Static code Analysis & Plugins

Static Code Analysis (SCA)

Definition
Static Code Analysis is the process of **examining source code without executing it** to find **bugs, vulnerabilities, or coding standard violations**.

- Helps ensure **code quality, maintainability, and security** early in development.

Why Static Code Analysis Is Needed

- Detect **bugs and errors early** (before runtime)
- Ensure **coding standards** and best practices are followed
- Identify **security vulnerabilities**
- Improve **code readability and maintainability**
- Reduce **cost of fixing defects** in later stages

How Static Code Analysis Works

1. **Parse source code**
2. **Analyze syntax and structure**
3. **Check against rules** (e.g., naming, complexity, null checks)
4. **Generate reports** for developers

Types of Static Code Analysis

Type	Purpose	Tools / Examples
Syntax & Semantic Checks	Detect syntax errors, type issues	Java Compiler, Eclipse/IntelliJ inspections
Code Style & Conventions	Enforce naming, formatting, documentation	Checkstyle, PMD
Code Complexity	Measure maintainability & cyclomatic complexity	SonarQube, PMD
Security Analysis	Detect vulnerabilities (SQLi, XSS, etc.)	Fortify, SonarQube, SpotBugs
Bug Detection	Detect null pointers, dead code, infinite loops	FindBugs, SpotBugs

---

Tools for Static Code Analysis

- **SonarQube** → Quality, security, maintainability
- **Checkstyle** → Coding standards
- **PMD** → Bug detection, code smells
- **SpotBugs (FindBugs)** → Detect runtime errors
- **IDE Built-in Analysis** → IntelliJ IDEA, Eclipse, VS Code

---

Real-Life Analogy

- Static code analysis = **spellcheck and grammar check** before sending a document
- Detects **errors and inconsistencies** without running the program

---

Best Practices

- Integrate SCA into **CI/CD pipeline**
- Regularly review and **fix reported issues**
- Combine with **unit tests and code reviews** for high-quality code
- Set **severity thresholds** to focus on critical issues

Common Interview Questions

Q1. What is static code analysis?

A. Examining source code without executing it to find bugs, vulnerabilities, or coding standard violations.

Q2. Difference between static and dynamic code analysis?

- Static → Examines code **without running**
- Dynamic → Analyzes code **during execution**

Q3. Name popular static code analysis tools for Java.

A. SonarQube, PMD, Checkstyle, SpotBugs

Q4. Why is static code analysis important in a project?

A. Early bug detection, enforce coding standards, improve maintainability, reduce costs.

Quick Summary Table

Feature	Static Analysis	Dynamic Analysis
Execution required	❌	✅
Detect runtime errors	Limited	Yes
Detect coding standard violations	✅	❌
Tools	SonarQube, PMD, Checkstyle	JUnit, Profilers

One-Line Summary

Static Code Analysis inspects source code without execution to detect bugs, enforce standards, and improve maintainability.

PMD Rules and Rulesets

Definition

PMD is a **static code analysis tool for Java** that detects **potential bugs, code smells, and bad practices.**

- **Rules** → Individual checks PMD performs (e.g., unused variables, empty catch blocks)
- **Rulesets** → Collections of multiple rules grouped for convenience

---

Why PMD Rules & Rulesets Are Needed

- Automate **code quality checks**
- Enforce **coding standards**
- Detect **bugs and maintainable issues** early
- Reduce **technical debt**

Key PMD Rules

Rule Category	Purpose	Examples
Basic	Detect common mistakes	Unused imports, unused local variables
Code Style	Enforce formatting and readability	Naming conventions, class and method naming
Design	Identify poor design choices	God class, long methods
Error Prone	Catch potential bugs	Empty catch block, null pointer risks

Rule Category	Purpose	Examples
Performance	Improve efficiency	Unnecessary object creation, inefficient loops
Security	Detect vulnerabilities	SQL injection risk, hardcoded credentials

---

Rulesets

- **Rulesets** = group of rules that can be **applied together**
- PMD comes with **predefined rulesets**, e.g.:
  - java-basic → basic coding rules
  - java-design → design-related rules
  - java-codesize → method/class length checks
  - java-unusedcode → unused variables, imports, methods
  - java-security → detect potential security issues
- Users can also **create custom rulesets** to match project standards

---

Example PMD Command (CLI)

```
pmd -d src/ -R rulesets/java/quickstart.xml -f text
```

- -d src/ → directory to scan
- -R rulesets/... → ruleset to apply
- -f text → output format

Real-Life Analogy

- PMD Rules = **individual inspection items** (check if windows are clean)
- Rulesets = **checklist for the entire house** (all inspections together)

Best Practices

- Use **standard PMD rulesets** to enforce **team-wide consistency**
- Customize rulesets to **match project coding standards**
- Integrate PMD in **CI/CD pipelines** for automated checks
- Regularly **review and update rules** as project evolves

Common Interview Questions

Q1. What is PMD?
A. A static code analysis tool for Java to detect bugs, bad practices, and code smells.

Q2. What is a PMD rule?
A. A single check that looks for a specific issue in the code.

Q3. What is a PMD ruleset?
A. A collection of PMD rules applied together.

Q4. Name some common PMD rulesets.

- java-basic, java-design, java-codesize, java-unusedcode, java-security

Q5. Can we create custom rulesets?
A. Yes, by grouping rules in an XML file tailored to project standards.

Quick Summary Table

Term	Description	Example
Rule	Single code check	Unused local variable
Ruleset	Collection of rules	java-basic.xml
Custom Ruleset	Tailored to project	myproject-rules.xml
CLI Command	Apply ruleset	pmd -d src/ -R rulesets/java/quickstart.xml -f text

---

One-Line Summary
PMD rules are individual code checks, and rulesets are groups of rules applied together to ensure code quality and maintainability.

PMD Configuration

Definition:
PMD Configuration refers to **setting up PMD** to analyze your project according to **specific rulesets and preferences**.

- Determines **which rules to apply, which files to scan, and how results are reported**.

Why PMD Configuration Is Needed

- Apply **custom coding standards**
- Avoid scanning **irrelevant files**
- Integrate PMD in **IDE or CI/CD pipelines**
- Generate **readable reports** for developers

Ways to Configure PMD

CLI Configuration

```
pmd -d src/ -R rulesets/java/quickstart.xml -f text
```

- -d → directory to scan
- -R → ruleset(s)
- -f → output format (text, xml, html)

Multiple rulesets example:

```
pmd -d src/ -R rulesets/java/basic.xml,rulesets/java/design.xml -f html
```

IDE Integration
Eclipse

- Install **PMD plugin**
- Configure **rulesets** in PMD preferences
- Right-click project → **PMD** → **Check code**

IntelliJ IDEA

- Install **PMD plugin**
- Configure **rulesets path** in settings
- Scan project or files → PMD results shown in IDE

Maven Configuration

- PMD can run via **Maven plugin** for CI/CD integration

pom.xml example:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.15.0</version>
      <configuration>
        <rulesets>
          <ruleset>/rulesets/java/basic.xml</ruleset>
          <ruleset>/rulesets/java/design.xml</ruleset>
        </rulesets>
```

```
        <failOnViolation>true</failOnViolation>
      </configuration>
      <executions>
        <execution>
          <phase>verify</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

- Generates **report and fails build** on violations

Gradle Configuration

```
apply plugin: 'pmd'

pmd {
    toolVersion = '6.59.0'
    ruleSets = ["java-basic", "java-design"]
}

tasks.withType(Pmd) {
    reports {
        xml.enabled = true
        html.enabled = true
    }
}
```

Real-Life Analogy

- PMD configuration = **setting rules for a spellcheck tool**
- Choose **which rules apply, which documents to scan, and how results appear**

Best Practices

- Use **consistent rulesets across IDE and CI/CD**
- Exclude **generated or third-party code**
- Enable **HTML/XML reports** for readability
- Update **rulesets periodically** to follow best practices

Common Interview Questions

Q1. How do you configure PMD for a Java project?

A. Using **CLI, IDE plugin, Maven plugin, or Gradle plugin** with rulesets.

Q2. How can PMD fail a build?

- Set <failOnViolation>true</failOnViolation> in Maven or equivalent in Gradle.

Q3. Can we use multiple rulesets together?

- Yes, separate them with commas or list multiple <ruleset> entries.

Q4. Why configure PMD instead of default rules?

- To **match project-specific coding standards** and avoid unnecessary checks.

Quick Summary Table

Configuration Type	How	Notes
CLI	`pmd -d src/ -R rulesets/... -f html`	Quick scan, custom output
Eclipse	PMD plugin	Right-click project → scan
IntelliJ IDEA	PMD plugin	Rulesets configured in settings
Maven	maven-pmd-plugin	Integrate with CI/CD, fail build on violations
Gradle	`pmd { ruleSets = [...] }`	Build integration, HTML/XML reports

One-Line Summary
PMD configuration defines rulesets, scanning scope, and reporting to enforce code quality consistently across projects.

Code Style Checking

Definition
Code Style Checking is the process of **ensuring that source code follows a set of coding standards** regarding naming, formatting, indentation, spacing, and structure.

- Promotes **readable, maintainable, and consistent code** across teams.

Why Code Style Checking Is Needed

- Improves **readability and maintainability**
- Ensures **team-wide consistency**
- Reduces **bugs caused by messy or unclear code**
- Simplifies **code reviews** and collaboration
- Can be **automated** using static code analysis tools

Common Code Style Rules

Category	Example
Naming Conventions	Classes in PascalCase, variables in camelCase
Braces and Indentation	4-space indentation, braces on same line for methods
Line Length	Maximum 100–120 characters per line
Spacing	Spaces after commas, around operators
Commenting & Documentation	JavaDoc for public methods, meaningful comments
Avoid Code Smells	No empty catch blocks, no magic numbers, consistent method length

---

Tools for Code Style Checking

Tool	Purpose	Notes
Checkstyle	Enforces Java coding standards	Integrates with IDE, Maven, Gradle
PMD	Detects code violations and style issues	Includes code conventions rules
SonarQube	Combines style, quality, and security	Reports and dashboards

Tool	Purpose	Notes
IDE Built-in Checkers	IntelliJ IDEA, Eclipse	Highlight formatting and naming issues in real-time

---

Example: Checkstyle Configuration

checkstyle.xml snippet:

```
<module name="Checker">
    <module name="TreeWalker">
        <module name="NamingConvention"/>
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>
        <module name="Indentation">
            <property name="basicOffset" value="4"/>
        </module>
    </module>
</module>
```

- Enforces **naming, indentation, and line length rules**

IDE Integration

- **IntelliJ IDEA:** Settings → Editor → Code Style → Apply rules automatically
- **Eclipse:** Preferences → Java → Code Style → Formatter → Save Actions

---

Real-Life Analogy

- Code style checking = **grammar and punctuation check in writing**
- Ensures **everyone writes clearly and consistently**

Best Practices

- Agree on **team-wide coding standards**
- Automate style checking via **CI/CD**
- Combine with **PMD / SonarQube** for full quality checks
- Fix style issues **early** during development

Common Interview Questions

Q1. What is code style checking?
A. Ensuring code follows formatting, naming, and readability standards.
Q2. Name tools for Java code style checking.
A. Checkstyle, PMD, SonarQube, IDE built-in checkers
Q3. Why is code style important?
A. Improves readability, maintainability, reduces bugs, and simplifies reviews.
Q4. Can code style checking be automated?
A. Yes, using IDE plugins, Maven/Gradle plugins, or CI/CD pipelines.

Quick Summary Table

Feature	Example / Tool	Purpose
Naming Conventions	Class: PascalCase, Var: camelCase	Standardize identifiers
Indentation & Braces	4 spaces, braces style	Readability
Line Length	Max 120 characters	Prevent clutter
Tool	Checkstyle / PMD / SonarQube	Enforce rules automatically
IDE Support	IntelliJ, Eclipse	Real-time formatting hints

One-Line Summary
Code style checking ensures that code follows consistent formatting and naming standards to improve readability and maintainability.

CheckStyle Configuration

Definition
CheckStyle is a **static code analysis tool** for Java that enforces **coding standards** by checking for formatting, naming conventions, and style violations.
Configuration defines **which rules to apply** and how violations are reported.

Why CheckStyle Configuration Is Needed

- Enforce **team-specific coding standards**
- Catch **style violations** automatically
- Integrate with **IDE and CI/CD pipelines**
- Maintain **consistent and readable code** across the project

---

Core Components of CheckStyle Configuration

1. **Checker** → Root module, runs all style checks
2. **TreeWalker** → Traverses Java AST to apply rules
3. **Modules** → Individual checks, e.g.:
    - LineLength → Max characters per line
    - Indentation → Enforces indentation rules
    - NamingConvention → Class/method/variable naming
    - Javadoc → JavaDoc checks
4. **Properties** → Parameters for modules (e.g., max line length, indent size)

Sample checkstyle.xml Configuration

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
    "-//Checkstyle//DTD Checkstyle Configuration 1.3//EN"
    "https://checkstyle.org/dtds/configuration_1_3.dtd">

<module name="Checker">
    <module name="TreeWalker">

        <!-- Naming conventions -->
        <module name="TypeName"/>
        <module name="MethodName"/>
        <module name="ParameterName"/>

        <!-- Indentation and spacing -->
        <module name="Indentation">
            <property name="basicOffset" value="4"/>
            <property name="tabWidth" value="4"/>
        </module>
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>

        <!-- Documentation -->
        <module name="JavadocMethod"/>
```

```
        <module name="JavadocType"/>

        <!-- Other common checks -->
        <module name="EmptyCatchBlock"/>
        <module name="MagicNumber"/>

    </module>
</module>
```

Integration with IDEs

- **Eclipse:**
  - Install CheckStyle plugin → Preferences → CheckStyle → Configure rules → Apply to project
- **IntelliJ IDEA:**
  - Settings → Plugins → CheckStyle-IDEA → Configure rules → Scan project

---

Integration with Build Tools

Maven

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>3.2.2</version>
    <configuration>
        <configLocation>checkstyle.xml</configLocation>
        <encoding>UTF-8</encoding>
        <failOnViolation>true</failOnViolation>
    </configuration>
    <executions>
        <execution>
            <phase>validate</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle

```
apply plugin: 'checkstyle'

checkstyle {
    toolVersion = '10.12.0'
    configFile = file('config/checkstyle/checkstyle.xml')
}

tasks.withType(Checkstyle) {
    reports {
        xml.enabled = true
        html.enabled = true
    }
}
```

Real-Life Analogy

- CheckStyle = **grammar and formatting checker for code**
- Configuration = **rules about spelling, punctuation, and style**

Best Practices

- Maintain **one standard checkstyle.xml** across the project
- Integrate into **CI/CD pipelines** to enforce rules automatically
- Update rules as coding standards evolve
- Focus on **critical rules first** (indentation, naming, line length)

Common Interview Questions

Q1. What is CheckStyle?

A. A tool to enforce Java coding standards via static analysis.

Q2. What is a CheckStyle configuration file?

A. An XML file defining which style rules and modules to apply.

Q3. Can CheckStyle be integrated with Maven or Gradle?

- Yes, via maven-checkstyle-plugin or Gradle checkstyle plugin.

Q4. How do you enforce coding standards across a team?

- Share **standard checkstyle.xml** and integrate it into **IDE and CI/CD pipelines.**

Quick Summary Table

Feature	Purpose	Example
Checker	Root module	<module name="Checker"/>
TreeWalker	Traverses AST	<module name="TreeWalker"/>
Module	Individual rule	<module name="LineLength"/>
Properties	Rule parameters	<property name="max" value="120"/>
IDE Integration	Real-time check	Eclipse / IntelliJ plugin
Maven/Gradle	CI/CD enforcement	maven-checkstyle-plugin / Gradle checkstyle

---

One-Line Summary

CheckStyle configuration defines the rules and properties to enforce consistent coding standards in Java projects.

CheckStyle Rules

Definition

CheckStyle Rules are **predefined or custom checks** applied to Java code to enforce **coding standards, formatting, and best practices.**

- Each rule defines **what is considered a violation** and can be configured in the **checkstyle.xml** file.

Why CheckStyle Rules Are Needed

- Ensure **consistent naming and formatting**
- Detect **common mistakes** like empty blocks or long methods
- Improve **code readability and maintainability**
- Enforce **team coding standards automatically**

Common Categories of CheckStyle Rules

Category	Purpose	Examples
Naming Rules	Ensure consistent identifiers	TypeName, MethodName, ParameterName, LocalVariableName
Whitespace & Indentation	Improve readability	Indentation, LeftCurly, RightCurly, WhitespaceAfter

Category	Purpose	Examples
Line Length	Prevent horizontal scrolling	LineLength (max 100–120 chars)
Coding Practices	Detect common mistakes	EmptyCatchBlock, MagicNumber, HiddenField
Javadoc & Documentation	Ensure proper comments	JavadocType, JavadocMethod, JavadocVariable
Imports & Package Rules	Organize imports	UnusedImports, ImportOrder, PackageDeclaration
Complexity Rules	Limit code complexity	CyclomaticComplexity, ClassDataAbstractionCoupling

---

Example Rules in checkstyle.xml

```
<module name="Checker">
    <module name="TreeWalker">

        <!-- Naming conventions -->
        <module name="TypeName"/>
        <module name="MethodName"/>
        <module name="ParameterName"/>

        <!-- Indentation and spacing -->
        <module name="Indentation">
            <property name="basicOffset" value="4"/>
        </module>
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>

        <!-- Documentation -->
        <module name="JavadocMethod"/>
        <module name="JavadocType"/>

        <!-- Common mistakes -->
        <module name="EmptyCatchBlock"/>
        <module name="MagicNumber"/>
    </module>
</module>
```

Real-Life Analogy

- CheckStyle Rules = **house rules**
- Naming rules → label rooms clearly
- Indentation → organize furniture neatly
- Documentation → instructions for guests

Best Practices

- Start with **standard CheckStyle rulesets** (Google Java Style, Sun Java Style)
- Customize **team-specific rules** as needed
- Regularly **update rules** based on evolving coding standards
- Combine with **PMD / SonarQube** for complete static code analysis

---

Common Interview Questions

Q1. What are CheckStyle rules?

A. Checks applied to Java code to enforce coding standards and detect violations.

Q2. Name some common CheckStyle rules.

- TypeName, MethodName, Indentation, LineLength, EmptyCatchBlock, JavadocMethod, MagicNumber

Q3. Why use CheckStyle rules in a project?

- Ensure **readable, maintainable, and consistent code**, and catch issues early.

Q4. Can we create custom CheckStyle rules?

- Yes, by defining **custom modules** in checkstyle.xml.

Quick Summary Table

Rule Category	Examples	Purpose
Naming	TypeName, MethodName	Consistent identifiers
Indentation & Whitespace	Indentation, LeftCurly	Readable formatting
Line Length	LineLength	Prevent long lines
Coding Practices	EmptyCatchBlock, MagicNumber	Detect mistakes
Documentation	JavadocMethod, JavadocType	Proper comments
Imports	UnusedImports, ImportOrder	Organized imports
Complexity	CyclomaticComplexity	Maintainable code

One-Line Summary

CheckStyle rules define coding standards and best practices that code must follow to ensure readability, consistency, and maintainability.

Code Quality Metrics

Definition

Code Quality Metrics are **quantitative measures** used to assess **how good, maintainable, and reliable code is**.

- Helps **evaluate code health, detect risks, and improve software quality.**

Why Code Quality Metrics Are Important

- Measure **maintainability, complexity, and readability**
- Detect **high-risk or error-prone areas**
- Improve **team productivity and collaboration**
- Support **continuous improvement and refactoring**
- Useful in **code reviews, CI/CD, and audits**

Common Code Quality Metrics

Metric	Purpose	How Measured	Tool Support
Cyclomatic Complexity	Measures **code complexity / decision paths**	Count of conditional branches	PMD, SonarQube
Lines of Code (LOC)	Size of code	Total lines	SonarQube, IDEs
Code Coverage	% of code tested by unit tests	# lines tested / total lines	JaCoCo, Cobertura
Maintainability Index	How easy code is to maintain	Calculated from LOC, complexity, comments	Visual Studio, SonarQube

Metric	Purpose	How Measured	Tool Support
Coupling Between Objects (CBO)	Measures class dependencies	Count references to other classes	SonarQube, PMD
Depth of Inheritance (DIT)	Measures inheritance tree depth	# of ancestor classes	PMD, SonarQube
Comment Density	% of code documented	# comment lines / total lines	IDE tools, SonarQube
Code Duplication	Measures repeated code	% duplicated code blocks	SonarQube, PMD
Fan-in / Fan-out	Method usage and dependencies	# calls to/from method	SonarQube

---

Example: Cyclomatic Complexity

```
int max(int a, int b, int c) {
    if(a > b && a > c) return a;
    else if(b > c) return b;
    else return c;
}
```

- **Decision points:** 3 (if, else if, else)
- **Cyclomatic Complexity = 3 + 1 = 4**

Real-Life Analogy

- Cyclomatic complexity → **number of routes on a map** (more routes → harder to navigate)
- Code coverage → **percentage of tasks checked** in a checklist
- Maintainability → **ease of repairing a machine**

Best Practices

- Keep **complexity low** for readability
- Write **unit tests** to improve coverage
- Reduce **class coupling** and **deep inheritance**
- Avoid **duplicate code**
- Use **automated tools** like SonarQube, PMD, or CheckStyle

Common Interview Questions

Q1. What are code quality metrics?

A. Quantitative measures to assess code maintainability, complexity, and reliability.

Q2. Name important code quality metrics in Java.

- Cyclomatic Complexity, LOC, Code Coverage, Maintainability Index, CBO, DIT, Comment Density, Code Duplication

Q3. Why is Cyclomatic Complexity important?

- Measures decision paths; high complexity → harder to maintain and test.

Q4. Tools for measuring code quality metrics?

- SonarQube, PMD, CheckStyle, JaCoCo, Cobertura

Quick Summary Table

Metric	Purpose	Tool
Cyclomatic Complexity	Complexity of decision paths	PMD, SonarQube
LOC	Code size	IDE, SonarQube
Code Coverage	% tested by unit tests	JaCoCo, Cobertura
Maintainability Index	Ease of maintenance	SonarQube, IDE
CBO	Class coupling	PMD, SonarQube

Metric	Purpose	Tool
DIT	Depth of inheritance	PMD, SonarQube
Comment Density	Documentation quality	SonarQube, IDE
Code Duplication	Repeated code blocks	PMD, SonarQube

One-Line Summary
Code quality metrics provide measurable insights into code complexity, maintainability, and reliability, guiding improvements and reducing risks.

Technical Debt

Definition
Technical Debt is the **cost of shortcuts, suboptimal solutions, or incomplete code** in software development that will **require extra effort to fix later**.

- Similar to **financial debt**: quick solutions give short-term gains but accumulate long-term "interest" in maintenance.

Why Technical Debt Happens

- Rushing to meet **deadlines**
- Lack of **coding standards or best practices**
- Incomplete **refactoring or design**
- Ignoring **automated testing or documentation**
- Legacy or poorly maintained code

Types of Technical Debt

Type	Description	Example
Deliberate Debt	Intentional shortcuts to meet deadlines	Hardcoding values, skipping unit tests
Accidental Debt	Poor code due to lack of knowledge or experience	Inefficient algorithms, tight coupling
Bit Rot / Legacy Debt	Accumulates over time in old code	Deprecated APIs, outdated libraries
Design Debt	Poor architecture choices	God classes, deep inheritance
Testing Debt	Missing or incomplete tests	Low test coverage

Impacts of Technical Debt

- Slower **feature development**
- Higher **maintenance cost**
- More **bugs and errors**
- Reduced **team productivity**
- Difficult **refactoring and scaling**

Measuring Technical Debt

- **Code Quality Tools:** SonarQube estimates debt in **hours** or **days**
- **Cyclomatic Complexity:** High complexity → higher debt
- **Code Duplication:** Repeated code increases maintenance effort
- **Coverage Gaps:** Low unit test coverage → hidden debt

Managing Technical Debt

1. **Identify Debt** → Use tools like SonarQube, PMD
2. **Prioritize** → Fix critical/high-risk areas first
3. **Refactor Regularly** → Clean up code incrementally
4. **Follow Best Practices** → Avoid shortcuts, write tests
5. **Track Debt** → Maintain a **debt backlog** like feature backlog

---

Real-Life Analogy

- Quick-fix plumbing → works today but **leaks later**
- Rushed code → works now but **costs more to maintain later**

Best Practices

- Treat debt **like a real cost** in planning
- Avoid **long-term hacks**
- Allocate time for **refactoring in sprints**
- Integrate **static analysis and testing** to prevent new debt

---

Common Interview Questions

Q1. What is technical debt?

A. The extra effort required later due to suboptimal or shortcut solutions in code.

Q2. Why does technical debt occur?

- Deadline pressure, poor design, lack of tests, legacy code, or accidental mistakes.

Q3. How can technical debt be managed?

- Identify, prioritize, refactor, follow best practices, track it in backlog.

Q4. Name tools to measure technical debt.

- SonarQube, PMD, CheckStyle, CodeClimate

Quick Summary Table

Aspect	Details
Definition	Cost of shortcuts or poor code requiring future fixes
Causes	Deadlines, poor design, lack of tests, legacy code
Types	Deliberate, Accidental, Design, Testing, Bit Rot
Impacts	Bugs, slower development, higher maintenance cost
Management	Identify, prioritize, refactor, best practices, track
Tools	SonarQube, PMD, CheckStyle

One-Line Summary

Technical debt is the accumulated cost of quick or suboptimal coding choices that must be addressed later to maintain software quality and productivity.

Code Smells

Definition

Code Smells are **warning signs in code** that indicate **poor design or bad practices**. They **don't break the program**, but they **make code hard to maintain, extend, or understand**.

- Think of them as **bad smells in food** – not spoiled yet, but a sign something's wrong.

Why Code Smells Occur

- Poor design decisions
- Copy-paste coding
- Lack of refactoring
- Time pressure / deadlines
- Inexperienced developers
- Ignoring coding standards

Why Code Smells Are Dangerous

- Increase **technical debt**
- Make debugging difficult
- Increase chance of **bugs**
- Reduce **code readability**
- Slow down future development

Common Types of Code Smells

A. Bloaters (Code too big)

1. Long Method

- Method doing **too many things**
- Hard to understand and test

  Fix: Extract Method

Example:

```
calculateSalaryAndPrintAndSave();
```

---

2. Large Class (God Class)

- Class with **too many responsibilities**

  Fix: Single Responsibility Principle

B. Object-Oriented Abusers

3. Feature Envy

- Method uses **another class's data more than its own**

  Fix: Move method to the relevant class

---

4. Refused Bequest

- Subclass doesn't use parent methods

  Fix: Re-design inheritance / use composition

C. Change Preventers

5. Shotgun Surgery

- One change requires edits in **many classes**

  Fix: Centralize logic

---

6. Divergent Change

- One class changes for **many different reasons**

  Fix: Split class responsibilities

D. Dispensables (Unnecessary code)

7. Duplicate Code

- Same logic repeated in multiple places

  Fix: Extract common method

---

8. Dead Code

- Unused variables, methods, classes

  Fix: Remove safely

E. Couplers

9. Tight Coupling

- Classes highly dependent on each other
  Fix: Use interfaces, dependency injection

---

10. Message Chains

- Long method calls like a.getB().getC().getD()
  Fix: Hide delegation

Code Smells vs Bugs

Aspect	Code Smell	Bug
Breaks program?	No	Yes
Compilation error	No	Often
Long-term impact	High	Immediate
Fix urgency	Medium	High

Detecting Code Smells

- **Static Analysis Tools**
  - SonarQube
  - PMD
  - CheckStyle
- **Code Reviews**
- High complexity / low readability
- Frequent bug-prone areas

Removing Code Smells (Refactoring Techniques)

Code Smell	Refactoring
Long Method	Extract Method
Duplicate Code	Extract Method/Class
Large Class	Split Class
Tight Coupling	Use Interfaces / DI
Feature Envy	Move Method

Real-Life Example

- A room full of **unnecessary furniture** → hard to move
- Code with smells → hard to change

Interview Questions & Answers

Q1. What are code smells?
A. Indicators of poor design that make code hard to maintain.

Q2. Are code smells bugs?
A. No, but they can lead to bugs later.

Q3. How do you remove code smells?
A. By refactoring and following design principles.

Q4. Name tools to detect code smells.
A. SonarQube, PMD, CheckStyle.

Quick Revision Table

Topic	Summary
Definition	Signs of poor code design
Causes	Bad design, copy-paste, no refactoring

Topic	Summary
Impact	Technical debt, bugs, low readability
Detection	Static analysis tools, reviews
Solution	Refactoring

One-Line Summary
Code smells are warning signs of poor code quality that don't break the program but harm maintainability.

SonarQube Dashboard

Definition
The **SonarQube Dashboard** is a **central visual interface** that shows the **overall health and quality of a project's code** using metrics like **bugs, vulnerabilities, code smells, coverage, and technical debt.**

- It answers one question quickly: **"Is my code production-ready?"**

Purpose of SonarQube Dashboard

- Monitor **code quality continuously**
- Identify **technical debt & risks**
- Enforce **quality gates**
- Improve **maintainability and security**
- Support **CI/CD pipelines**

Main Sections of the Dashboard
A. Project Overview
Shows **high-level quality status**:

- **Bugs**
- **Vulnerabilities**
- **Code Smells**
- **Coverage**
- **Duplications**
- **Quality Gate Status (Pass/Fail)**

First screen interviewers expect you to explain.

B. Quality Gate
A **set of rules** that code must satisfy before merge/deploy.
Example conditions:

- No **new critical bugs**
- Coverage ≥ **80%**
- Duplication ≤ **3%**

If failed → **build should stop**

C. Reliability (Bugs)

- Logical or runtime errors
- Classified by severity:
  - Blocker
  - Critical
  - Major
  - Minor

Bugs = **things that can break the app**

D. Security (Vulnerabilities)

- Security flaws like:
  - SQL Injection
  - Hardcoded credentials
  - Weak cryptography

Vulnerabilities = **things hackers can exploit**

E. Maintainability (Code Smells)

- Poor design or bad practices
- Represent **technical debt**

Example:

- Long methods
- Duplicate code
- Large classes

Expressed in **time to fix (e.g., 2d 4h)**

F. Coverage

- Percentage of code covered by **unit tests**
- Derived from:
  - Lines covered
  - Branch coverage

Higher coverage → lower risk

G. Duplications

- Percentage of duplicated code blocks
- High duplication = **higher maintenance cost**

Metrics Shown on Dashboard

Metric	Meaning
Bugs	Code errors
Vulnerabilities	Security risks
Code Smells	Maintainability issues
Coverage	Test coverage
Duplication	Repeated code
Technical Debt	Effort to fix issues

Issue Drill-Down Feature

- Click any metric → see:
  - File name
  - Line number
  - Rule violated
  - Suggested fix

Very useful in **code reviews & debugging**

New Code vs Overall Code

- **New Code** → recently added/modified code
- **Overall Code** → entire codebase

Best practice: **focus on clean new code first**

Real-Life Analogy

- **Medical health dashboard**:
  - Bugs → illness
  - Vulnerabilities → infections
  - Coverage → immunity
  - Debt → pending treatments

Interview Questions & Answers

Q1. What is SonarQube Dashboard?

A. A visual interface to monitor code quality, security, and maintainability.

Q2. What does Quality Gate mean?

A. A set of conditions that code must meet before merging or deployment.

Q3. Difference between bugs and code smells?

- Bugs → can break functionality
- Code smells → design issues affecting maintainability

Q4. What is technical debt in SonarQube?

A. Estimated time required to fix code smells.

Quick Revision Table

Section	Focus
Overview	Overall project health
Quality Gate	Pass / Fail criteria
Bugs	Reliability
Vulnerabilities	Security
Code Smells	Maintainability
Coverage	Test quality
Duplications	Code reuse

One-Line Summary

The SonarQube Dashboard provides a real-time view of code quality, security, and technical debt to ensure production-ready software.

Quality Gates (SonarQube)

Definition

A **Quality Gate** is a **set of conditions** that a project's code **must satisfy** to be considered **acceptable for release or merge.**

- Simply: **Pass = deploy | Fail = fix first**

Purpose of Quality Gates

- Prevent **bad-quality code** from reaching production
- Enforce **coding standards automatically**
- Control **technical debt growth**
- Act as a **decision point in CI/CD pipelines**

When Quality Gates Are Checked

- After **code analysis**
- During **CI/CD pipeline execution**
- Before **merge / deployment**

If the gate fails → pipeline should stop

Common Quality Gate Conditions

Metric	Typical Condition
Bugs	No new critical/blocker bugs
Vulnerabilities	No new security issues
Code Smells	Debt ratio below threshold
Coverage	≥ 80% on new code
Duplications	≤ 3% on new code

New Code vs Overall Code

- **New Code** → recently written or modified code
- **Overall Code** → entire project

Best practice:
Stricter rules on New Code, lenient on legacy code

Default Quality Gate (Sonar Way)

SonarQube provides a built-in gate:

- No new bugs
- No new vulnerabilities
- No new code smells
- Coverage ≥ 80% on new code

Used by most teams initially

Custom Quality Gates

Organizations can define their own gates:
Example:

- Coverage ≥ 85%
- Zero blocker issues
- Debt ratio < 5%

Useful for enterprise standards (Cognizant-style projects)

Quality Gate Status

- **Passed** → Code can move forward
- **Failed** → Fix issues before proceeding

Displayed clearly on:

- SonarQube Dashboard
- CI tools (Jenkins, GitHub Actions)

Real-Life Analogy

- **Exam passing marks**:
  - Score ≥ pass → next semester
  - Score < pass → re-exam

Quality Gate = **minimum quality passing mark**

Interview Questions & Answers

Q1. What is a Quality Gate?
A. A set of quality conditions that code must pass to be accepted.
Q2. Why focus on "new code"?
A. To prevent introducing new technical debt while improving legacy code gradually.
Q3. What happens if a Quality Gate fails?
A. Build or merge should be blocked until issues are fixed.
Q4. Difference between Quality Profile and Quality Gate?

- **Quality Profile** → Rules to analyze code
- **Quality Gate** → Conditions to accept/reject code

Quick Revision Table

Aspect	Summary
Definition	Acceptance criteria for code quality
Purpose	Block bad code
Key Focus	New Code
Status	Pass / Fail
Default Gate	Sonar Way
Usage	CI/CD pipelines

One-Line Summary
Quality Gates ensure only clean, secure, and maintainable code moves forward in the development pipeline.

Quality Gates (SonarQube)

Definition
A **Quality Gate** is a **set of conditions** that a project's code **must satisfy** to be considered **acceptable for release or merge.**

- Simply: **Pass = deploy | Fail = fix first**

Purpose of Quality Gates

- Prevent **bad-quality code** from reaching production
- Enforce **coding standards automatically**
- Control **technical debt growth**
- Act as a **decision point in CI/CD pipelines**

When Quality Gates Are Checked

- After **code analysis**
- During **CI/CD pipeline execution**
- Before **merge / deployment**

If the gate fails → pipeline should stop

Common Quality Gate Conditions

Metric	Typical Condition
Bugs	No new critical/blocker bugs
Vulnerabilities	No new security issues
Code Smells	Debt ratio below threshold
Coverage	≥ 80% on new code
Duplications	≤ 3% on new code

New Code vs Overall Code

- **New Code** → recently written or modified code
- **Overall Code** → entire project

Best practice:
Stricter rules on New Code, lenient on legacy code

Default Quality Gate (Sonar Way)
SonarQube provides a built-in gate:

- No new bugs
- No new vulnerabilities
- No new code smells
- Coverage ≥ 80% on new code

Used by most teams initially

Custom Quality Gates
Organizations can define their own gates:
Example:

- Coverage ≥ 85%
- Zero blocker issues
- Debt ratio < 5%

Useful for enterprise standards (Cognizant-style projects)

Quality Gate Status

- **Passed** → Code can move forward
- **Failed** → Fix issues before proceeding

Displayed clearly on:

- SonarQube Dashboard
- CI tools (Jenkins, GitHub Actions)

Real-Life Analogy

- **Exam passing marks**:
  - Score ≥ pass → next semester
  - Score < pass → re-exam

Quality Gate = **minimum quality passing mark**

Interview Questions & Answers
Q1. What is a Quality Gate?
A. A set of quality conditions that code must pass to be accepted.
Q2. Why focus on "new code"?
A. To prevent introducing new technical debt while improving legacy code gradually.
Q3. What happens if a Quality Gate fails?
A. Build or merge should be blocked until issues are fixed.
Q4. Difference between Quality Profile and Quality Gate?

- **Quality Profile** → Rules to analyze code
- **Quality Gate** → Conditions to accept/reject code

Quick Revision Table

Aspect	Summary
Definition	Acceptance criteria for code quality
Purpose	Block bad code
Key Focus	New Code
Status	Pass / Fail
Default Gate	Sonar Way
Usage	CI/CD pipelines

One-Line Summary
Quality Gates ensure only clean, secure, and maintainable code moves forward in the development pipeline.

Quality Gates (SonarQube)

Definition
A **Quality Gate** is a **set of rules/conditions** that decides whether the code **can move forward** (merge, deploy, release) or **must be fixed first.**

- Simple meaning: **Quality check PASS or FAIL**

Purpose of Quality Gates

- Stop **bad-quality code** early
- Control **technical debt**
- Ensure **secure, reliable, maintainable** code
- Act as a **checkpoint in CI/CD pipelines**

When Quality Gates Are Applied

- After **SonarQube analysis**
- During **CI/CD build**
- Before **merge or deployment**

If gate fails → **build should fail**

Common Quality Gate Conditions

Metric	Typical Rule
Bugs	No new blocker/critical bugs
Vulnerabilities	No new security issues
Code Smells	Technical debt under limit
Coverage	≥ 80% on new code
Duplications	≤ 3% on new code

New Code vs Overall Code

- **New Code** → Recently added/changed code
- **Overall Code** → Entire project

Best practice:
Strict rules for New Code, relaxed for legacy code

Default Quality Gate – "Sonar Way"
Provided by SonarQube:

- No new bugs
- No new vulnerabilities
- No new code smells
- ≥ 80% coverage on new code

Used by most teams initially

Custom Quality Gates
Organizations can define their own rules, for example:

- Coverage ≥ 85%
- Zero blocker issues
- Debt ratio < 5%

Common in **enterprise projects (Cognizant)**

Quality Gate Status

- **Passed** → Code allowed
- **Failed** → Fix issues first

Visible in:

- SonarQube Dashboard

- Jenkins / CI pipeline logs

Real-Life Example

- **Exam result system**:
  - Marks ≥ pass → promoted
  - Marks < pass → reattempt

Quality Gate = **minimum quality passing mark**

Interview Questions & Answers

Q1. What is a Quality Gate?

A. A set of conditions that code must satisfy to be accepted.

Q2. Why are Quality Gates important?

A. They prevent bad-quality code from reaching production.

Q3. What happens if a Quality Gate fails?

A. The build/merge should be blocked.

Q4. Quality Gate vs Quality Profile?

- **Quality Profile** → Rules to detect issues
- **Quality Gate** → Decision to pass or fail code

Quick Revision Table

Aspect	Point
Role	Accept / Reject code
Focus	New Code
Used In	CI/CD
Default	Sonar Way
Result	Pass / Fail

One-Line Summary

Quality Gates act as automated checkpoints that ensure only clean, secure, and maintainable code moves forward.

SonarQube Integration

Definition

SonarQube Integration is the process of **connecting SonarQube with your project, build tools, and CI/CD pipeline** so that **code quality is analyzed automatically.**

- In short: **Build + SonarQube = Automatic code quality check**

Why SonarQube Integration Is Important

- Detect **bugs, vulnerabilities, and code smells early**
- Enforce **Quality Gates automatically**
- Reduce **manual code reviews**
- Maintain **high code quality** in teams
- Mandatory in most **enterprise projects**

Where SonarQube Is Integrated

Area	Tools
Build Tools	Maven, Gradle
CI/CD	Jenkins, GitHub Actions, GitLab
IDE	IntelliJ, Eclipse (SonarLint)

Area	Tools
Version Control	GitHub, GitLab

SonarQube Integration Flow

1. Developer commits code
2. CI pipeline starts
3. SonarQube Scanner runs
4. Code is analyzed
5. Results sent to SonarQube Server
6. Quality Gate **Pass / Fail**

Integrating SonarQube with Maven (Java)

Steps

- Configure SonarQube server
- Run command:

