FindBugs (SpotBugs) Eclipse Plugin – Presentation Pack

A ready-to-present, demo-driven introduction to manual & automated Java code verification with the FindBugs™/SpotBugs Eclipse plugin and Jenkins CI.

---

0) Agenda

1. Why code quality matters & where static analysis fits
2. FindBugs/SpotBugs overview (what, why, how it works)
3. Install the Eclipse plugin (2 paths: legacy FindBugs vs modern SpotBugs)
4. Key features & bug patterns
5. **Demo 1** – Manual verification in Eclipse
6. **Demo 2** – Automated verification in Jenkins (Maven/Gradle)
7. Wrap-up, Q&A, references

---

1) Theory (context you can present)

1.1 Importance, motivation, scope

- **Goal**: reduce defects early (cheaper than after release), raise maintainability & security.
- **Scope**: Java applications and libraries; focus on bytecode-level static analysis (FindBugs/SpotBugs), but position among other QA activities.

1.2 Where code quality assurance (CQA) fits

- **Preventive**: coding standards, linters, type system, unit tests, static analysis.
- **Detective**: code review (peer), integration tests, e2e tests, fuzzing, telemetry.
- **Corrective**: refactoring, backlog of tech debt, security patching.

Techniques you can name-drop: unit/integration tests; inspections; static analyzers (FindBugs/SpotBugs, PMD, Checkstyle); code review; CI gates; security SAST/DAST; mutation testing; coverage; architecture linting.

1.3 Bug “templates” & review criteria

- **Bug patterns** (a.k.a. templates): recurring code idioms correlated with defects (e.g., null deref, wrong equals/hashCode, bad API usage, concurrency hazards, SQL injection via FindSecBugs, etc.).
- **Code review basis**: coding guidelines (e.g., Google Java Style), architectural constraints, and quality models (e.g., maintainability, reliability, security).

1.4 Automatic vs manual code verification

Aspect	Automatic (static analysis)	Manual (peer review)
Strength	Fast, repeatable, wide coverage; finds mechanical issues & some complex bugs	Context-aware, architectural & product understanding; catches logic, UX, requirements
Weakness	False positives; limited context; rule tuning needed	Human time-consuming; subjective; variable consistency
Best Use	CI gates, pre-commit checks, nightly jobs	Design decisions, readability, risk, test adequacy
Synergy	Use static analysis to flag candidates and **reduce reviewer noise**	Reviewers focus on high-value aspects; enforce fixing of tool-reported issues

---

2) Tool Overview

FindBugs™: classic Java static analyzer (now legacy) that scans **bytecode** for ~hundreds of bug patterns and ranks findings by *priority* (impact) and *confidence* (certainty).

SpotBugs: actively maintained successor to FindBugs with modern Java support. Eclipse plugin is named **"SpotBugs Eclipse Plugin"**. You can still demo using the FindBugs name/topic while running SpotBugs in current Eclipse versions.

Core ideas: - Bytecode analysis = catches issues independent of formatting; complements style linters. - **Bug rank** (Scariest…Of Concern) & **confidence** (High/Medium/Low) help triage. - Extensible via plugins: **fb-contrib** (extra correctness/perf rules), **FindSecBugs** (security rules).

---

3) Installation (Eclipse)

> Choose one path depending on your Eclipse version. Modern Eclipse ⇒ SpotBugs.

A) Modern pathway – SpotBugs Eclipse Plugin 1. Eclipse → **Help → Install New Software…** 2. Click **Add…**; Name: `SpotBugs` ; Location (update site): `https://spotbugs.github.io/eclipse/` 3. Select *SpotBugs Plugin* → Next → Accept → Restart. 4. Enable per project: Right-click project → **Properties → SpotBugs** (tick *Enable SpotBugs*).

B) Legacy pathway – FindBugs Eclipse Plugin (only if you must) 1. Help → Install New Software… 2. Add site: `https://findbugs.cs.umd.edu/eclipse/` 3. Install → Restart. (Only works reliably on old Eclipse 3.x)

Optional: add plugins - **FindSecBugs** (security) and **fb-contrib** (extra detectors). With Maven/Gradle these are easiest to apply; in pure IDE, drop the plugin JARs into the corresponding directories or rely on the Maven/Gradle build to run them.

---

4) Key Features (what to show)

- **Bug categories**: Correctness, Performance, Multithreading, Internationalization, Security (via FindSecBugs), Bad Practices, etc.
- **Ranks & Confidence**: prioritize the “Scariest/High” first; defer “Of Concern/Low”.
- **Rich reports**: tree view by package/class/category; source highlighting.
- **Suppress/Filter**: `@SuppressFBWarnings("PATTERN")` for justified cases; XML include/exclude filters in builds.
- **Custom configuration**: effort (Min/Default/Max), threshold (High/Medium/Low), annotation detectors on/off.
- **Integrations**: Maven, Gradle, Ant, Jenkins (Warnings NG), SonarQube import.

---

5) DEMO 1 – Manual verification in Eclipse (SpotBugs plugin)

5.1 Project seed (copy-paste code)

Create a Maven project `demo-findbugs` (or a plain Java project) and add these classes to trigger common findings.

`src/main/java/demo/BadEquals.java`

```java
package demo;
import java.util.Objects;
public class BadEquals {
    private final String id;
    public BadEquals(String id) { this.id = id; }
    // Bug: equals without hashCode; and String reference compare
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof BadEquals)) return false;
        BadEquals other = (BadEquals) o;
        return id == other.id; // should use Objects.equals(id, other.id)
    }
}
```

`src/main/java/demo/NullDeref.java`

```java
package demo;
public class NullDeref {
    public static int lenOrZero(String s) { return s.length(); } // NPE risk
}
```

`src/main/java/demo/Recursion.java`

```
package demo;
public class Recursion {
    public String toString() { return toString(); } // infinite recursion
}
```

`src/main/java/demo/Deadlock.java`

```
package demo;
public class Deadlock {
    private final Object a = new Object();
    private final Object b = new Object();
    public void m1() { synchronized (a) { synchronized (b) { /* ... */ } } }
    public void m2() { synchronized (b) { synchronized (a) { /* ... */ } } }
}
```

(Optional security) `src/main/java/demo/SqlInjection.java`

```
package demo;
import java.sql.*;
public class SqlInjection {
    public static ResultSet findUser(Connection c, String name) throws
Exception {
        Statement st = c.createStatement();
        return st.executeQuery("SELECT * FROM users WHERE name='" + name +
"'");
    }
}
```

5.2 Run analysis

1. Right-click project → **SpotBugs → Find Bugs** (or **Run SpotBugs**).
2. Open **SpotBugs Perspective** (Window → Perspective → Open → SpotBugs) to show the tree of findings.
3. Double-click a finding to jump to source; read the rule help; note *pattern id*, *rank*, *confidence*.

5.3 Evaluate & fix

- **BadEquals**: implement `hashCode()` and fix `equals()` to use value equality.
- **NullDeref**: handle null (`s == null ? 0 : s.length()`).
- **Recursion**: implement `toString()` safely.
- **Deadlock**: lock ordering or use higher-level concurrency primitives.
- **SqlInjection**: use `PreparedStatement` with parameters.

Re-run SpotBugs and take a screenshot for your slide.

5.4 Document decisions

- Mark truly intentional warnings with `@SuppressFBWarnings("PATTERN", justification = "…")` or via an exclude filter.

---

6) DEMO 2 – Automated verification in Jenkins (with Maven or Gradle)

Objective: run SpotBugs in CI and publish the results with Jenkins **Warnings NG** for trend charts and gating.

6.1 Maven configuration (`pom.xml`)

```
<build>
  <plugins>
    <plugin>
      <groupId>com.github.spotbugs</groupId>
      <artifactId>spotbugs-maven-plugin</artifactId>
      <version>4.9.6.0</version>
      <configuration>
        <effort>Max</effort>
        <threshold>Low</threshold>
        <plugins>
          <plugin>
            <groupId>com.h3xstream.findsecbugs</groupId>
            <artifactId>findsecbugs-plugin</artifactId>
            <version>1.14.0</version>
          </plugin>
          <!-- fb-contrib (optional extra detectors) -->
          <plugin>
            <groupId>com.mebigfatguy.fb-contrib</groupId>
            <artifactId>fb-contrib</artifactId>
            <version>7.6.4</version>
          </plugin>
        </plugins>
        <xmlOutput>true</xmlOutput>
        <xmlOutputDirectory>${project.build.directory}</xmlOutputDirectory>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>spotbugs</goal>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

This writes `target/spotbugsXml.xml` for Jenkins and `target/site/spotbugs.html` for local viewing.

6.2 Gradle (alternative)

```
plugins { id 'com.github.spotbugs' version '6.2.5' }

spotbugs {
  effort = 'max'
  reportLevel = 'low'
}

tasks.withType(com.github.spotbugs.snom.SpotBugsTask).configureEach {
  reports.create('xml') { required = true; outputLocation = file("$buildDir/
reports/spotbugs/spotbugs.xml") }
  reports.create('html') { required = true }
}

dependencies {
  spotbugsPlugins 'com.h3xstream.findsecbugs:findsecbugs-plugin:1.14.0'
  spotbugsPlugins 'com.mebigfatguy.fb-contrib:fb-contrib:7.6.4'
}
```

6.3 Jenkinsfile (Declarative Pipeline)

```
pipeline {
  agent any
  tools { jdk 'jdk17'; maven 'maven3' }
  stages {
    stage('Build') {
      steps { sh 'mvn -B -DskipTests clean package' }
    }
    stage('Static Analysis') {
      steps { sh 'mvn -B spotbugs:spotbugs spotbugs:check -
Dspotbugs.effort=Max -Dspotbugs.threshold=Low' }
    }
  }
  post {
    always {
      recordIssues tools: [spotBugs(pattern: '**/spotbugsXml.xml')],
qualityGates: [[threshold: 1, type: 'TOTAL', unstable: true]]
      archiveArtifacts artifacts: '**/target/spotbugsXml.xml,**/target/site/
spotbugs.html', fingerprint: true
    }
    }
  }
```

What to show: Jenkins build → “Static Analysis Warnings” → SpotBugs tab → trend graph, categories, new vs total issues, drill-down to file & line.

6.4 Iterate

- Fix one or two issues (e.g., `NullDeref.lenOrZero`) → commit → Jenkins reruns → show delta (New issues = 0, Total reduced).

---

7) Slides – quick outline you can paste into PowerPoint/Google Slides

1. **Title** – FindBugs (SpotBugs) Eclipse plugin: Manual & Automated Verification
2. **Why Quality** – Cost of defects curve; goals; scope
3. **Where Static Analysis Fits** – V-model or SDLC swimlane
4. **FindBugs/SpotBugs Overview** – bytecode, bug patterns, ranks, confidence
5. **Install** – update-site URLs; plugin enablement; add-ons (FindSecBugs, fb-contrib)
6. **Features** – categories, filters, suppression, integrations
7. **Demo 1** – Eclipse run & fixes (with screenshots placeholders)
8. **Demo 2** – Jenkins pipeline & dashboards
9. **Compare Auto vs Manual** – side-by-side table
10. **Good Practices** – gate on *Scary/High* first; tune filters; educate via review
11. **Appendix** – links, references, troubleshooting

---

8) Speaker notes (cheat sheet)

- Stress that **static analysis ≠ code review**; they complement each other. Use tools to **reduce review noise** so humans focus on design.
- Start with **Low threshold & Max effort** in CI to discover scope, then ratchet gates.
- Treat false positives seriously: suppress with justification, adjust filters, or upgrade rules.
- Add **security** early (FindSecBugs) – easy wins.
- Show **before/after** Jenkins trend to prove impact.

---

9) Troubleshooting

- No findings? Ensure project is **built** (class files exist) and SpotBugs is **enabled**.
- Jenkins shows “no parser found”? Ensure XML path matches (`spotbugsXml.xml`) and publish via Warnings NG.
- Gradle plugin only outputs XML *or* HTML by default; configure tasks or use helper plugins to produce both.
- Many false positives in tests? Exclude `**/*Test*.java` via SpotBugs filters or configuration.

---

10) Extra: Example SpotBugs filter files

`spotbugs-exclude.xml`

```
<FindBugsFilter>
  <Match>
    <Class name="~.*Test.*"/>
  </Match>
</FindBugsFilter>
```

`spotbugs-include-security.xml`

```
<FindBugsFilter>
  <Match>
    <Bug category="SECURITY"/>
  </Match>
</FindBugsFilter>
```

---

11) What to say about other tools (1 slide)

- **Codacy, CodeFactor, Code Climate/Qlty, SonarQube/SonarCloud**: hosted or self-hosted code quality platforms; integrate multiple linters & provide PR annotations, dashboards, and policies.
- When to choose them: multi-language repos, policy-as-code, org-wide governance, developer UX for PR comments.

End of pack – good luck with your presentation!