软件指标与质量分析
Software Metrics & Quality Analysis

By, Fahad Sabah

TFSE_Fall2025_北京工业大学，中国

TFSE_Fall2025_Beijing University of Technology, China

CONTENTS

软件指标与质量分析
Software Metrics & Quality Analysis

01
- 项目管理与经济学。
- Introduction to Software Quality.

02
- 为开源项目计算指标。
- Calculating metrics for open source projects.

1. 项目管理与经济学
1. Introduction to Software Quality

1.1 The Importance of Software Quality
- *Historical software failures and their costs*
- *Economic impact of quality in software projects*
- *Quality vs. Speed: The eternal trade-off*

1.2 Evolution of Software Quality Models
- *Early models: McCall, Boehm, FURPS*
- *ISO 9126 to ISO 25010 transition*
- *Industry standards (CMMI, SPICE)*

1.1 软件质量的重要性
- 历史软件故障及其成本
- 软件项目质量的经济影响
- 质量与速度：永恒的权衡

1.2 软件质量模型的演变
- 早期型号：McCall、Boehm、FURPS
- ISO 9126 向 ISO 25010 过渡
- 行业标准（CMMI，SPICE）

1.1 软件质量的重要性
1.1 The Importance of Software Quality

- *Historical software failures and their costs*

1: Ariane 5 Rocket Failure (1996)

What happened?

- *A software bug in the guidance system caused the $500 million rocket to self-destruct 40 seconds after launch*
- *The bug was in code reused from Ariane 4 without proper testing for new conditions*

The Ariane 5 Rocket Disaster

- On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its liftoff from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. http://ima.umn.edu/~arnold/disasters/ariane.html

1.1 软件质量的重要性
1.1 The Importance of Software Quality

- *Historical software failures and their costs*

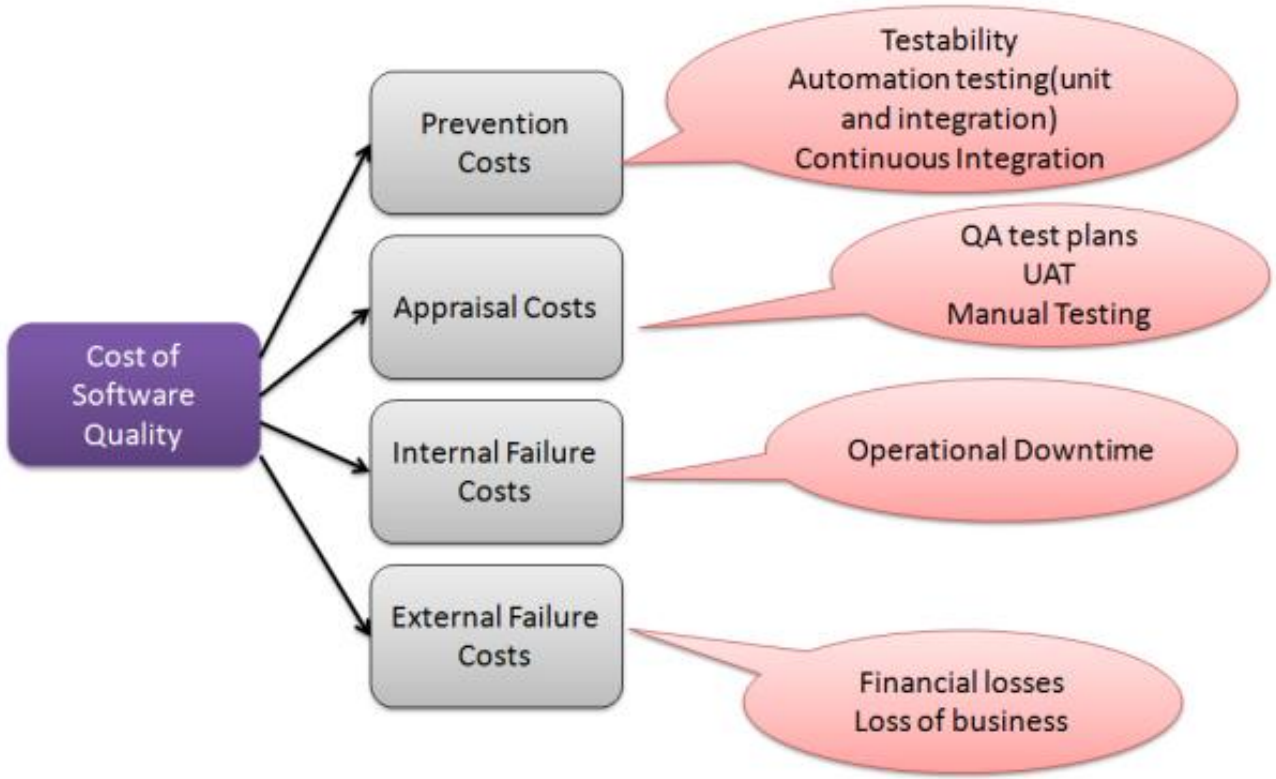
2: Knight Capital Group (2012)

What happened?

- *Software deployment error caused $440 million loss in 45 minutes*
- *Old code reactivated, processing millions of unintended trades*

1.1 软件质量的重要性
1.1 The Importance of Software Quality

❑*Economic Impact of Quality in Software Projects*



1.1 软件质量的重要性
1.1 The Importance of Software Quality

- *Economic Impact of Quality in Software Projects*

There is a saying;

The quality costs till development phase but later it becomes cheaper

```
# High-quality LMS design upfront
class HighQualityLMS:
    def __init__(self):
        self.unit_tests = True      # $10,000 investment
        self.code_reviews = True    # $5,000 investment
        self.documentation = True   # $3,000 investment
        self.ux_testing = True      # $7,000 investment
        # Total prevention cost: $25,000
```

1.1 软件质量的重要性
1.1 The Importance of Software Quality

```
# Testing and verification costs
class AppraisalCosts:
   def __init__(self):
      self.manual_testing = 200_hours * $50 = $10,000
      self.automated_tests = 100_hours * $75 = $7,500
      self.security_audit = $5,000
      self.performance_testing = $3,000
      # Total appraisal cost: $25,500
```

