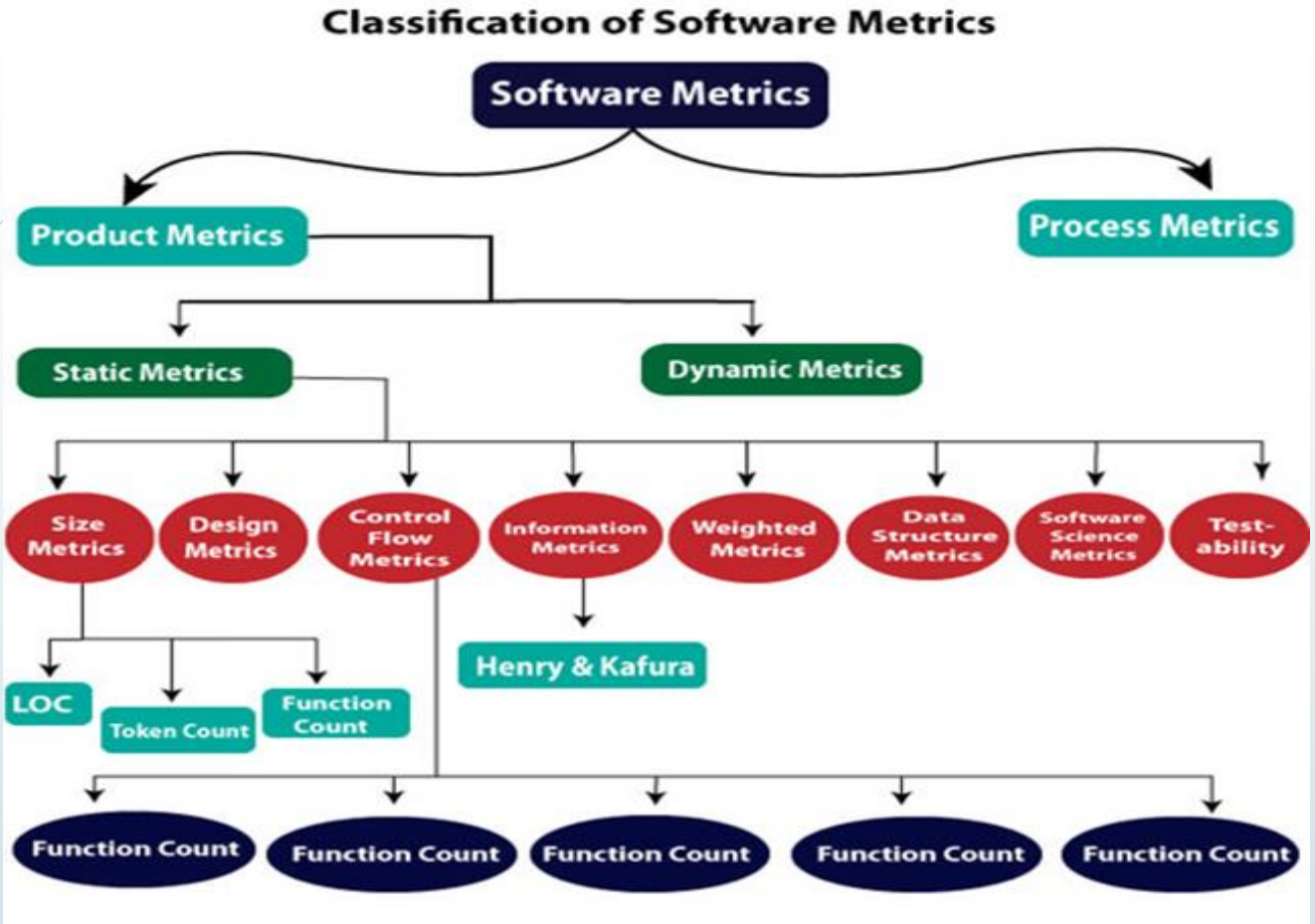
  These measurements are useful for evaluating performance, estimating workloads, tracking productivity, and improving processes.

  They play a role similar to management functions—such as planning, organizing, controlling, and improving—within the development lifecycle.

Types of Software Metrics

- **Product Metrics** – These relate to the attributes of the software itself, such as:
  - The size and complexity of the codebase.
  - The quality and reliability of the software product.

    These metrics can be gathered at different stages of the Software Development Life Cycle (SDLC).
- **Process Metrics** – These focus on the software development process. For example, they measure how effective methods are at detecting defects. Process metrics evaluate the efficiency of tools, techniques, and procedures used during development.



- Software Metrics. A metric is a way to gauge how closely… | by Ruchita Chaudhari | Medium

Types of Software Metrics

- **Internal Metrics:**

  These metrics assess attributes that are primarily of interest to software developers. A common example is *Lines of Code (LOC)*, which helps in evaluating the size or complexity of the codebase.

- **External Metrics:**

  These focus on features that are significant from the end-user's perspective, such as usability, portability, reliability, and functionality.

- **Hybrid Metrics:**

  These metrics blend aspects of product, process, and resource evaluation. An example is *Cost per Function Point (FP)*, where function points quantify software functionality.

Project Metrics

- Used by project managers to monitor the status and performance of a project. These rely on historical project data to predict and track time, cost, and effort. By comparing actual results with initial estimates, these metrics help in reducing risk, improving quality, and managing resources more effectively.

Benefits of Software Metrics

- Enable comparison of various design methods.
- Support evaluation of programming languages based on specific characteristics.
- Assist in assessing the productivity of development teams.
- Help in defining software quality benchmarks.
- Ensure compliance with system specifications.

Benefits of Software Metrics

- Estimate the required development effort.
- Gauge code complexity.
- Decide if complex modules should be divided.
- Assist resource managers in efficient allocation.
- Compare development vs. maintenance costs.
- Provide feedback on progress and quality during development phases.
- Aid in optimal allocation of testing resources.

Limitations of Software Metrics

- Implementation can be complex and costly.
- Based on historical data that may lack accuracy or verification.
- Not suitable for evaluating individual performance.
- Often rely on assumptions and may vary based on tools or environment.
- Estimations in predictive models may be based on imprecise inputs.

Size-Oriented Metrics: LOC (Lines of Code)

- LOC is one of the earliest and simplest ways to measure software size. It assesses productivity by normalizing outputs (e.g., quality or defects) based on the total code length.

Key Points:

- LOC is considered a baseline for size.
- Originated in the era of FORTRAN and COBOL.
- Productivity = KLOC / Effort (person-months).
- Language-dependent and doesn't suit GUI-based systems.
- Different organizations may use inconsistent methods for counting LOC.
- Derived metrics include:

- Errors/KLOC
- Cost/KLOC
- Defects/KLOC
- Documentation/KLOC
- Productivity = KLOC/PM

- **Advantages:**
- Easy to measure and understand.
- **Disadvantages:**
- Ignores software design, complexity, and user functionality.
- Highly dependent on programming language.
- Not applicable to GUI-based or non-code artifacts.
- May incentivize writing unnecessarily lengthy code.

Halstead’s Software Metrics

- Halstead's model views a program as a series of *tokens* (operators and operands). It derives several metrics based on the frequency and variety of these tokens.

Key Metrics:

- **n_1**: Number of unique operators
- **n_2**: Number of unique operands
- **N_1**: Total occurrences of operators
- **N_2**: Total occurrences of operands
- **Program Size (N)** = $N_1 + N_2$
- **Vocabulary Size (n)** = $n_1 + n_2$

Data Structure Metrics

- These metrics help estimate the time and effort needed to process data within software:
- **Amount of Data:**
  - *VARS*: Count of variables.
  - η_2: Count of operands (variables + constants + labels).
  - N_2: Total variable occurrences.
- **Data Usage within Modules:**
  Measures average number of live variables (variables in active use from declaration to last reference).

- **Program Weakness:**

  A program's structural weakness is based on module cohesion. Less cohesive modules require more time and effort.

  - *Module Weakness (WM)* = LV × γ
  - *Program Weakness (WP)* is calculated based on weaknesses of individual modules.

- **Data Sharing Among Modules:**

  High inter-module data sharing increases coupling, thus requiring more effort for coordination and maintenance.

Software Requirement Specification (SRS)

- The **SRS** document is created during the requirements phase of software development. It formally defines what the software must do, serving as a communication bridge between clients and developers.

Functions of SRS:

- Acts as a blueprint for development.
- Allows clients to verify requirements.
- Serves as a contractual agreement when written by developers.

Main Components of an SRS

- Business Drivers: Motivation behind the project.
- Business Model: Organizational context, process flow.
- Functional and System Requirements: Hierarchically organized requirements.
- Use Cases: UML diagrams showing system interactions.
- Technical Specifications: Non-functional constraints and environment details.
- System Characteristics: Qualities like security, scalability, and maintainability.
- Limitations & Assumptions: Constraints and assumptions during development.
- Acceptance Criteria: Standards that must be met for final client approval.

