1. Explain more about PMD static code analysed and how to define custom rules to get call hierarchy
2. **ChatGPT said:**

PMD (Programming Mistake Detector) is a popular static code analysis tool that identifies programming flaws, potential bugs, code smells, and rule violations in Java and other languages. It operates by analyzing source code without executing it and uses predefined or custom rules to check for issues.

**Key Features of PMD:**

* 1. **Rule-based analysis**: Uses rules written in XML or Java for identifying specific patterns in code.
  2. **Multi-language support**: Primarily supports Java but also works with Apex, JavaScript, XML, and others.
  3. **Customizable**: Developers can define their own rules for unique use cases.
  4. **Integration**: Integrates with build tools like Maven, Gradle, and CI/CD pipelines.
  5. **Open-source**: Free to use under an open-source license.

**Defining Custom Rules in PMD**

Creating custom rules for PMD involves writing rules in XML or Java to define specific patterns or conditions you want to check in your codebase.

**1. Writing Custom XML Rules**

* 1. PMD's XML-based ruleset allows you to specify rules using XPath expressions.
  2. XPath can traverse the Abstract Syntax Tree (AST) generated for your code and match patterns.

**Example of a custom rule in XML**:

xml

Copy code

<ruleset name="Custom Rules" xmlns="https://pmd.github.io/ruleset/2.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://pmd.github.io/ruleset/2.0.0 https://pmd.github.io/ruleset\_2\_0\_0.xsd"> <rule name="AvoidPrintStackTrace" language="java" message="Avoid using printStackTrace; use a logger instead." class="net.sourceforge.pmd.lang.rule.XPathRule"> <description> Identifies calls to printStackTrace(). </description> <priority>3</priority> <properties> <property name="xpath"> <value>//MethodCall/Name[@Image='printStackTrace']</value> </property> </properties> </rule> </ruleset>

Save this as an .xml file, and reference it in your PMD configuration.

**2. Defining Custom Rules in Java**

* 1. If XPath cannot express the complexity of your custom rule, you can write rules in Java.
  2. This involves extending the AbstractJavaRule class and overriding its visit methods to define rule logic.

**Example of a Java rule**:

java

Copy code

import net.sourceforge.pmd.lang.java.ast.ASTMethodCall; import net.sourceforge.pmd.lang.java.rule.AbstractJavaRule; public class AvoidPrintStackTraceRule extends AbstractJavaRule { @Override public Object visit(ASTMethodCall node, Object data) { if ("printStackTrace".equals(node.getMethodName())) { addViolation(data, node); } return super.visit(node, data); } }

Compile and package the custom rule into a JAR and configure PMD to use it.

**Building a Call Hierarchy Rule**

To build a rule that identifies call hierarchies:

* 1. **Use XPath**:
     + Write an XPath expression to identify specific method calls or trace relationships in the AST.

**Example XPath**:

xml

Copy code

//ClassOrInterfaceBody//MethodDeclaration/Block//MethodCall

This locates method calls within a specific method.

* 1. **Use Java for Complex Hierarchies**:
     + Write a Java rule that:
       - Traverses AST nodes to track method calls.
       - Logs or maps relationships between caller and callee methods.
       - Optionally builds a visual call graph using libraries like GraphViz.

**Integrating Custom Rules with PMD**

* 1. Add the custom rules XML or JAR file to your project.
  2. Update the PMD configuration file (pmd-ruleset.xml) to include the new rules.

xml

Copy code

<ruleset> <rule ref="path/to/custom-rules.xml" /> </ruleset>

* 1. Run PMD with your rules:

sh

Copy code

pmd -d <source-directory> -R <path-to-ruleset.xml> -f text

By defining custom rules, you can enforce coding standards, prevent specific issues, and gain insights into patterns like method call hierarchies within your codebase.

1. **You said:**

Share Java code to define custom rule to find call hierarchy that can build using java8

1. **ChatGPT said:**

Creating a custom PMD rule in Java to find the call hierarchy involves extending PMD's AbstractJavaRule class. This rule will analyze the Abstract Syntax Tree (AST) and log method calls and their relationships.

Here’s an example Java code for a custom PMD rule to find call hierarchies:

**Java Custom Rule: Call Hierarchy Detector**

java

Copy code

import net.sourceforge.pmd.lang.java.ast.ASTClassOrInterfaceDeclaration; import net.sourceforge.pmd.lang.java.ast.ASTMethodDeclaration; import net.sourceforge.pmd.lang.java.ast.ASTMethodCall; import net.sourceforge.pmd.lang.java.rule.AbstractJavaRule; import java.util.HashMap; import java.util.List; import java.util.Map; public class CallHierarchyRule extends AbstractJavaRule { // Map to store the call hierarchy: Caller -> List of Callees private final Map<String, String> callHierarchy = new HashMap<>(); @Override public Object visit(ASTClassOrInterfaceDeclaration node, Object data) { String className = node.getImage(); node.findDescendantsOfType(ASTMethodDeclaration.class).forEach(method -> { String methodName = method.getName(); String caller = className + "." + methodName; // Find all method calls inside this method List<ASTMethodCall> methodCalls = method.findDescendantsOfType(ASTMethodCall.class); for (ASTMethodCall methodCall : methodCalls) { String callee = methodCall.getMethodName(); if (callee != null) { callHierarchy.put(caller, calle); } } }); return super.visit(node, data); } @Override // New Visit code combined there . }``

Here’s the complete and corrected implementation to traverse method calls and build a call hierarchy in Java using PMD:

**Full Implementation**

java

Copy code

import net.sourceforge.pmd.lang.java.ast.ASTClassOrInterfaceDeclaration; import net.sourceforge.pmd.lang.java.ast.ASTMethodDeclaration; import net.sourceforge.pmd.lang.java.ast.ASTMethodCall; import net.sourceforge.pmd.lang.java.rule.AbstractJavaRule; import java.util.\*; public class CallHierarchyRule extends AbstractJavaRule { // Map to store the call hierarchy: Caller -> List of Callees private final Map<String, List<String>> callHierarchy = new HashMap<>(); @Override public Object visit(ASTClassOrInterfaceDeclaration node, Object data) { String className = node.getImage(); // Get the class name // Process all methods in the class List<ASTMethodDeclaration> methods = node.findDescendantsOfType(ASTMethodDeclaration.class); for (ASTMethodDeclaration method : methods) { String methodName = method.getName(); String caller = className + "." + methodName; // Find all method calls within this method List<ASTMethodCall> methodCalls = method.findDescendantsOfType(ASTMethodCall.class); for (ASTMethodCall methodCall : methodCalls) { String callee = methodCall.getMethodName(); if (callee != null) { callHierarchy.computeIfAbsent(caller, k -> new ArrayList<>()).add(callee); } } } return super.visit(node, data); } @Override public Object visit(ASTMethodDeclaration node, Object data) { // Optional: Add additional logic to handle methods directly return super.visit(node, data); } @Override public void end(RuleContext ctx) { // Output the call hierarchy callHierarchy.forEach((caller, callees) -> { StringBuilder output = new StringBuilder(caller + " calls: "); output.append(String.join(", ", callees)); ctx.getReport().addRuleViolation( createViolation(ctx, 0, output.toString()) ); }); super.end(ctx); } }

**Key Points of the Implementation**

* 1. **Class Structure**:
     + The class extends AbstractJavaRule to enable custom rule creation for PMD.
  2. **Class-Level AST Traversal**:
     + ASTClassOrInterfaceDeclaration is used to iterate through classes and interfaces in the Java source code.
  3. **Method Traversal**:
     + For each class, all methods are analyzed using ASTMethodDeclaration.
  4. **Method Call Detection**:
     + Inside each method, all method calls are extracted using ASTMethodCall.
  5. **Building Call Hierarchy**:
     + A Map is used to store caller-callee relationships. The caller is represented by the class and method name, and the callees are added to a list.
  6. **End Rule Handling**:
     + In the end method, the hierarchy is printed or added as rule violations for PMD reporting.

