Module 1: Introduction to Software Engineering and SDLC

Class Notes

June 24, 2025

Contents

Module 1: Introduction to Software Engineering and SDLC

Welcome, everyone! Today, we're diving into the fundamentals of Software Engineering. Think of this as laying the groundwork for everything else we'll learn about building great software.

0.1 Introduction to Software Engineering

Let's start by understanding what we mean by "engineering" in this context.

✅What is Engineering?

- Engineering is fundamentally about applying **scientific principles** to design, build, and maintain structures, machines, or systems. The core goal is to **solve real-world problems**.
- It's a blend of creativity, rigorous analysis, structured problem-solving, and systematic thinking. You don't just "build"; you engineer a solution.

✅Why Software Development is not Construction?

This is a common misconception, especially for beginners. Let's clarify why these two are different:

- **Construction Analogy Shortcomings**: In traditional construction (like building a house), you typically work from a fixed blueprint. Changes after the planning phase are costly and often minimal.
- **The Nature of Software**:
  - *Evolving Requirements*: Software requirements are notorious for changing. Customer needs evolve, market conditions shift, and new insights emerge during development. This is a key difference.
  - *Intangibility and Flexibility*: Software is intangible – you can't touch it. It's highly flexible and primarily logic-driven. This contrasts sharply with the rigid, physical nature of construction materials.
  - *Complexity of Logic*: The complexity in software often lies in intricate logic and interdependencies, which can be harder to visualize and manage than physical structures.
- **The "Engineering + Design + Creativity" Mix**: So, while there's a "building" aspect (coding), software development is much more akin to an engineering discipline combined with elements of creative design and continuous refinement. It's not just about following a static plan.

✅What is Software Engineering?

Formally, Software Engineering is:

> The application of engineering principles to **design, develop, test, and maintain software systems** in a systematic, disciplined, and quantifiable way.

The keywords here are *systematic, disciplined, and quantifiable.* This is what elevates software development from a haphazard craft to an engineering discipline.

✅Evolution and Impact of Software Engineering

Software development hasn't always been this structured.

- **1940s–1960s (The Dawn)**: Code was often written by mathematicians or scientists for specific, often academic or military, tasks. There was little to no formal process. It was more art than science.
- **1970s–1980s (The "Software Crisis")**: As software became more complex and critical, projects started failing at an alarming rate. They were over budget, behind schedule, unreliable, and difficult to maintain. This period highlighted the need for better approaches, leading to the emergence of Software Engineering as a distinct field.
- **Present Day**: We now have structured methodologies (like Agile, DevOps), sophisticated tools, emphasis on team collaboration, and a focus on quality assurance throughout the lifecycle.
- **Impact**: The adoption of software engineering principles has led to massive improvements in:
  - **Productivity**: Delivering more, faster.
  - **Quality**: More reliable and robust software.
  - **Cost Efficiency**: Better resource management and less rework.
  - **Reliability**: Software that users can depend on.

0.2 Software Development Life Cycle (SDLC)

Now that we know *what* software engineering is, let's look at *how* we generally approach building software.

✅What is SDLC?

- The Software Development Life Cycle (SDLC) is a **structured process** or framework used for planning, creating, testing, and deploying high-quality software.
- Its main purpose is to break down the complex task of software development into **clear, manageable, and distinct phases**. This helps in better project management, resource allocation, and quality control.

0.2.1 Phases of SDLC

Regardless of the specific SDLC model used (which we'll discuss next), most will involve these general phases, though their order and emphasis might vary:

1. **Requirements Gathering and Analysis**:
   - *Purpose*: To understand and document what the software should do.
   - *Activities*: Involves talking to stakeholders (users, clients, business analysts), defining features, functionalities, constraints, and user expectations. The output is often a Software Requirements Specification (SRS) document.
   - *Key Question*: "What problem are we solving, and for whom?"
2. **Design**:
   - *Purpose*: To create a blueprint for the system. This phase translates requirements into a detailed plan for implementation.
   - *Activities*: Involves decisions on:
     - System Architecture (e.g., microservices, monolithic)
     - Database Design (schemas, relationships)
     - User Interface (UI) and User Experience (UX) layout
     - Module design and their interactions
   - *Output*: Design documents (e.g., high-level design, low-level design).
   - *Key Question*: "How will we build the solution?"
3. **Development (Implementation/Coding)**:
   - *Purpose*: To write the actual code based on the design specifications.
   - *Activities*: Programmers select appropriate programming languages, frameworks, and tools to build the software components. This is where the design is translated into a working system.
   - *Key Question*: "Let's build it!"
4. **Testing**:
   - *Purpose*: To verify that the software works as expected and to identify defects.
   - *Activities*: Involves various levels of testing:
     - **Unit Testing**: Testing individual components or modules.
     - **Integration Testing**: Testing the interaction between integrated modules.
     - **System Testing**: Testing the complete system against requirements.
     - **Acceptance Testing**: Users test the system to ensure it meets their needs.
   - *Key Question*: "Does it work correctly and meet the requirements?"
5. **Deployment**:
   - *Purpose*: To release the tested software to the users or into the production environment.

- *Activities*: May involve installation, configuration, data migration, and user training. Can be a phased rollout or a full release.
- *Key Question*: "How do we get it to the users?"

6. **Maintenance**:
   - *Purpose*: To ensure the software continues to perform as expected after deployment and adapts to changing needs.
   - *Activities*:
     - **Corrective Maintenance**: Fixing bugs discovered post-release.
     - **Adaptive Maintenance**: Modifying the software to work in new or changed environments (e.g., new OS).
     - **Perfective Maintenance**: Enhancing features or improving performance based on user feedback.
     - **Preventive Maintenance**: Making changes to prevent future problems.
   - *Key Question*: "How do we keep it running well and evolving?"

1 Software Development Models

Different projects and teams require different approaches. SDLC models provide these frameworks. Let's look at some common ones in detail.

1.1 ● 1. Waterfall Model

The classic, linear approach.

- **Structure**:
  - Linear, sequential flow. Like a waterfall, progress flows steadily downwards.
  - Each phase must be fully completed before the next one begins.
  - *Sequence*: Requirements → Design → Implementation → Testing → Deployment → Maintenance.
- **Advantages**:
  - Simple, easy to understand and use.
  - Clearly defined stages, deliverables, and milestones.
  - Good for projects where requirements are very well understood and fixed from the start.
  - Strong emphasis on documentation.
- **Disadvantages**:
  - Highly inflexible; no room for changes once a phase is completed without significant rework.

    - Issues are often discovered late, typically during testing, making them expensive to fix.
    - Not suitable for complex, long-term, or evolving projects where requirements are uncertain.
    - Working software is not produced until late in the lifecycle.
- **When to Use**:
    - Projects with clear, stable, and well-documented requirements.
    - Short-term projects or small maintenance updates.
    - Situations where the technology stack is well-understood.

1.2 ● 2. Iterative and Incremental Model

Build a little, test it, get feedback, build some more.

- **Structure**:
    - Develops software in small, manageable parts called increments.
    - Each increment goes through its own mini-cycle of requirements, design, implementation, and testing (an iteration).
    - Each iteration builds upon the previous one, adding new functionality or refining existing features.
    - *Example Cycle*:
        - Iteration 1: Plan → Design (Core) → Implement (Core) → Test (Core) → Deliverable 1
        - Iteration 2: Plan (New Features) → Design (Additions) → Implement (Additions) → Test (Integration) → Deliverable 2
        - ...and so on.
- **Advantages**:
    - A working version of the software is available early in the development lifecycle.
    - More flexible and easier to manage risks compared to Waterfall. Changes can be accommodated between iterations.
    - Customer feedback can be incorporated continuously, leading to a product that better meets user needs.
    - Easier to test and debug smaller increments.
- **Disadvantages**:
    - Requires more comprehensive planning and robust design for the overall system architecture to ensure increments integrate well.
    - Can involve more documentation if not managed carefully.

  - If the initial iterations are poorly designed, it may lead to architectural issues later (technical debt).
  - Total cost and timeline might not be clear at the very beginning.

- **When to Use**:
  - Medium to large projects where features can be developed and delivered incrementally.
  - Projects where requirements are expected to evolve.
  - When early user feedback is crucial.
  - Projects that can benefit from early prototypes or proofs-of-concept.

1.3 ● 3. V-Shaped Model (Verification and Validation Model)

A more disciplined Waterfall with early test planning.