1.1 软件质量的重要性
1.1 The Importance of Software Quality

```
# Bugs found during development
internal_failures = {
   'rework_hours': 500,          # 500 hours at $75/hour = $37,500
   'retesting_hours': 200,       # 200 hours at $50/hour = $10,000
   'delayed_launch_penalty': 0   # Caught early, no delay
   # Total internal failure cost: $47,500
}
```

1.1 软件质量的重要性
1.1 The Importance of Software Quality

```
# Real-world costs from a low-quality LMS
external_failures = {
    'support_calls': {
        'monthly_volume': 1000,
        'cost_per_call': $25,
        'annual_cost': 1000 * 25 * 12 = $300,000
    },
    'data_loss': {
        'incidents_per_year': 5,
        'recovery_cost_per_incident': $10,000,
        'annual_cost': $50,000
    },
    'lost_members': {
        'members_lost': 500,
        'annual_value_per_member': $100,
        'annual_revenue_loss': $50,000
    },
    'system_downtime': {
        'hours_per_year': 48,
        'cost_per_hour': $2,000,
        'annual_cost': $96,000
    }
    # Total external failure cost: $496,000/year!
}
```

1.1 软件质量的重要性
1.1 The Importance of Software Quality

- *Economic Impact of Quality in Software Projects*

TOTAL COST OF POOR QUALITY LMS:		*TOTAL COST OF HIGH-QUALITY LMS:*	
Prevention (skipped):	*$0*	*Prevention:*	*$25,000*
Appraisal (minimal):	*$5,000*	*Appraisal:*	*$25,500*
Internal Failures (high):	*$75,000*	*Internal Failures (low):*	*$10,000*
External Failures (massive):	*$496,000*	*External Failures (minimal):*	*$5,000*
TOTAL 1st Year:	*$576,000*	*TOTAL 1st Year:*	*$65,500*

QUALITY DIVIDEND: $576,000 - $65,500 = $510,500 SAVINGS

Return on Quality Investment: 780% in first year!

1.1 软件质量的重要性
1.1 The Importance of Software Quality

❑*Economic Impact of Quality in Software Projects*

```
# Economic impact on library operations
library_impact = {
  'low_quality_system': {
    'annual_operating_cost': 150000,  # High support, fixes
    'member_satisfaction': 60%,       # Frequent issues
    'staff_productivity': 70%,        # Workarounds needed
    'data_accuracy': 85%,             # Errors in records
    'annual_member_growth': -5%,      # Losing members
  },
  'high_quality_system': {
    'annual_operating_cost': 80000,   # Lower support needs
    'member_satisfaction': 95%,       # Reliable service
    'staff_productivity': 120%,       # Efficient workflows
    'data_accuracy': 99.9%,           # Trustworthy data
    'annual_member_growth': +15%,     # Attracting users
  }
}

# Net annual benefit:
# ($150,000 - $80,000) + (15% growth vs -5% loss) + productivity gains
# = $70,000 direct savings + community value + staff morale
```

1.1 软件质量的重要性
1.1 The Importance of Software Quality

❑*Quality vs. Speed: The Eternal Trade-off*

SPEED-FOCUSED DEVELOPMENT (1 month)
==========

Delivered: 12 features
Initial Velocity: 12 features/month
Month 2 Velocity: 8 features/month (25% slowdown)
Month 3 Velocity: 5 features/month (58% slowdown)
Month 6: Complete rewrite needed

Total Features (6 months): 12 + 8 + 5 + 3 + 2 + 0 = 30 features
Maintenance Cost: 70% of developer time
User Satisfaction: ⭐ ⭐ ☆ ☆ ☆

QUALITY-FOCUSED DEVELOPMENT (1 month)
==========

Delivered: 6 features
Initial Velocity: 6 features/month
Month 2 Velocity: 7 features/month (16% increase)
Month 3 Velocity: 8 features/month (33% increase)
Month 6: Stable, scalable system

Total Features (6 months): 6 + 7 + 8 + 9 + 10 + 11 = 51 features
Maintenance Cost: 20% of developer time
User Satisfaction: ⭐ ⭐ ⭐ ⭐ ⭐

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

- *Early models: McCall, Boehm, FURPS*
- *ISO 9126 to ISO 25010 transition*
- *Industry standards (CMMI, SPICE)*

- 早期型号：McCall、Boehm、FURPS
- ISO 9126 向 ISO 25010 过渡
- 行业标准（CMMI，SPICE）

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

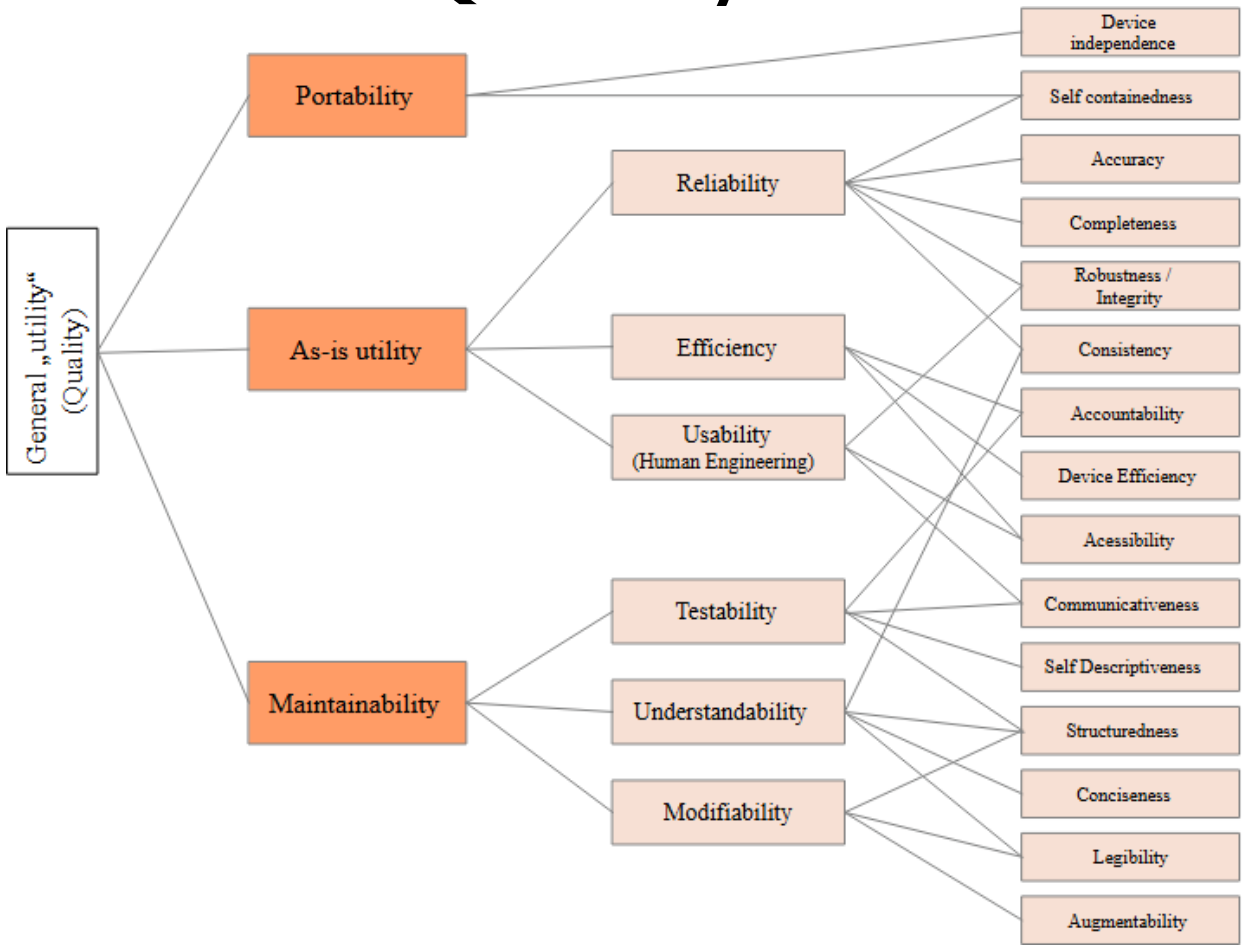
Authors / Name	Year	Summary	Quality attributes
Boehm	1976	Hierarchical model. Top-level qualities: utility, maintainability, portability.	24
McCall	1977	Hierarchical model. Top-level areas: operation, revision, transition.	11
ISO-9126	1991	Hierarchical model. No safety, security underrated, disputable terminology	27
FURPS	1992	Single level with functionality, usability, reliability, performance, supportability. Lacks operational qualities and safety	6
IBM FURPS+	1999	Add lots of sub-characteristics to FURPS, addressing requirements in general.	30

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

Authors / Name	Year	Summary	Quality attributes
VOLERE	1999	Integrated in sophisticated template for requirements. Combines qualities and constraints	8
ISO-25010	2011	Supersedes ISO-9126. Hierarchical model with 8 top-level qualities. Adds security as top-level.	32
ISO-25010, draft 2022	2022	Proposal to add safety and change a number of terms and definitions.	39

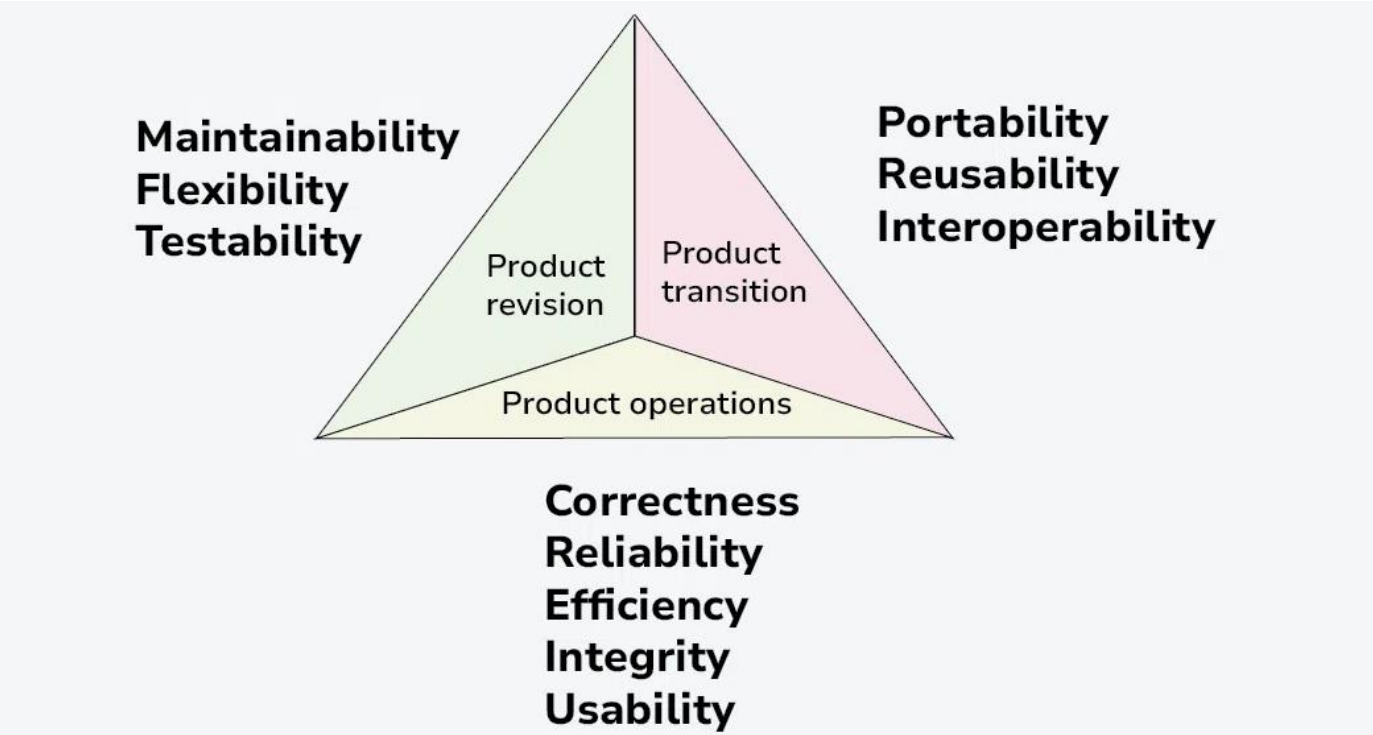
1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