```
mvn clean verify sonar:sonar
```

Uses pom.xml configuration

Integrating SonarQube with Gradle

```
gradle sonar
```

Uses build.gradle

Integrating SonarQube with Jenkins

- Install **SonarQube Plugin**
- Configure:
  - SonarQube URL
  - Authentication token
- Add SonarQube stage in pipeline
- Fail pipeline if **Quality Gate fails**

Common in Cognizant projects

IDE Integration (SonarLint)

- Detects issues **while coding**
- Supports:
  - IntelliJ IDEA
  - Eclipse
  - VS Code
- Can sync rules with SonarQube server

Authentication in Integration

- Uses **SonarQube Token**
- Secure and preferred over passwords
- Configured in CI tools

Real-Life Analogy

- **Factory quality inspection**:
  - Products checked automatically
  - Defective items blocked
  - Only quality items move forward

SonarQube = **Quality inspector for code**

Interview Questions & Answers

Q1. What is SonarQube Integration?

A. Connecting SonarQube with build and CI tools for automated code analysis.

Q2. How does SonarQube stop bad code?
A. Using Quality Gates to fail builds if rules are violated.
Q3. Which plugin is used in IDEs?
A. SonarLint.
Q4. Where do we usually integrate SonarQube?
A. CI/CD pipelines like Jenkins.

Quick Revision Table

Feature	Purpose
Maven / Gradle	Build-time analysis
Jenkins	CI enforcement
SonarLint	Developer-side checks
Quality Gate	Accept / Reject code

One-Line Summary
SonarQube integration enables continuous, automated code quality checks across development and CI pipelines.

Code Coverage Analysis

Definition
Code Coverage Analysis measures **how much of your source code is executed** when your **unit tests run.**
Simply: **How much code is tested?**

Purpose of Code Coverage

- Ensure important logic is **tested**
- Identify **untested code**
- Improve **code quality & reliability**
- Reduce **bugs in production**
- Support **Quality Gates in SonarQube**

Common Types of Code Coverage

Type	Meaning
Line Coverage	Lines of code executed
Branch Coverage	if/else paths tested
Method Coverage	Methods invoked
Statement Coverage	Statements executed
Condition Coverage	Boolean conditions tested

Branch coverage is more reliable than line coverage.

Example