- **Structure**:
  - An extension of the Waterfall model where testing activities are planned in parallel with the corresponding development phase.
  - Forms a V-shape, where the left side represents development (Verification phases) and the right side represents testing (Validation phases).
  - Each development stage has a corresponding testing phase.
  - *Mapping*:
    * Requirements Analysis $\longleftrightarrow$ Acceptance Testing
    * System Design $\longleftrightarrow$ System Testing
    * Architectural/High-Level Design $\longleftrightarrow$ Integration Testing
    * Module/Low-Level Design $\longleftrightarrow$ Unit Testing
    * Coding (the bottom point of the V)
- **Advantages**:
  - Strong emphasis on verification (Are we building the product right?) and validation (Are we building the right product?).
  - Early planning for testing activities, which can help detect defects earlier in the corresponding development phase.
  - Clear deliverables and review processes at each stage.
  - Good for projects where quality and reliability are paramount.
- **Disadvantages**:
  - Still quite rigid and inflexible, similar to Waterfall. Changes in requirements can be costly.

  - No early prototypes are produced. Working software is available only at the end.
  - If issues are found late in a testing phase, it can be expensive to go back and fix the corresponding development phase.
- **When to Use**:
  - Projects requiring high reliability, such as medical software, aviation systems, or critical infrastructure.
  - When requirements are well-defined and stable.
  - Projects where extensive testing and quality assurance are non-negotiable.

1.4 ● 4. RAD (Rapid Application Development)

Focus on speed, prototypes, and user feedback.

- **Structure**:
  - Emphasizes rapid prototyping and iterative development with minimal planning upfront.
  - Users are heavily involved throughout the process, providing continuous feedback on prototypes.
  - Phases are often:
    1. **Requirements Planning**: High-level requirements.
    2. **User Design**: Users interact with analysts to build prototypes. This is iterative.
    3. **Construction**: Development based on prototypes and feedback, often using automated tools or reusable components.
    4. **Cutover (Deployment)**: Implementation, testing of the final system, and user training.
- **Advantages**:
  - Fast development cycle with quick feedback loops.
  - Encourages strong user involvement, leading to higher user satisfaction.
  - Can significantly reduce development time if scope is well-contained.
  - Functionality can be demonstrated early.
- **Disadvantages**:
  - Not suitable for large, complex, or highly scalable systems.
  - Requires highly skilled developers and very committed users who can provide timely feedback.
  - May compromise on overall system architecture or quality if speed is overemphasized.

  - Can be hard to manage if scope creep is not controlled.
- **When to Use**:
  - Projects needing quick delivery and where requirements can be modularized.
  - Small-to-medium sized systems with clear business objectives and available users for feedback.
  - When there's a need for rapid prototyping to validate concepts.

1.5 ● 5. Prototyping Model

Build a mock-up to clarify needs, then build the real thing.

- **Structure**:
  - Focuses on creating a prototype (an early, incomplete but functional version) of the software.
  - This prototype is shown to users to gather feedback and refine requirements.
  - The cycle repeats: Build Prototype → User Evaluation → Refine Requirements → (Potentially) New Prototype.
  - Once requirements are clear, the actual system is developed (often from scratch or by evolving the final prototype).
- **Advantages**:
  - Helps to understand and clarify unclear, vague, or incomplete requirements early on.
  - Encourages user feedback and involvement, leading to a better final product.
  - Reduces the risk of building the wrong system.
  - Can uncover missing functionalities.
- **Disadvantages**:
  - Users might mistake the prototype for the final system and have unrealistic expectations about its completeness or robustness.
  - Can lead to an "analyze-prototype-code" loop that delays final development if not managed.
  - Costly if prototyping is overused or if developers spend too much time on throw-away prototypes.
  - The final system might be poorly structured if the prototype is simply patched up instead of being properly re-engineered.
- **When to Use**:
  - Projects where requirements are evolving, not fully understood, or ambiguous at the start.

  - When developing systems with a significant user interface component, as prototypes help visualize interactions.
  - For innovative products where user reactions are hard to predict.

1.6 ● 6. Spiral Model (Boehm's Spiral)

Risk-driven, iterative approach for large, complex projects.

- **Structure**:
  - Combines elements of design, prototyping, and risk management in iterative stages (spirals).
  - Each spiral or cycle passes through four main quadrants:
    1. **Planning**: Determine objectives, alternatives, and constraints.
    2. **Risk Analysis**: Identify and resolve risks. Prototypes may be built here.
    3. **Engineering (Development & Test)**: Develop and test the next level of the product.
    4. **Evaluation (Customer Review)**: Assess the results of the spiral and plan for the next one.
  - The project grows with each spiral, adding more detail and functionality.
- **Advantages**:
  - Strong emphasis on risk analysis and mitigation, making it suitable for high-risk projects.
  - Flexible; allows for changes and additions of functionality during development.
  - Users see the system early (through prototypes in risk analysis).
  - Can accommodate new technologies or changes in requirements.
- **Disadvantages**:
  - Complex to manage; requires expertise in risk assessment.
  - Can be expensive due to the repeated phases and detailed risk analysis.
  - May not be suitable for small or low-risk projects due to its overhead.
  - Success heavily depends on the effectiveness of the risk analysis phase.
- **When to Use**:
  - Large, complex, and high-risk projects.
  - Mission-critical systems (e.g., military, aerospace, critical research).
  - Projects where significant changes are expected during the development lifecycle.
  - When research and development are involved, and the path forward is not entirely clear.