**Packaging and Testing the Rule**

* 1. **Compile the Rule**:
     + Package the Java rule into a JAR file.
  2. **Add the Rule to PMD**:
     + Include the custom JAR in your PMD classpath.
  3. **Create RuleSet XML**:
     + Reference the custom rule in a PMD ruleset file:

xml

Copy code

<ruleset name="Custom Rules" xmlns="https://pmd.github.io/ruleset/2.0.0" xsi:schemaLocation="https://pmd.github.io/ruleset/2.0.0 https://pmd.github.io/ruleset\_2\_0\_0.xsd"> <description>Custom rules for call hierarchy</description> <rule name="CallHierarchyRule" language="java" class="your.package.CallHierarchyRule" /> </ruleset>

* 1. **Run PMD**:
     + Execute PMD with your source directory and ruleset:

bash

Copy code

pmd -d <source-directory> -R <ruleset.xml> -f text

**Output Example**

For a class like:

java

Copy code

public class Example { public void methodA() { methodB(); methodC(); } public void methodB() {} public void methodC() {} }

The output might look like:

yaml

Copy code

Example.methodA calls: methodB, methodC

This rule can be further extended to log the entire call chain or handle external dependencies.

1. **You said:**

Share complex example to get entire call chain when some Of the classes are part of jar files and use as class path

1. **ChatGPT said:**

To create a PMD rule that generates the **entire call chain**, including calls to methods in classes within external JAR files (classpath dependencies), you'll need to:

* 1. Parse the source code to extract direct method calls.
  2. Resolve method calls from the classpath using a **bytecode analysis library** like **ASM** or **BCEL** to handle external JAR dependencies.
  3. Traverse the call hierarchy recursively until no further callees are found.

Here’s how to achieve this:

**Full Implementation for Call Chain Analysis**

java

Copy code

import net.sourceforge.pmd.lang.java.ast.\*; import net.sourceforge.pmd.lang.java.rule.AbstractJavaRule; import java.io.IOException; import java.util.\*; import java.util.jar.JarEntry; import java.util.jar.JarFile; import java.lang.reflect.Method; import java.net.URLClassLoader; import java.net.URL; public class CallChainRule extends AbstractJavaRule { // Map to store the call chain private final Map<String, Set<String>> callChain = new HashMap<>(); // Class loader to resolve external classes private ClassLoader externalClassLoader; public CallChainRule() { try { // Initialize class loader with external JAR files externalClassLoader = new URLClassLoader(new URL[]{ new URL("file:///path/to/external.jar") }); } catch (Exception e) { e.printStackTrace(); } } @Override public Object visit(ASTClassOrInterfaceDeclaration node, Object data) { String className = node.getImage(); // Process all methods in the class List<ASTMethodDeclaration> methods = node.findDescendantsOfType(ASTMethodDeclaration.class); for (ASTMethodDeclaration method : methods) { String methodName = className + "." + method.getName(); resolveMethodCalls(method, methodName); } return super.visit(node, data); } private void resolveMethodCalls(ASTMethodDeclaration method, String caller) { // Find all method calls in the current method List<ASTMethodCall> methodCalls = method.findDescendantsOfType(ASTMethodCall.class); for (ASTMethodCall methodCall : methodCalls) { String callee = methodCall.getMethodName(); if (callee != null) { callChain.computeIfAbsent(caller, k -> new HashSet<>()).add(callee); // Attempt to resolve external methods if applicable resolveExternalMethodCall(caller, methodCall); } } } private void resolveExternalMethodCall(String caller, ASTMethodCall methodCall) { String className = methodCall.getQualifier() != null ? methodCall.getQualifier().getImage() : null; if (className != null) { try { // Load external class Class<?> clazz = externalClassLoader.loadClass(className); for (Method method : clazz.getDeclaredMethods()) { if (method.getName().equals(methodCall.getMethodName())) { // Add to call chain String callee = className + "." + method.getName(); callChain.computeIfAbsent(caller, k -> new HashSet<>()).add(callee); // Optionally recurse into the external method's body if available // This requires additional bytecode analysis using ASM/BCEL. } } } catch (ClassNotFoundException e) { // Log or handle the missing class System.err.println("Class not found: " + className); } } } @Override public void end(RuleContext ctx) { // Print the full call chain for (Map.Entry<String, Set<String>> entry : callChain.entrySet()) { StringBuilder output = new StringBuilder(entry.getKey() + " calls: "); output.append(String.join(", ", entry.getValue())); ctx.getReport().addRuleViolation( createViolation(ctx, 0, output.toString()) ); } super.end(ctx); } }