```
if (age > 18) {
    allow();
} else {
    deny();
}
```

- Line coverage: 100%
- Branch coverage:

    - Only if tested → 50%
    - Both if and else tested → 100%

Code Coverage Tools

- **JaCoCo** (Java – most common)
- Cobertura
- IntelliJ built-in coverage
- SonarQube (reports coverage)

Code Coverage in SonarQube

- Shows:
    - Overall coverage
    - Coverage on **new code**
- Used in **Quality Gates**
- Helps reduce **technical debt**

Typical requirement: **≥ 80% on new code**

High Coverage ≠ Good Tests

- 100% coverage doesn't guarantee:
    - No bugs
    - Good logic testing

Quality tests matter more than numbers.

Best Practices

- Focus on **critical business logic**
- Aim for **high branch coverage**
- Avoid testing getters/setters
- Write **meaningful assertions**
- Maintain coverage on **new code**

Real-Life Analogy

- CCTV coverage:
    - More coverage → fewer blind spots
    - Blind spots → hidden risks

Code coverage = **test visibility**

Interview Questions & Answers

Q1. What is code coverage?
A. A metric showing how much code is executed by tests.
Q2. Which coverage type is best?
A. Branch coverage.
Q3. Which tool is used in Java?
A. JaCoCo.
Q4. Does 100% coverage mean bug-free code?
A. No, test quality matters more.

Quick Revision Table

Aspect	Summary
Measures	Tested code
Tools	JaCoCo, SonarQube
Used In	Quality Gates
Best Type	Branch Coverage
Target	80%+ on new code

One-Line Summary
Code coverage analysis shows how effectively your tests execute the code and helps ensure reliability and maintainability.

JaCoCo Analysis

Definition
JaCoCo (Java Code Coverage) is a **Java library** used to **measure code coverage** by tracking which parts of the code are executed during **unit tests.**
Simply: **JaCoCo tells how much Java code is tested**

Why JaCoCo Is Used

- Measure **test effectiveness**
- Identify **untested code**
- Generate **coverage reports**
- Integrate with **Maven, Gradle**
- Feed coverage data to **SonarQube**

How JaCoCo Works (Flow)

1. Tests start execution
2. JaCoCo agent instruments bytecode
3. Code execution data is collected
4. Coverage report is generated
5. SonarQube reads the report

Types of Coverage Reported by JaCoCo

Coverage Type	Meaning
Instruction	Bytecode instructions executed
Line	Lines executed
Branch	if/else paths executed
Method	Methods called
Class	Classes touched

Branch coverage is most important in interviews.

JaCoCo with Maven (Configuration)
pom.xml

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.11</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
```

```
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Runs automatically during mvn test

Generating JaCoCo Reports

After running tests:

```
mvn test
```

Reports generated at:

```
target/site/jacoco/index.html
```

JaCoCo + SonarQube Integration

- JaCoCo generates coverage data
- SonarQube **reads the JaCoCo report**
- Coverage shown in **SonarQube Dashboard**
- Used in **Quality Gates**

JaCoCo = data provider, SonarQube = visualizer

JaCoCo Thresholds

You can fail builds if coverage is low:

Example:

- Line coverage < 80% → build fails

Enforced via Maven or SonarQube Quality Gates

Common Problems in JaCoCo

- Tests not running → 0% coverage
- Wrong report path in SonarQube
- Excluding test classes incorrectly

Real-Life Analogy

- Fitness tracker:
  - Tracks steps taken
  - Shows report
  - Helps improve activity

JaCoCo = **fitness tracker for test execution**

Interview Questions & Answers

Q1. What is JaCoCo?

A. A Java code coverage tool.

Q2. How does JaCoCo work?

A. Instruments bytecode during test execution and generates coverage reports.

Q3. How does JaCoCo integrate with SonarQube?

A. SonarQube reads JaCoCo coverage reports.

Q4. Where is JaCoCo report generated?

A. target/site/jacoco/index.html

Quick Revision Table

Feature	Details
Tool Type	Code coverage
Language	Java
Build Tools	Maven, Gradle
Reports	HTML, XML

Feature	Details
Used By	SonarQube

One-Line Summary
JaCoCo analyzes Java code coverage by tracking executed bytecode during unit tests and feeds results to SonarQube.

Integration with Maven

Definition
Maven Integration means configuring **tools (like JaCoCo, SonarQube, JUnit)** inside a **Maven project** so they run **automatically during the build lifecycle.**
Simply: **Build once → analysis runs automatically**

Why Maven Integration Is Important

- Automates **testing & analysis**
- Ensures **consistent builds**
- Enforces **code quality rules**
- Required for **CI/CD pipelines**
- Widely used in **enterprise projects**

Maven Build Lifecycle (Interview Must-Know)

Phase	Purpose
compile	Compile source code
test	Run unit tests
package	Create JAR/WAR
verify	Verify quality
install	Install to local repo
deploy	Deploy to remote repo

Most tools hook into **test / verify** phase

JaCoCo Integration with Maven
Purpose

- Measure **code coverage**
- Generate reports
- Share data with SonarQube

pom.xml Configuration

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.11</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
```

```
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Automatically runs during mvn test

Running Maven with JaCoCo

mvn clean test

Report location:

target/site/jacoco/index.html

SonarQube Integration with Maven

Command

mvn clean verify sonar:sonar

Maven triggers SonarQube Scanner

What Happens Internally

1. Maven builds project
2. JUnit tests run
3. JaCoCo collects coverage
4. SonarQube reads coverage
5. Quality Gate evaluated

Maven + SonarQube + JaCoCo Relationship

Tool	Role
Maven	Build automation
JUnit	Run tests
JaCoCo	Coverage data
SonarQube	Code quality analysis

Maven is the **orchestrator**

Common Maven Integration Issues

- Tests skipped → 0% coverage
- Wrong plugin phase
- Missing Sonar token
- Incorrect report paths

Real-Life Analogy

- Assembly line:
  - Maven → conveyor belt
  - JaCoCo → quality scanner
  - SonarQube → inspector

Everything runs **automatically**

Interview Questions & Answers

Q1. Why integrate tools with Maven?

A. To automate testing and quality checks during builds.

Q2. Which phase does JaCoCo hook into?

A. Test phase.

Q3. What command runs SonarQube analysis?

A. mvn verify sonar:sonar

Q4. Role of Maven in CI/CD?
A. Build automation and tool orchestration.

Quick Revision Table

Aspect	Summary
Tool	Maven
Role	Build automation
Integrates	JaCoCo, SonarQube
Key Phase	test / verify
Benefit	Automation

One-Line Summary
Maven integration automates testing, coverage, and code quality analysis during the build lifecycle.

Coverage Metrics (Line, Branch, Instruction)

Definition
Coverage Metrics are measurements that show **how much of your code is executed** when tests run.
In short: **They tell what part of the code is tested**

Why Coverage Metrics Matter

- Identify **untested logic**
- Improve **test quality**
- Reduce **hidden bugs**
- Enforce **Quality Gates**
- Support **SonarQube & JaCoCo analysis**

Line Coverage
Definition
Measures the **percentage of source code lines executed** during tests.
Formula
(Line executed / Total lines) × 100
Example

```
int sum(int a, int b) {
    return a + b;
}
```

- Tested → 100% line coverage

Pros

- Simple to understand

Limit

- Doesn’t ensure all conditions are tested

Branch Coverage
Definition
Measures whether **each possible path (if/else, switch)** is executed.
Example