Requirement Analysis in Software Engineering

- Requirement analysis is a crucial phase that follows requirement elicitation. During this stage, the gathered requirements are examined, refined, and clarified to eliminate inconsistencies and ambiguities, aiming for a clear and complete set of system requirements. This process may involve creating visual representations of the system and further interaction with stakeholders to ensure all parties have a shared understanding and to prioritize the requirements effectively.

Steps Involved in Requirement Analysis

- **Creating a Context Diagram:** A context diagram is a high-level model that outlines the system's scope, showing its boundaries and how it interacts with external entities. It helps define what lies within and outside the system, identifying users and other systems involved in data exchanges.

Steps Involved in Requirement Analysis

- **Prototype Development (Optional):** Prototyping involves building a simplified version of the software to help stakeholders visualize the proposed solution. Feedback from users is used to iteratively improve the prototype. This method is particularly useful when requirements are unclear or subject to change.
- **Modeling Requirements:** This step involves using graphical tools to represent data, processes, and system behavior. Techniques such as Data Flow Diagrams (DFDs), Entity-Relationship Diagrams (ERDs), Data Dictionaries, and State Transition Diagrams help in detecting errors, redundancies, and gaps in the requirements.
- **Finalizing Requirements:** Once the models are reviewed and refined, the requirements are finalized. By this point, the system's behavior is well-understood, ambiguities have been resolved, and the data flows between modules are clearly mapped.

Data Flow Diagram (DFD)

- A **Data Flow Diagram (DFD)** is a graphical tool used to depict the flow of data within a system. It illustrates how data moves between processes, data stores, and external entities. DFDs are widely used in structured system analysis, especially in methods like SSADM.

Purpose of DFDs:

- They provide a simplified, visual representation of system processes, helping developers and stakeholders understand system functionality without delving into the programming logic.
- **Basic rules of DFD**
- Label data flows with descriptive text indicating the type of data.
- Label processes with short verb phrases.
- Use noun phrases for data stores.

Basic Rules of DFDs

- Every process and data store must have at least one input and one output.
- Data stores cannot be directly connected to external entities.
- Data should always flow through a process before reaching a data store.
- Avoid intersecting data flow lines to maintain clarity.

Key Components

- **External Entities** – users or systems interacting with the system.
- **Processes** – actions or transformations performed on data.
- **Data Stores** – repositories where data is held.
- **Data Flows** – movement of data between elements.

Data Dictionary

- A **Data Dictionary** serves as a centralized repository of metadata, providing detailed information about data elements used within the system. It supports system design and helps maintain consistency across the project.

Key Elements of a Data Dictionary

- Data item names
- Aliases or alternative names
- Descriptions and purposes
- Relationships to other data items
- Valid value ranges
- Data structures and formats
- Only database administrators typically interact directly with the data dictionary, though its definitions impact all users indirectly.

Software Quality

- **Software Quality** is defined by how well a product meets user needs and requirements as specified in the SRS. However, a software product that is functionally correct may still lack quality if, for example, it has a poor user interface.

Key Quality Attributes:

- **Portability:** Ability to function across different platforms and environments.
- **Usability:** Ease with which users of varying skill levels can use the software.
- **Reusability:** Extent to which code or components can be reused in other applications.
- **Correctness:** Alignment with all stated requirements.
- **Maintainability:** Ease of fixing defects, adding features, or updating functionalities.

Quality standards

- **ISO 9000 Certification**
- **ISO 9000** is a set of international standards for quality management systems. Introduced in 1987, these standards guide organizations in establishing processes that ensure consistent product quality.
- *ISO 9000 Standards Relevant to Industry:*
- **ISO 9001:** Applicable to organizations involved in design, development, production, and service delivery (including software development).
- **ISO 9002:** Applies to companies involved in production only, not design (e.g., manufacturing industries).
- **ISO 9003:** Pertains to organizations focused solely on product installation and testing (e.g., utility companies).
- ISO 9000 focuses on *processes*, not the product itself, with the belief that good processes lead to quality outcomes.

Capability Maturity Model (CMM)

- The **Capability Maturity Model (CMM)** is a framework developed by the Software Engineering Institute (SEI) to assess and improve an organization's software development processes. It defines five levels of maturity, each representing more structured and effective process capabilities.
- *Purpose of CMM:*
- CMM is used to benchmark and enhance the consistency, efficiency, and quality of software development practices within an organization.

Six Sigma in Software Engineering

- **Six Sigma** is a methodology that aims to improve process quality by identifying and eliminating defects. It was first implemented by engineer Bill Smith at Motorola in 1986 to address quality issues in electronics manufacturing.

Key Points:

- Uses data-driven techniques to reduce variability and improve performance.
- Became a recognized standard for quality improvement and was later trademarked by Motorola.
- Focuses on continuous process improvement rather than just final product quality.

Characteristics of Six Sigma

- **Statistical Quality Measurement:**
  Six Sigma takes its name from the Greek letter sigma (σ), which represents standard deviation in statistics—a key measure used to assess output quality and variability in processes.
- **Structured Methodology:**
  It follows a well-defined, step-by-step approach. The two primary Six Sigma methodologies are **DMAIC** (Define, Measure, Analyze, Improve, Control) used for improving existing processes, and **DMADV** (Define, Measure, Analyze, Design, Verify) used for developing new products or processes.
- **Data-Driven and Analytical:**
  Six Sigma relies heavily on statistical data and analysis, ensuring decisions and improvements are backed by measurable evidence rather than assumptions.

Characteristics of Six Sigma

- **Goal-Oriented Implementation:**
  Six Sigma projects are carefully selected and executed based on specific objectives, such as reducing defects or enhancing process efficiency, ensuring alignment with business goals.
- **Customer-Centric:**
  A core principle of Six Sigma is meeting or exceeding customer expectations. All improvements are designed with customer needs and satisfaction as the top priority.
- **Collaborative Quality Management:**
  Six Sigma emphasizes teamwork and cross-functional collaboration. Organizations must build dedicated teams and cultivate a culture focused on continuous quality improvement.

Trends in software engineering

- **AI and Machine Learning Integration**: AI is now embedded in software tools, enabling code completion (e.g., GitHub Copilot), testing automation, and bug detection.ML models are being integrated into software systems to provide smarter features and analytics.
- **Cloud-Native Development & DevOps**: Companies continue migrating from monoliths to microservices hosted on Kubernetes or serverless platforms. DevOps practices like CI/CD are maturing into **GitOps** and **platform engineering**.
- **Security-First Development (DevSecOps)** : With increasing threats, security is integrated early into the software lifecycle. Shift-left security, secure coding practices, and automated security testing are becoming standards.
- **Remote-First and Distributed Teams** : Collaboration tools, asynchronous workflows, and distributed version control (e.g., Git) are central. **Relevance**: Engineers must be proficient in communication, documentation, and remote tools (e.g., Slack, Notion).

Trends in software engineering

- **Low-Code/No-Code Platforms**: These tools accelerate development for non-technical users and prototyping therefore Engineers might shift to building extensible platforms or APIs for these tools
- **Trend**: Integration with traditional development environments.
- **Cross-Platform Development**: Tools like Flutter, React Native, and Xamarin allow code reuse across platforms. Full-stack and cross-platform development skills are increasingly valuable.
- Growing demand for seamless user experience across mobile, web, and desktop.
- **Software Observability and Reliability:** There is Emphasis on performance monitoring, logging, and alerting for complex systems.
- **Trend**: Use of tools like Prometheus, Grafana, and OpenTelemetry.

Trends in software engineering

- **Domain-Driven Design and Event-Driven Architectures**: Better system modularity and business alignment. Event streaming with Kafka, CQRS, and DDD patterns.
- **Relevance**: Helps manage complexity in large, evolving systems.
- **Package Ecosystem and Software Reusability**: Reuse of third-party libraries is the norm; managing dependencies is critical. Software Composition Analysis (SCA) tools to ensure security and license compliance.
- **Relevance**: Developers must assess risks and choose packages wisely.
- **Ethical Engineering and Sustainability**: Focus on inclusive design, energy-efficient software, and ethical AI use. Regulations (like GDPR, AI Act) shape engineering decisions.
- **Relevance**: Engineers are increasingly accountable for the societal impact of their work.

References

- Acuna, S. T. (2006). *New Trends in Software Process Modelling.* London: World Scientific.
- Marvin. (2001). *Advances in Computers Trends in Computer Engineering .* Merryland: ACADEMIC PRESS
- Six Sigma in Software Engineering | GeeksforGeeks
- http://www. Tpoint Tech

Next Lecture

Digital Forensics and Ethical Hacking