**Steps Explained**

* 1. **Class Loader for External JARs**:
     + A custom URLClassLoader loads external JARs to resolve method calls in those classes.

java

Copy code

new URLClassLoader(new URL[]{ new URL("file:///path/to/external.jar") });

* 1. **Resolving Method Calls**:
     + Within each method, direct method calls are resolved using AST.
     + For external calls, the class name is extracted, and reflection (ClassLoader.loadClass) resolves the target method.
  2. **Call Chain Tracking**:
     + A Map<String, Set<String>> stores the caller-to-callee relationships.
  3. **Handling Recursion**:
     + You can extend the implementation to analyze the method bodies in external classes. This requires bytecode analysis tools like **ASM** or **BCEL** to decompile and traverse the bytecode.

**Advanced: Analyzing Method Bodies in JARs**

If you want to analyze the **implementation of methods in external JARs**, integrate a bytecode library like ASM:

xml

Copy code

<dependency> <groupId>org.ow2.asm</groupId> <artifactId>asm</artifactId> <version>9.5</version> </dependency>

Then, use ASM to parse .class files inside the JAR to find methods and their calls:

java

Copy code

private void analyzeMethodBytecode(String classFilePath) { try (JarFile jarFile = new JarFile("/path/to/external.jar")) { Enumeration<JarEntry> entries = jarFile.entries(); while (entries.hasMoreElements()) { JarEntry entry = entries.nextElement(); if (entry.getName().endsWith(".class")) { ClassReader classReader = new ClassReader(jarFile.getInputStream(entry)); classReader.accept(new ClassVisitor(Opcodes.ASM9) { @Override public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[] exceptions) { return new MethodVisitor(Opcodes.ASM9) { @Override public void visitMethodInsn(int opcode, String owner, String name, String descriptor, boolean isInterface) { // Add to call chain String callee = owner.replace('/', '.') + "." + name; callChain.computeIfAbsent("ExternalMethod", k -> new HashSet<>()).add(callee); } }; } }, 0); } } } catch (IOException e) { e.printStackTrace(); } }

**Sample Output**

For the following setup:

* 1. Class A in your source calls B.method1() and C.method2() in external JARs.
  2. Class B calls D.method3().

The output will be:

less

Copy code

A.method calls: B.method1, C.method2 B.method1 calls: D.method3

**Final Notes**

* 1. To handle deeply nested call chains, recursively analyze methods and expand the call graph.
  2. Integrating bytecode analysis tools like **ASM** is essential to fully understand external methods when source code is unavailable.
  3. Ensure that your PMD environment can dynamically load and execute the custom rule.

1. **You said:**

Does ASM supports java8

1. **ChatGPT said:**

Yes, **ASM** fully supports Java 8 and all its bytecode features. ASM is a widely used bytecode manipulation and analysis library that supports Java versions from 1.0 up to the latest versions, including Java 8. It can handle Java 8 features such as:

* 1. **Lambda Expressions**: Java 8 introduced lambdas, which are compiled to use the invokedynamic instruction. ASM supports this instruction and can analyze or manipulate lambda-related bytecode.
  2. **Default Methods in Interfaces**: ASM can analyze and modify interfaces with default methods introduced in Java 8.
  3. **Annotations**: ASM supports parsing annotations, including those with new retention policies or target types introduced in Java 8.
  4. **Method Parameter Reflection**: ASM supports the new MethodParameter information in the bytecode for accessing parameter names and modifiers.

**ASM Compatibility with Java 8 Bytecode**

When using ASM for Java 8, make sure to:

* 1. Use an appropriate version of ASM that supports Java 8 bytecode (invokedynamic and other instructions).
  2. ASM versions from **ASM 5.0** onward fully support Java 8.

**Maven Dependency for ASM**

To include ASM in your project, use the following Maven dependency:

xml

Copy code

<dependency> <groupId>org.ow2.asm</groupId> <artifactId>asm</artifactId> <version>5.0</version> </dependency>

For later versions that also support newer Java versions, you can use a more recent ASM version (e.g., 9.5).

**Example: Parsing a Java 8 Lambda Using ASM**

Here’s an example of using ASM to analyze bytecode with a lambda expression:

java

Copy code

import org.objectweb.asm.\*; import java.io.InputStream; public class Java8LambdaAnalyzer { public static void main(String[] args) throws Exception { // Load a class containing a lambda InputStream classStream = Java8LambdaAnalyzer.class.getResourceAsStream("/YourClassWithLambda.class"); // Parse the bytecode using ASM ClassReader reader = new ClassReader(classStream); reader.accept(new ClassVisitor(Opcodes.ASM9) { @Override public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[] exceptions) { return new MethodVisitor(Opcodes.ASM9) { @Override public void visitInvokeDynamicInsn(String name, String descriptor, Handle bootstrapMethodHandle, Object... bootstrapMethodArguments) { System.out.println("Lambda found: " + name + " with descriptor: " + descriptor); } }; } }, 0); } }

**Key Points for Java 8 with ASM**

* 1. Ensure you’re using ASM version 5.0 or later.
  2. Java 8 bytecode features like invokedynamic (for lambdas) and parameter metadata are fully supported.
  3. ASM can be used to analyze, modify, or generate Java 8-compliant bytecode efficiently.

1. **You said:**

Maven arche type supports java8

1. **ChatGPT said:**

Yes, **Maven archetypes** can be configured to support Java 8 by specifying the Java version in the generated project's **POM file**. Maven itself does not impose any restrictions on Java versions, so it fully supports Java 8 as long as you configure it correctly.

**Steps to Create a Java 8-Compatible Maven Project**

* 1. **Use a Maven Archetype**:
     + Run the following command to create a new Maven project using an archetype:

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=java8-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

* 1. **Configure Java 8 in**pom.xml:
     + Update the generated pom.xml to set the Java version to 8:

xml

Copy code

<build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <version>3.8.1</version> <configuration> <source>1.8</source> <target>1.8</target> </configuration> </plugin> </plugins> </build>

* + - The <source> and <target> tags specify the Java version for compilation. Use 1.8 for Java 8 compatibility.
  1. **Verify Java 8 Environment**:
     + Ensure Java 8 is installed and available in your environment.
     + Check your Maven installation is pointing to Java 8:

bash

Copy code

mvn -v

Output should include something like:

javascript

Copy code

Java version: 1.8.x Java home: /path/to/java8

* 1. **Run the Maven Project**:
     + Compile and test the project to confirm Java 8 compatibility:

bash

Copy code

mvn clean install

**Maven Archetypes Supporting Java 8**

Some common archetypes for starting Java 8 projects:

* 1. **Quickstart Archetype**:

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=java8-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

* 1. **Web Application Archetype** (for Java 8 web apps):

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=java8-web-app -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

* 1. **Spring Boot Archetype**: For Java 8 Spring Boot projects:

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=spring-boot-java8-app -DarchetypeGroupId=org.springframework.boot -DarchetypeArtifactId=spring-boot-sample-simple-archetype -DinteractiveMode=false

**Setting Default Java Version for Maven**

If you have multiple Java versions installed and need Maven to use Java 8 consistently:

* 1. Set the JAVA\_HOME environment variable to point to your Java 8 installation:

bash

Copy code

export JAVA\_HOME=/path/to/java8 export PATH=$JAVA\_HOME/bin:$PATH

* 1. Verify Maven uses the correct Java version:

bash

Copy code

mvn -v

With these steps, Maven archetypes can be used to set up and manage Java 8-compatible projects seamlessly.