❑*Boehm's Model (1976) - The "Utility" Model*



1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

❑*McCall's Model (1977) - The "Triangle" Model*



1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

Practical McCall Evaluation Table for LMS:

Factor	Question for LMS	Good Example	Bad Example
Correctness	Are late fees calculated correctly?	(days_late * 0.50) = $2.50	(days_late * 50) = $250.00
Reliability	Does system stay up during peak hours?	99.9% uptime, handles 100 concurrent users	Crashes when 20+ users login
Usability	Can new librarians learn system in 1 day?	Intuitive icons, logical workflow	Requires 50-page manual
Maintainability	How long to fix a checkout bug?	2 hours (well-documented code)	2 weeks (spaghetti code)
Portability	Can it run on library's existing computers?	Runs on Windows 10, Linux, MacOS	Requires Windows Server 2019 only

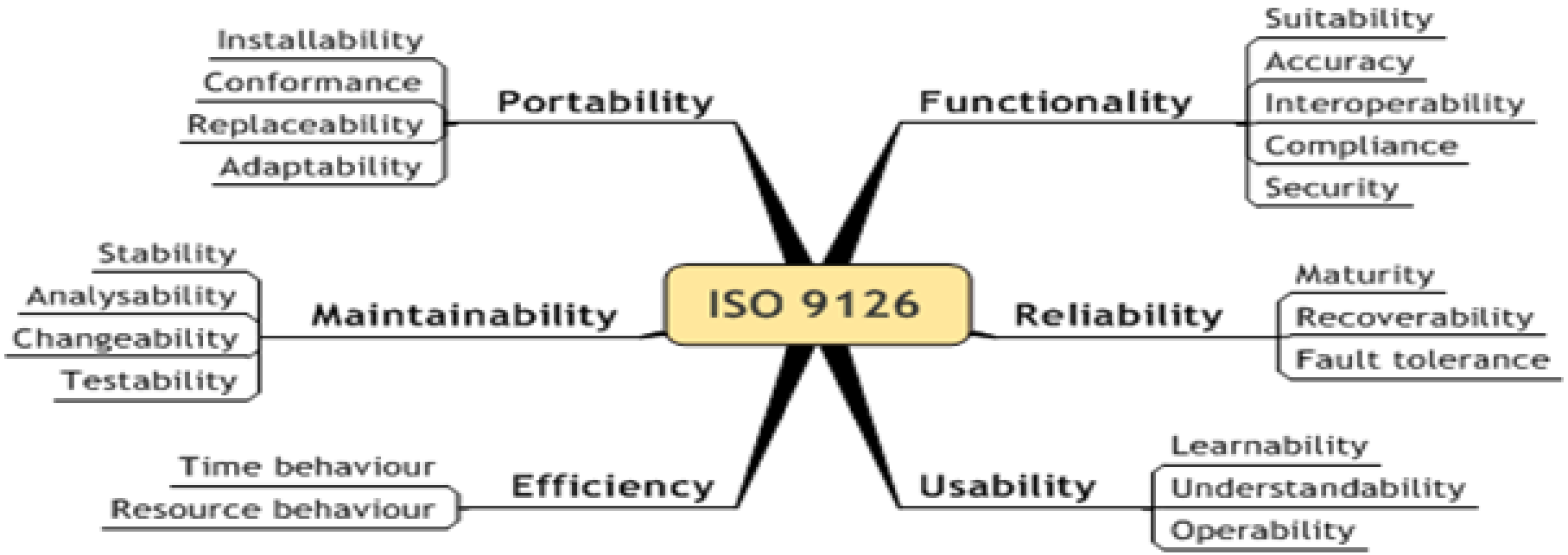
1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

❑*FURPS Model (HP, 1987) - The "Practical" Model*

- ❑ **Functionality**
- ❑ **Usability** is concerned with characteristics such as aesthetics and consistency in the user interface.
- ❑ **Reliability** is concerned with characteristics such as availability (the amount of system “uptime”), accuracy of system calculations, and the system’s ability to recover from failure.
- ❑ **Performance** is concerned with characteristics such as throughput, response time, recovery time, start-up time, and shutdown time.
- ❑ **Supportability** is concerned with characteristics such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability.

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

- *ISO 9126*



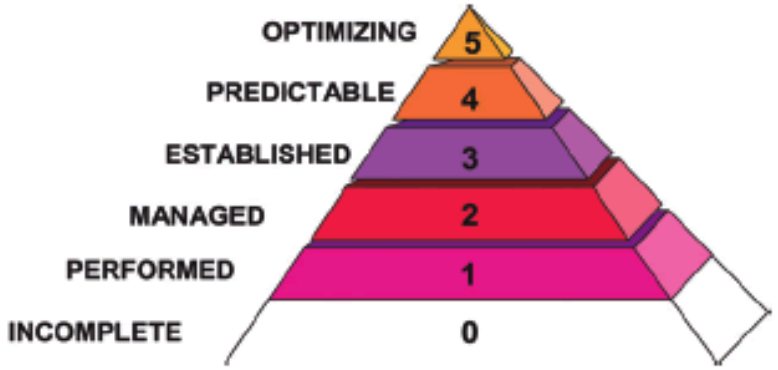
1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

❑*ISO 25010*

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	**PERFORMANCE EFFICIENCY**	**COMPATIBILITY**	**INTERACTION CAPABILITY**	**RELIABILITY**	**SECURITY**	**MAINTAINABILITY**	**FLEXIBILITY**	**SAFETY**
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

❑*Industry standards (CMMI, SPICE)*



Parameters Of Comparison	CMMI	ASPICE
Launched	CMMI was launched as the first version of CMM in the year 2002.	ASPICE was developed as the variant of ISO that was launched is in the year 2001.
Full form	The expanded form of CMMI is Capability Maturity Model Integration.	The expanded form of ASPICE is Automotive Software Performance Improvement and Capability dEtermination.
Focus	CMMI is not only focused on software development but also gives importance to vehicle systems and their safety.	ASPICE is focused on the different stages of development and functionality of the software; generally associated with the automotive industry.
Factors	Safety is the prime factor that is concerned by CMMI, other aspects like cost and schedule are not concerned.	The main factors and aspects that are concerned by ASPICE are cost, schedule, and safety of the product.
Readability	There 5 stages and 3 criteria are provided by CMMI makes it very easy and convenient to understand.	ASPICE is not classified like CMMI which makes it a bit hard to understand.

