CSE 6324

Advanced Topic in Software Engineering

Farnaz Farahanipad

Objectives

- Understand the Role of Implementation in the SDLC.
- Key Concepts and Activities: Learn the technical aspects of writing clean, efficient code, debugging, and testing.
- Introduction to Coding Tools & Environments: Familiarize with Integrated Development Environments (IDEs), version control systems, and testing frameworks.
- Best Practices in Coding: Emphasize modularity, code reuse, and adherence to coding standards.

Overview of the Implementation Phase

- **Definition**: The process of translating system design into executable code.
- **Objective**: Create working software that meets functional and non-functional requirements.
- **Deliverables**:
  - Source code in the chosen programming language.
  - Modularized software components.
  - Initial unit test cases to ensure functionality.

Key Activities in the Implementation Phase

- **Coding**: Writing source code based on detailed design specifications.
- **Debugging**: Using tools like GDB, Xdebug, and breakpoints to identify and fix code defects.
- **Unit Testing**: Writing automated unit tests using frameworks like JUnit, PyTest.
- **Code Integration**: Merging different modules of code using tools like Jenkins or Git.

Technical Details: Software Modularity

- What is Modular Programming?
  - Dividing the program into distinct, self-contained modules or functions.
  - Each module should perform a single task (single responsibility principle).

- Advantages:
  - Improves code maintainability.
  - Facilitates testing and debugging.
  - Enables code reuse across projects.

Version Control in Depth

- **Git Basics:**
  - git init, git add, git commit, git push: Basic Git commands to manage code.
  - Branching and Merging: Use branches to isolate development and merge them back into the main branch after code reviews.
- **Best Practices for Git:**
  - Commit Frequently: Make small, meaningful commits.
  - Write Descriptive Commit Messages: Help other developers understand the purpose of changes.

Coding Standards and Code Quality

- **Adherence to Coding Standards:**
  - Use linting tools like ESLint for JavaScript, Pylint for Python, and Checkstyle for Java.
  - Enforce naming conventions (camelCase, snake_case) and consistent formatting.

- **Code Quality Metrics:**
  - Cyclomatic complexity: Measures code complexity.
  - Code coverage: The percentage of code tested by automated tests.

Code Review: A Critical Process

- **Peer Review Process:**
  - Ensure code adheres to design principles, and meets performance standards.
- **Tools for Code Review:**
  - GitHub pull requests, Bitbucket, and Gerrit for collaborative code review.
- **Focus Areas:**
  - Code readability, maintainability, and optimization.

Common Debugging Techniques

- **Breakpoint Debugging**: Pause code execution and inspect variable states in IDEs like PyCharm or Visual Studio.
- **Print Statements and Logging:** Use logging libraries (Log4j for Java, Python's logging module) to trace errors in production environments.
- **Memory Profiling Tools:** Detect memory leaks or inefficient resource usage with tools like Valgrind or VisualVM.

Automated Testing and CI/CD

- **Unit Testing Frameworks:**
  - JUnit (Java), PyTest (Python), NUnit (.NET): Automate tests for individual functions or classes.
- **Integration Testing:**
  - Testing how different components interact, using tools like Selenium for web apps or Postman for API testing.
- **Continuous Integration (CI):**
  - Automating builds and tests after every commit using Jenkins, GitLab CI, or Travis CI.

Error Handling and Exception Management

- **Importance of Error Handling:**
  - Prevent application crashes, ensure graceful recovery, and provide meaningful error messages.

- **Exception Handling Mechanisms:**
  - Try-Catch Blocks: In Java, Python, C#, etc.
  - Custom exceptions: Create user-defined exceptions for specific error cases.

- **Best Practices:**
  - Log exceptions with enough context for debugging.
  - Avoid silent failures—inform users or logs of critical issues.

Refactoring Code for Performance and Maintainability

- **What is Refactoring?**
  - Modifying code without changing its external behavior to improve structure, readability, and performance.
- **Common Refactoring Techniques:**
  - Extract Method: Breaking large methods into smaller, more manageable ones.
  - Rename Variables: To provide more descriptive names.
  - Eliminate Redundancy: Remove duplicated code by creating reusable functions.
- **Tools for Refactoring:**
  - Refactoring tools within IDEs (e.g., IntelliJ, Visual Studio Code).

Documentation in the Coding Phase

- **Types of Documentation:**
  - API Documentation: Swagger for REST APIs, Javadoc for Java.
  - Code Comments: Proper inline comments to explain logic and reasoning behind complex code sections.
  - README Files: Provide setup instructions, dependencies, and how to contribute to the project.

Handling Legacy Code

- **What is Legacy Code?**
  - Outdated code that’s hard to maintain but still critical to the system.
- **Challenges with Legacy Code:**
  - Lack of documentation.
  - High risk of introducing bugs when making changes.
- **Strategies for Managing Legacy Code:**
  - Incremental refactoring to modernize parts of the system.
  - Introduce unit tests to cover existing functionality.
  - Use dependency injection to decouple components.

Final Thoughts on the Coding Phase

- **Focus on Quality:** Clean code, consistent standards, and thorough testing are key to successful implementation.
- **Iterate and Improve:** Always refactor and optimize code for performance and maintainability.
- **Collaboration is Crucial**: Engage in regular code reviews and communication within the development team.

Questions: