**Software Engineering**

**Name: Mark Mureithi**

**Reg No: SCT221-C004-0038/2023**

1. **Cost estimation models in Software engineering**

A software cost estimating methodology is an indirect metric used by software professionals to estimate project costs. They're utilized for a variety of things. They are used to predict the financial and resource requirements for developing a software project. These models help project managers, software developers, and stakeholders make informed decisions about budgeting, scheduling, and resource allocation. It contains the following items −

1. Budgeting − the most desired capability is for the overall estimate to be correct. As a result, the first focus is on estimating the software product's budget.
2. Analysis of tradeoffs and risks − the ability to expose the cost and schedule sensitivity of software project choices is a significant added feature (scoping, staffing, tools, reuse, etc.).
3. Controlling and planning the project − another option is to break down costs and schedules by component, stage, and activity. Investment analysis for software enhancements Tools, reuse, and process maturity are all beneficial to the software development process.

There are several cost estimation models, each with its own approach and underlying assumptions. Here are some of the most commonly used software cost estimation models:

1. **COCOMO (Constructive Cost Model):** COCOMO is one of the most widely used models for software cost estimation. It was developed by Barry Boehm. It comes in three variations:
2. Basic COCOMO: Estimates effort based on the size of the software.
3. Intermediate COCOMO: Includes additional factors like personnel, software reliability, etc.
4. Detailed COCOMO: Considers a broader set of parameters and has more detailed calculations.
5. **Expert Judgment:** Expert judgment is one of the simplest and often the most readily available cost estimation techniques. It relies on the expertise of experienced professionals who use their knowledge and judgment to estimate the costs involved in a software project.
6. **Function Points (FP):** Function points are a measurement of software functionality based on the user's perspective. They provide a way to estimate the size and complexity of a software project, which can then be used to estimate costs.
7. **Use Case Points (UCP):** UCP is a cost estimation model that focuses on use cases to determine the size and complexity of a software project. It considers aspects such as actors, use cases, and transaction volumes.
8. **Parametric Models:** Parametric models use mathematical relationships to estimate costs based on historical data and project parameters. Examples include PRICE Systems, SEER-SEM, and TruePlanning.
9. **Expert Systems:** Expert systems are computer-based systems that incorporate expert knowledge and rules to estimate software project costs. They can be customized for specific organizations and domains.
10. **Machine Learning-Based Models:** With advancements in machine learning, there has been an emergence of models that use historical project data and machine learning algorithms to estimate software development costs.
11. **Analogous Estimation:** This approach involves comparing the current project to similar past projects and using their costs as a reference. It assumes that projects with similar characteristics will have similar costs.
12. **Bottom-Up Estimation:** This approach involves estimating the costs for individual components, such as modules or features, and then aggregating them to determine the overall project cost. It is useful for detailed cost breakdowns.
13. **Top-Down Estimation**: In contrast to bottom-up estimation, top-down estimation starts with an overall project cost and then allocates costs to various project components. It's useful when there's a high-level budget constraint.

The choice of which cost estimation model to use depends on factors such as project size, complexity, available data, and the organization's preferences and expertise.

1. **Discuss software metrics under the following topics ; maintainability, documentation, complexity, reliability, availability, comprehensibility, walkthroughs, reviews and inspection, dynamic testing, traceability matrices.**
2. **Maintainability**

These metrics focus on the ease with which software can be maintained, enhanced, or fixed after its initial development. Some metrics for maintainability include cyclomatic complexity, code churn and code duplication.

1. Cyclomatic Complexity: Quantifies the complexity of a program by counting the number of linearly independent paths.
2. Code Churn: Measures the frequency of code changes. High churn might indicate potential maintainability issues.
3. Code Duplication: Measures the amount of duplicated code. High duplication can impact maintainability negatively.
4. **Documentation**

Documentation metrics assess the quality, coverage, and usefulness of documentation associated with the software. Metrics could include the number of pages, coverage percentage, the number of diagrams, and the frequency of updates to documentation. Some of documentation metrics include;

1. Documentation Coverage: Measures the extent to which the codebase is documented. It can be expressed as a percentage of documented components.
2. Documentation Consistency: Checks for consistency in the documentation, including adherence to coding standards, naming conventions, and comments.
3. Documentation Clarity: Assesses how well the documentation conveys the purpose and functionality of the software.
4. **Complexity**

Complexity metrics gauge the intricacy of the software system. Common metrics include cyclomatic complexity, Halstead complexity measures, and maintainability index. These metrics help in understanding the code's complexity and its impact on maintainability and reliability.

1. Cyclomatic Complexity: Measures the complexity of the control flow in the code, helping to identify areas that may be error-prone or difficult to understand.
2. Halstead Complexity Measures: These metrics include various measures related to the number of operators and operands used in the code.
3. **Reliability**

Reliability metrics evaluate the software's ability to perform its functions without failure. Metrics include Mean Time Between Failures (MTBF), Failure Rate, Fault Density, and Failure Density. These metrics help in predicting and ensuring the software's stability and uptime.

1. Failure Rate: Measures the frequency of software failures over a specific period.
2. Mean Time Between Failures (MTBF): The average time between software failures.
3. Failure Density: Measures the number of failures per unit of time or code size.
4. **Availability**

Availability metrics measure the extent to which the software system is operational and accessible. Metrics might include system uptime percentage, mean time to recover (MTTR), and mean time between failures (MTBF).

1. System Uptime: Measures the amount of time the software is available and operational.
2. Mean Time To Recover (MTTR): Measures the average time to restore service after a failure.
3. Mean time between failures (MTBF): The duration during which the software is not available. Availability is often expressed as a percentage of uptime.
4. **Comprehensibility**

Comprehensibility metrics assess how easily a software system can be understood. This could include metrics related to readability, such as average sentence length, the Flesch-Kincaid Grade Level, and the Coleman-Liau index. Some of Comprehensibility metrics include;

1. Readability Score: Evaluates how readable the code is by assessing factors like variable naming, code structure, and commenting.
2. Maintainability Index: While primarily a maintainability metric, a higher maintainability index can also indicate better comprehensibility.
3. Documentation Quality: The quality and clarity of code comments and inline documentation can impact comprehensibility.
4. **Walkthroughs, Reviews, and Inspections**

Metrics for these processes include the number of defects found, defect density, inspection effort, and inspection rate. These metrics help evaluate the effectiveness of the review processes in identifying issues and ensuring software quality. Some example of its metrics include;

1. Review Defect Density: Measures the number of defects or issues identified during code reviews per lines of code or other metrics.
2. Review Efficiency: Evaluates how effective and efficient the review process is, including the time and resources required for reviews.
3. Inspection Findings: Counts the number and severity of defects found during formal code inspections.
4. **Dynamic Testing**

Dynamic testing metrics assess the effectiveness and coverage of testing efforts. Metrics include code coverage (statement, branch, path), test execution time, and defect detection rate.

1. Code Coverage: Measures the percentage of code exercised by a test suite.
2. Defect Discovery Rate: Quantifies the number of defects discovered during testing.
3. **Traceability Matrices**

Traceability matrices metrics measure the completeness and accuracy of traceability links between requirements, design, code, and test cases. Metrics might include the percentage of traceable requirements and traceability gap.

1. Requirement Traceability: Measures the extent to which each requirement is traced forward to design and implementation.
2. Traceability Gap: Identifies areas where requirements lack corresponding elements in the development process.