1.2 软件质量模型的演变
1.2 Evolution of Software Quality Models

Quality Model Selection Matrix for LMS:

Library Type	Primary Model	Why	Key Focus Areas
Small Public Library	FURPS+	Simple, covers essentials	Functionality, Usability
Academic Library	ISO 25010	Complex needs, integrations	Security, Compatibility
Library Consortium	ISO 25010 + CMMI	Multiple stakeholders, process rigor	All characteristics, Process maturity
Developing Custom LMS	All models	Complete quality assurance	Everything, especially Maintainability

2. 复杂度度指标基础
2. Complexity Metrics Foundation

2.1 Code Complexity Theory
- ❑*What makes code complex?*

2.2 Cyclomatic Complexity (McCabe)
- ❑*2.2.1 Theoretical Foundations*
  - ❑ *Graph theory basis*
  - ❑ *Control flow graphs*
  - ❑ *Decision points and paths*
- ❑*2.2.2 Calculation Methods*
  - ❑ *Manual calculation examples*
  - ❑ *Automated tools overview*
- ❑*2.2.3 Interpretation Guidelines*
  - ❑ *Industry thresholds and standards*
  - ❑ *Context-aware interpretation*
  - ❑ *Limitations and criticisms*

❑*2.3 Halstead Complexity Metrics*

2.1 代码复杂性理论
- ❑ 是什么让代码变得复杂?

2.2 环状复杂性（McCabe）
- 2.2.1 理论基础
  - ❑ 图论基础
  - ❑ 控制流程图
  - ❑ 决策点与路径
- 2.2.2 计算方法
  - ❑ 手动计算示例
  - ❑ 自动化工具概述
- 2.2.3 解释指南
  - ❑ 行业门槛与标准
  - ❑ 上下文感知解释
  - ❑ 局限性与批评

2.3 霍尔斯特复杂度指标

2. 复杂度度指标基础
2. Complexity Metrics Foundation

2.1 代码复杂性理论

- *是什么让代码变得复杂?*
  - *规模复杂性——“过大”问题*
  - *控制流复杂性——“意大利面条代码”问题*

2.1 Code Complexity Theory

- *What makes code complex?*
  - *Size Complexity - The "Too Big" Problem*
  - *Control Flow Complexity - The "Spaghetti Code" Problem*

Low Complexity

```
def calculate_total(prices)
  return sum(prices)
```

High Complexity

```
def calculate_total(prices)
  total = 0
  for price as prices {
    if (price >= 0) {
      while price >= 0 {
        total += þ 1
      }
      return total
  }
```

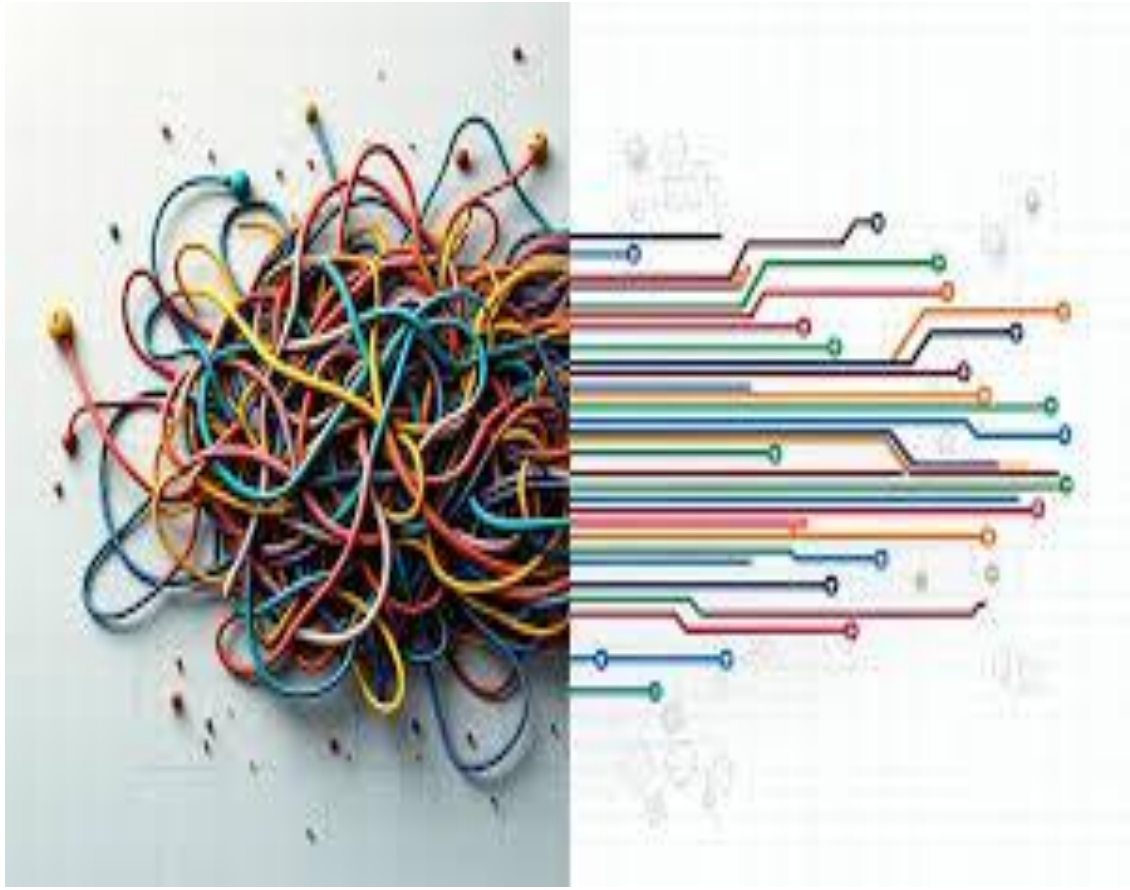
2. 复杂度度指标基础
2. Complexity Metrics Foundation

2.1 代码复杂性理论

- *是什么让代码变得复杂?*
  - *规模复杂性——“过大”问题*
  - *控制流复杂性——“意大利面条代码”问题*

2.1 Code Complexity Theory

- *What makes code complex?*
  - *Size Complexity - The "Too Big" Problem*
  - *Control Flow Complexity - The "Spaghetti Code" Problem*