```
if (marks >= 40) {
    pass();
```

```
} else {
    fail();
}
```

Tested Path	Branch Coverage
Only if	50%
Both if & else	100%

Why Important

- Detects **logical gaps**
- Preferred in interviews

Instruction Coverage (JaCoCo Specific)

Definition

Measures **bytecode instructions executed**, not source lines.

Key Point

- Most **accurate & low-level**
- Used internally by JaCoCo

A single line may contain **multiple instructions**

Comparison Table

Metric	What it Measures	Level
Line	Source code lines	High-level
Branch	Decision paths	Logical
Instruction	Bytecode instructions	Low-level

Which Metric Is Best?

- **Instruction Coverage** → Most precise
- **Branch Coverage** → Best for logic validation
- **Line Coverage** → Basic confidence

Branch coverage is most interview-relevant

Real-Life Analogy

- Line coverage → Did you visit rooms?
- Branch coverage → Did you take all doors?
- Instruction coverage → Did you step on every tile?

Tools Supporting These Metrics

- **JaCoCo** (Instruction, Line, Branch)
- SonarQube (Displays results)
- IntelliJ Coverage Tool

Interview Questions & Answers

Q1. Difference between line and branch coverage?

A. Line checks executed lines; branch checks decision paths.

Q2. What is instruction coverage?

A. Coverage of bytecode instructions executed.

Q3. Which coverage is more reliable?

A. Branch coverage.

Q4. Which tool provides instruction coverage?

A. JaCoCo.

Quick Revision Table

Metric	Key Use
Line	Basic testing

Metric	Key Use
Branch	Logic validation
Instruction	Precise measurement

One-Line Summary
Coverage metrics measure how effectively tests execute code, with branch and instruction coverage giving deeper confidence than line coverage.

Bug Detection

Definition
Bug Detection is the process of **identifying errors, defects, or faults** in software that can cause **incorrect or unexpected behavior**.
Simply: **Finding problems before users do**

Why Bug Detection Is Important

- Prevents **application failures**
- Improves **software reliability**
- Reduces **cost of fixing defects**
- Increases **customer satisfaction**
- Essential for **enterprise-quality software**

Types of Bugs

Type	Description	Example
Logical Bug	Wrong logic	Incorrect condition in if
Runtime Bug	Error during execution	NullPointerException
Syntax Bug	Code syntax error	Missing semicolon
Performance Bug	Slow execution	Infinite loop
Security Bug	Vulnerability	SQL Injection

Bug Detection Techniques
A. Static Bug Detection

- Analyzes code **without executing it**
- Finds:
  - Code smells
  - Potential bugs
  - Security vulnerabilities

Tools: SonarQube, PMD, CheckStyle

B. Dynamic Bug Detection

- Detects bugs **during execution**
- Uses:
  - Unit testing
  - Integration testing
  - Runtime analysis

Tools: JUnit, JaCoCo, Debuggers

Bug Detection in SonarQube

- Detects:
  - Null pointer risks

    - Resource leaks
    - Dead code
- Classifies bugs by:
    - Blocker
    - Critical
    - Major
    - Minor

Shows exact file and line number

Bug Detection Using Testing

- **Unit Tests** → catch logic errors
- **Integration Tests** → catch system issues
- **Regression Tests** → prevent old bugs

More tests = fewer surprises

Real-Life Analogy

- Medical tests:
    - Blood test → early disease detection
    - Code analysis → early bug detection

Earlier detection = **cheaper fix**

Bug vs Code Smell

Aspect	Bug	Code Smell
Breaks program	Yes	No
Immediate impact	High	Low
Needs urgent fix	Yes	Not always

Best Practices

- Use **static analysis tools**
- Write **unit tests**
- Enable **CI/CD checks**
- Fix bugs on **new code first**
- Perform **code reviews**

Interview Questions & Answers

Q1. What is bug detection?

A. Identifying defects in software before release.

Q2. Static vs Dynamic bug detection?

- Static → without running code
- Dynamic → during execution

Q3. Which tools detect bugs in Java?

A. SonarQube, PMD, JUnit, Debugger.

Q4. Why early bug detection is important?

A. Reduces cost and improves quality.

Quick Revision Table

Aspect	Summary
Purpose	Find defects
Methods	Static & Dynamic
Tools	SonarQube, JUnit
Benefit	Quality & reliability

One-Line Summary
Bug detection identifies defects early using analysis tools and testing to ensure reliable software.

SpotBugs Annotations

Definition
SpotBugs Annotations are **Java annotations** used to **inform SpotBugs static analysis tool** about code intentions, nullability, and expected behavior, so it can **detect bugs more accurately** and **reduce false positives.**
Simply: **Hints to SpotBugs about your code**

Why SpotBugs Annotations Are Needed

- Improve **bug detection accuracy**
- Avoid **false warnings**
- Document **developer intent**
- Help SpotBugs understand **null safety**
- Improve **code readability**

Commonly Used SpotBugs Annotations
A. Nullability Annotations
@Nonnull

- Value **must not be null**

```
@Nonnull
String getName() { return name; }
```

@Nullable

- Value **can be null**

```
@Nullable
String getMiddleName() { return middleName; }
```

Helps detect **NullPointerExceptions**

B. Ownership & Exposure Annotations
@CheckForNull

- Similar to @Nullable, but explicitly says **caller must check**

```
@CheckForNull
User findUser(int id);
```

C. Suppression Annotations
@SuppressFBWarnings

- Suppresses specific SpotBugs warnings

```
@SuppressFBWarnings(
  value = "NP_NULL_ON_SOME_PATH",
  justification = "Handled safely"
)
```

Used when warning is **false positive**

D. Value & Correctness Annotations
@OverrideMustInvoke

- Subclass **must call superclass method**

```
@OverrideMustInvoke
void close() { cleanup(); }
```

@Confidence

- Indicates confidence level of bug report
  (mostly internal usage)

Where SpotBugs Annotations Are Used

- Method return types
- Method parameters
- Fields
- Classes

Mostly used for **null-safety and correctness**

SpotBugs vs Java Standard Annotations

SpotBugs	Java
@Nonnull	@NotNull (JSR-305 / Jakarta)
@SuppressFBWarnings	@SuppressWarnings
Tool-specific	Compiler-level

Dependency Required

```
<dependency>
  <groupId>com.github.spotbugs</groupId>
  <artifactId>spotbugs-annotations</artifactId>
  <version>4.8.3</version>
</dependency>
```

Only annotations, **not the tool itself**

Best Practices

- Use @Nonnull by default
- Mark exceptions with @Nullable
- Use @SuppressFBWarnings **sparingly**
- Combine with **code reviews**
- Document justification when suppressing warnings

Real-Life Analogy

- Road signs:
  - “No Entry” → @Nonnull
  - “Check Before Entering” → @CheckForNull
  - “Ignore This Sign” → @SuppressFBWarnings

Interview Questions & Answers

Q1. What are SpotBugs annotations?
A. Annotations that help SpotBugs analyze code more accurately.
Q2. Purpose of @Nonnull?
A. Indicates value must never be null.
Q3. When to use @SuppressFBWarnings?
A. When SpotBugs warning is a false positive.
Q4. Do these annotations affect runtime?
A. No, they are for static analysis only.

Quick Revision Table

Annotation	Purpose
@Nonnull	Not null
@Nullable	Can be null
@CheckForNull	Must be checked

Annotation	Purpose
@SuppressFBWarnings	Suppress warnings

One-Line Summary

SpotBugs annotations guide static analysis by clearly expressing nullability and developer intent, improving bug detection accuracy.

Common Bug Patterns (Java-focused):

Null Pointer Dereference, Resource Leaks (not closing streams), Incorrect Equals/HashCode, Off-by-One Errors, and Concurrency Issues (race conditions, deadlocks).

Peer Review:

A process where developers examine each other's code to find defects, improve quality, share knowledge, and ensure standards before merging or release.

Code Review Checklist (Quick & Practical):

- **Correctness:** Logic is correct, edge cases handled, no obvious bugs
- **Readability:** Clear naming, proper formatting, simple and understandable code
- **Standards:** Follows coding conventions, style rules, and best practices
- **Design:** Proper separation of concerns, no tight coupling, reusable code
- **Performance:** No unnecessary loops, queries, or heavy operations
- **Security:** Input validation, no hardcoded secrets, safe APIs used
- **Testing:** Unit tests present, good coverage, meaningful assertions
- **Maintainability:** No code smells, minimal duplication, easy to modify