1.7 ● 7. Agile Model

Modern, flexible, collaborative, and results-focused.

- **Structure**:
  - An umbrella term for a set of principles and practices that are iterative and collaborative.
  - Development occurs in short cycles called iterations or sprints (typically 1-4 weeks).
  - Cross-functional teams work closely with customers/stakeholders continuously.
  - Emphasizes adaptability and responding to change.
- **Core Principles (from the Agile Manifesto)**:
  - **Individuals and interactions** over processes and tools.
  - **Working software** over comprehensive documentation.
  - **Customer collaboration** over contract negotiation.
  - **Responding to change** over following a plan.
- **Advantages**:
  - Highly adaptive to changing requirements, even late in development.
  - Focuses on delivering working software frequently, providing tangible progress.
  - Promotes strong collaboration between the development team and stakeholders.
  - Leads to higher customer satisfaction due to continuous involvement and feedback.
  - Improved quality through continuous integration and testing.
- **Disadvantages**:
  - Requires strong team coordination, experienced members, and a high degree of commitment.
  - Can be challenging to predict the total effort or timeline at the outset due to evolving scope.
  - Documentation can sometimes be neglected if not carefully managed.
  - Less suitable for projects with fixed scope and budget from the very start, or where extensive upfront design is critical.
- **When to Use**:
  - Projects where requirements are expected to change frequently or are not fully known at the start.
  - Teams that can work closely with stakeholders and adapt to feedback.
  - Environments that value rapid delivery of functional software.

- Innovative projects where exploration and learning are part of the process.

Figure 1: Conceptual Agile/Scrum Flow (Placeholder - replace image2.png)

1.8 • 8. SCRUM (a type of Agile Framework)

A popular, structured framework for implementing Agile.

- **Structure**:
  - Work is done in fixed-length iterations called **Sprints** (usually 2–4 weeks).
  - Each Sprint aims to deliver a potentially shippable increment of the product.
- **Key Roles**:
  - **Product Owner**: Represents the customer/stakeholders. Defines and prioritizes features in the Product Backlog. Responsible for the "what".
  - **Scrum Master**: Facilitates the Scrum process, removes impediments for the team, and ensures Scrum practices are followed. A servant-leader.
  - **Development Team**: A self-organizing, cross-functional group of professionals who do the actual work of building the product increment (design, code, test).
- **Key Events/Artifacts**:
  - **Product Backlog**: A prioritized list of all desired features/requirements for the product.
  - **Sprint Planning**: Team selects items from Product Backlog to work on in the upcoming Sprint, creating a Sprint Backlog.
  - **Daily Scrum (Stand-up)**: A short (e.g., 15-min) daily meeting for the Development Team to synchronize and plan for the next 24 hours. (What did I do yesterday? What will I do today? Any impediments?)
  - **Sprint Review**: At the end of the Sprint, the team demonstrates the completed work (increment) to stakeholders and gets feedback.
  - **Sprint Retrospective**: After the Sprint Review, the team reflects on the past Sprint to identify what went well, what could be improved, and how to make changes for the next Sprint.
- **Advantages**:
  - Delivers working software frequently and fast.
  - Improves team communication, collaboration, and transparency.
  - Encourages accountability and continuous improvement.
  - Highly adaptable to changes.
- **Disadvantages**:

  - Requires discipline, experienced team members, and strong commitment from all roles.
  - Can be chaotic if not properly managed or if the Scrum Master is ineffective.
  - Scope creep can be an issue if the Product Owner doesn't manage the Product Backlog well.
  - Not ideal for projects with very stable requirements and a fixed deadline/budget that requires extensive upfront planning.

- **When to Use**:
  - Projects with frequent changes, evolving requirements, and tight deadlines.
  - Best suited for modern software development, product teams, and startups.
  - When rapid feedback and continuous delivery of value are critical.