2. 复杂度度指标基础
2. Complexity Metrics Foundation

2.2 Cyclomatic Complexity (McCabe)
- *2.2.1 Theoretical Foundations*
- *2.2.2 Calculation Methods*
- *2.2.3 Interpretation Guidelines*

2.2 环状复杂性（McCabe）
- 2.2.1 理论基础
- 2.2.2 计算方法
- 2.2.3 解释指南

2. 复杂度度指标基础
2. Complexity Metrics Foundation

Cyclomatic Complexity	Meaning
1 – 10	• Structured and Well Written Code • High Testability • Less Cost and Effort
10 – 20	• Complex Code • Medium Testability • Medium Cost and Effort
20 – 40	• Very Complex Code • Low Testability • High Cost and Effort
> 40	• Highly Complex Code • Not at all Testable • Very High Cost and Effort

Grade	Complexity Range	Meaning	Recommended Action	Typical Characteristics
A	1-5	Excellent - Simple, easy to understand	No action needed	• Single responsibility • Few/no conditionals • Easy to test • Clear logic
B	6-10	Good - Moderate complexity	Monitor for growth	• Several conditionals • Minor nesting • Still testable • Reasonable logic flow
C	11-20	Needs Attention	Consider refactoring	• Multiple nested ifs • Several logical paths • Testing requires effort • Harder to maintain
D	21-30	Poor	Should be refactored	• Deep nesting (3+ levels) • Many decision points • Hard to test fully • High bug risk
E	31-40	Very Poor	Urgent refactoring needed	• Excessive complexity • Many code paths • Nearly untestable • Very high bug risk
				• Spaghetti code • Unmaintainable

2. 复杂度度指标基础
2. Complexity Metrics Foundation

Additional Context: Block Type Indicators

Symbol	Meaning	Location	Example
M	Method/Function	Inside or outside classes	M 22:4 find_book
C	Class	Top level (0 indentation)	C 1:0 BookManager
F	Function (Standalone)	Module level, not in class	F 5:0 calculate_total

2. 复杂度度指标基础
2. Complexity Metrics Foundation

- **Cyclomatic Complexity:** Measures the number of linearly independent paths through a function or module. Higher numbers generally mean more complex logic and more testing needed. (McCabe's Complexity)

Nodes: 6 (Start, N1, N2/N3, N4, End)

Edges: 6

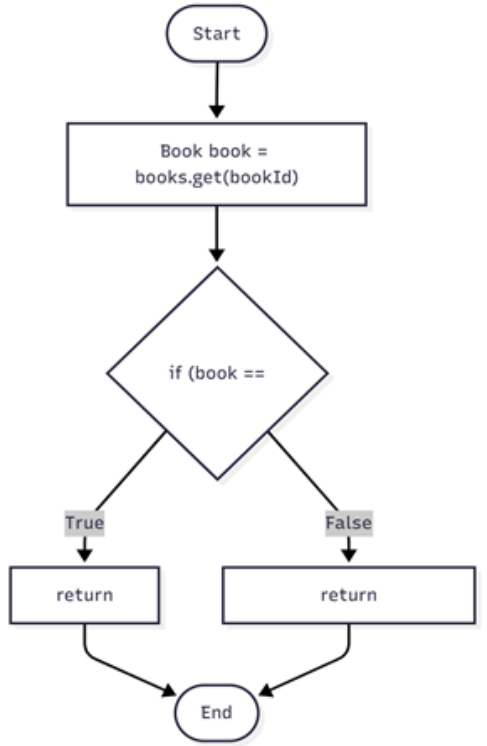
Cyclomatic Complexity: V(G) = E - N + 2 = 6 - 6 + 2 = 2

Independent Paths: 2

Path 1: Start → N1 → N2 → N3 → End (book is null)

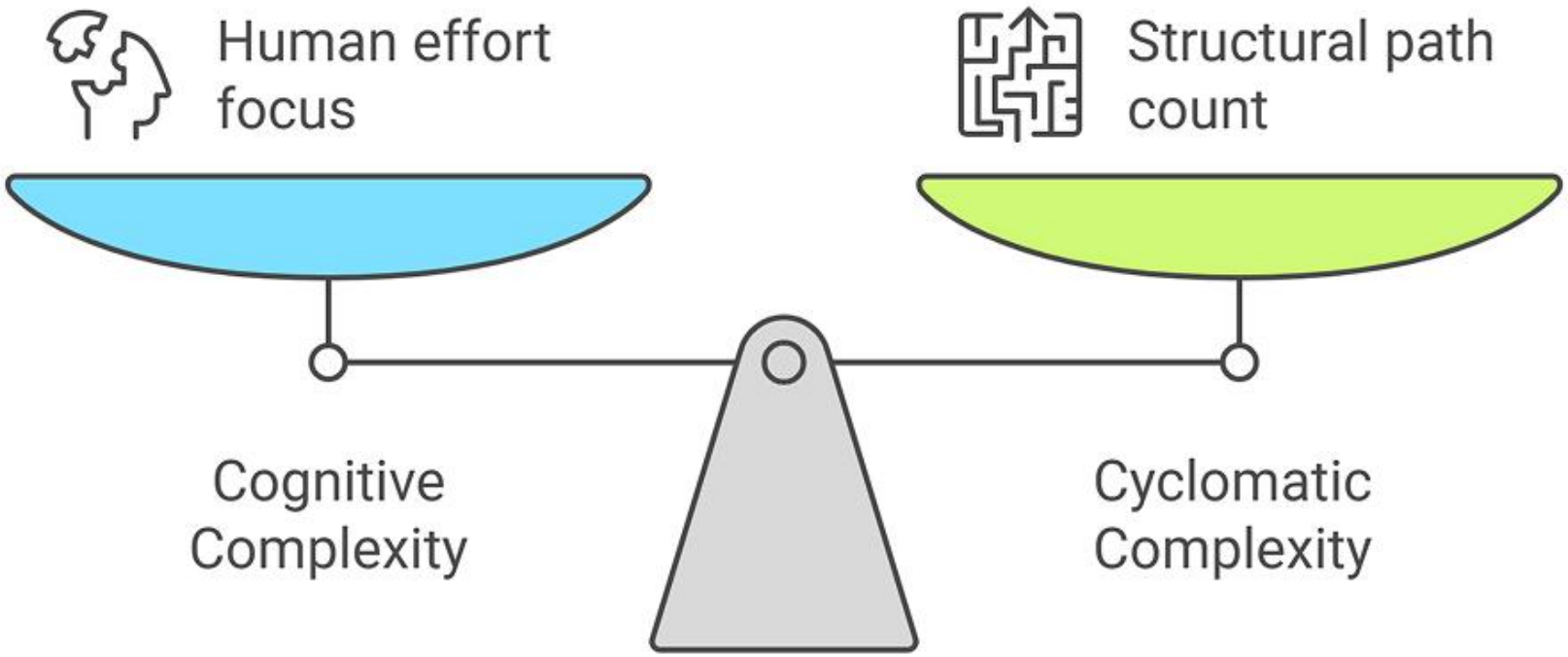
Path 2: Start → N1 → N2 → N4 → End (book exists, return availability)

Test Requirements: Minimum 2 tests to achieve branch coverage



2. 复杂度度指标基础
2. Complexity Metrics Foundation

- **Cognitive Complexity:** Attempts to measure how difficult a piece of code is for a human to understand. It considers nesting, control flow breaks, and other factors that impact readability. Often considered a better indicator of maintainability than cyclomatic complexity.



2. 复杂度度指标基础
2. Complexity Metrics Foundation

- **Halstead Complexity Measures:** A set of metrics based on the number of operators and operands in the code. Includes volume, difficulty, effort, and time required to understand the code.

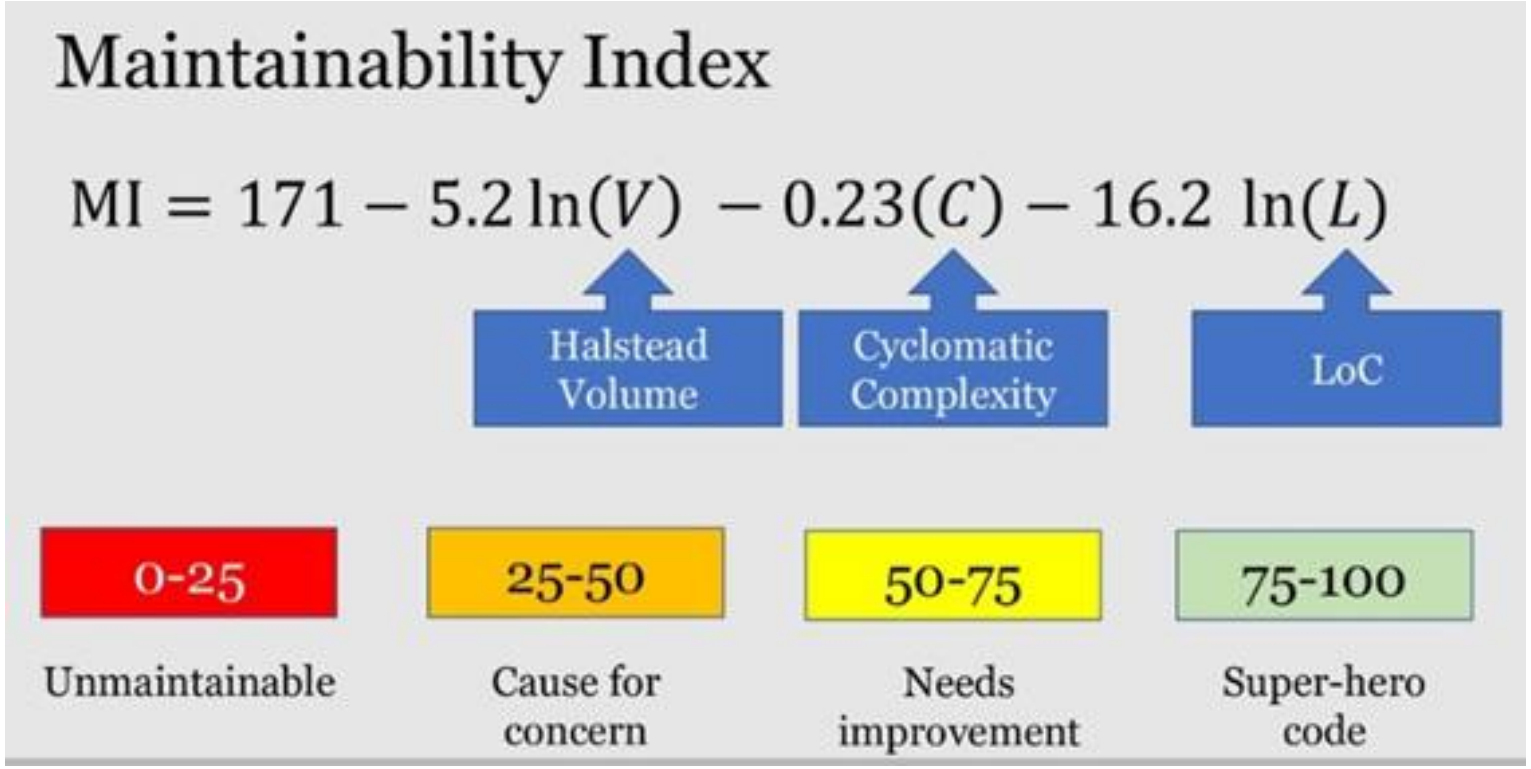
Halstead's Complexity Metrics

n1 = the number of distinct operators
n2 = the number of distinct operands
N1 = the total number of operators
N2 = the total number of operands

Program length	$N = N1 + N2$
Program vocabulary	$n = n1 + n2$
Volume	$V = N * (LOG_2\ n)$
Difficulty	$D = (n1 / 2) * (N2 / n2)$
Effort	$E = D * V$

2. 复杂度度指标基础
2. Complexity Metrics Foundation

- **Maintainability Index:** A composite metric that combines cyclomatic complexity, lines of code, and Halstead volume to give an overall score of how easy the code is to maintain.



2. 复杂度度指标基础
2. Complexity Metrics Foundation

- **Lines of Code (LOC):** A simple measure of the size of the code. While not a direct measure of complexity, larger codebases are often more complex.
- **Nesting Depth:** How deeply nested control structures (if statements, loops) are within a function. Deep nesting can make code harder to follow.
- **Fan-in/Fan-out:** Fan-in refers to the number of functions that call a given function. Fan-out refers to the number of functions a function calls. High fan-in/fan-out can indicate a function is a central point of complexity.

线条与样式检查器（通常带有复杂度检查）
Linters & Style Checkers (Often with Complexity Checks)

These tools primarily focus on code style and potential errors, but many now include basic complexity checks.

- **SonarLint:** (Free, IDE Integration) Excellent for real-time feedback in your IDE (VS Code, IntelliJ, Eclipse, etc.). It identifies code smells, bugs, and vulnerabilities, and can flag overly complex code. Connects to SonarQube/SonarCloud for more in-depth analysis.
- **ESLint (JavaScript):** (Free) Highly configurable linter for JavaScript. Plugins can add complexity checks (e.g., eslint-plugin-complexity).
- **Pylint (Python):** (Free) A widely used Python linter. It includes checks for cyclomatic complexity and can be customized.
- **Flake8 (Python):** (Free) Another popular Python linter that can be extended with plugins for complexity analysis.
- **RuboCop (Ruby):** (Free) Ruby linter with complexity checks.
- **Checkstyle (Java):** (Free) A configurable tool for enforcing coding standards in Java, including complexity metrics.
- **PMD (Java):** (Free) Source code analyzer for Java that finds common programming flaws like unused variables, empty catch blocks, and overly complex code.
- **ktlint (Kotlin):** (Free) Official linter for Kotlin.

静态分析工具（专用复杂度分析）
Static Analysis Tools (Dedicated Complexity Analysis)

These tools are specifically designed to analyze code complexity and provide detailed reports. They often go beyond simple metrics and offer insights into code structure and maintainability.

- **SonarQube/SonarCloud:** (Free/Commercial) A very popular platform for continuous inspection of code quality. It supports many languages and provides detailed reports on complexity, code smells, bugs, vulnerabilities, and more. **CodeClimate:** (Commercial) Similar to SonarQube, CodeClimate analyzes code quality and provides feedback on complexity, maintainability, and security. Integrates with GitHub, GitLab, and Bitbucket. (Focus: Cyclomatic, Maintainability, Code Smells)
- **Understand (SciTools):** (Commercial) A powerful static analysis tool that provides a wealth of information about code structure, dependencies, and complexity. Supports many languages. It's particularly strong at visualizing code and understanding complex relationships. (Focus: All metrics, including Halstead, Call Graphs, Dependency Analysis)
- **NDepend (C#/.NET):** (Commercial) Specifically designed for .NET code. Provides detailed analysis of code dependencies, complexity, and quality. Excellent for large .NET projects. (Focus: Cyclomatic, Dependency Analysis, Code Quality Rules)
- **SourceMeter (Java):** (Free/Commercial) Focuses on measuring the size and complexity of Java codebases. Provides metrics like lines of code, cyclomatic complexity, and maintainability index.
- **PMD CPD (Copy/Paste Detector):** (Free) While PMD is listed above, its CPD component is worth highlighting. Detecting duplicated code is a form of complexity reduction. Duplication makes code harder to maintain and understand.

语言专用工具与库
Language-Specific Tools & Libraries

These are often smaller, more focused tools or libraries that you can integrate into your build process or use for specific tasks.

- ❑ **radon (Python):** (Free) A Python library for computing cyclomatic complexity. Can be used from the command line or integrated into your testing framework.
- ❑ **McCabe (Python):** (Free) Another Python library for calculating cyclomatic complexity.
- ❑ **jscomplexity (JavaScript):** (Free) A command-line tool for analyzing the complexity of JavaScript code.
- ❑ **complexity-report (JavaScript):** (Free) Generates a complexity report for JavaScript projects.
- ❑ **Go Complexity (Go):** (Free) A tool to calculate cyclomatic complexity for Go code.

IDE 插件
IDE Plugins

Many IDEs have plugins that provide complexity analysis directly within the editor.

IntelliJ IDEA: Has built-in code inspection features that can detect complex code. Plugins like SonarLint further enhance this.
Visual Studio Code: SonarLint, ESLint (with complexity plugins), and other extensions provide complexity analysis.
Eclipse: Similar to IntelliJ, Eclipse has built-in inspections and supports plugins like SonarLint.

选择合适的工具
Choosing the Right Tool

The best tool for you depends on:

- **Your Programming Language:** Some tools are language-specific.
- **Project Size:** For large projects, a platform like SonarQube/SonarCloud is often the best choice.
- **Budget:** Many excellent free and open-source tools are available.
- **Integration:** Consider how well the tool integrates with your IDE and build process.
- **Specific Metrics:** Do you need to focus on cyclomatic complexity, cognitive complexity, or other metrics?
- **Team Collaboration:** If you're working in a team, a platform that supports collaboration and tracking code quality over time is essential.

建议：
Recommendations:

- ❑ **For most projects (especially if you're starting out):** Start with a linter (Pylint, ESLint, RuboCop, etc.) and a complexity plugin. This provides quick feedback during development.
- ❑ **For larger projects or teams:** SonarQube/SonarCloud is an excellent choice.
- ❑ **For .NET projects:** NDepend is a powerful option.
- ❑ **For in-depth analysis and visualization:** Understand is a great tool, but it's commercial.

To narrow down the best recommendations for yourself:

- ❑ What programming language(s) are you using?
- ❑ What is the size of your project (roughly, in lines of code or number of files)?
- ❑ Are you working on a team, or is this a personal project?
- ❑ What is your budget? (Free, willing to pay a small amount, or willing to pay for a commercial solution?)
- ❑ What are your primary goals for analyzing complexity? (e.g., identify overly complex functions, improve maintainability, reduce bugs)

为开源项目计算指标。
Calculating metrics for open source projects.

1. Select appropriate open-source projects for metrics analysis
   i. e.g. Library Management System
   ii. Hotel Reservation system
   iii. Or you can download/clone any open source project from github
2. Set up automated tools for complexity and quality measurement
   i. Anyone of your own choice or you can implement one by yourself
3. Calculate Cyclomatic Complexity and Halstead metrics
4. Assess software quality using ISO/IEC 25010 model
5. Present findings with actionable recommendations