UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Ingegneria del Software**

Strumenti dell'Analisi Statica del Software

Anno Accademico 2017/2018

Candidato:

Claudio Moscato

matr. N46/032

[Dedica]

Indice

Introduzione

Quasi tutti i software contengono difetti. Alcuni difetti si trovano facilmente mentre per altri non è così semplice, in genere perché emergono raramente o per niente. Alcuni difetti emergono relativamente spesso ma passano altrettanto spesso inosservati semplicemente perché non sono percepiti come errori o non sono sufficientemente gravi. I difetti del software possono dar luogo a diversi tipi di errori, che vanno da quelli logici / funzionali (il programma a volte calcola valori errati) agli errori di runtime (il programma tipicamente si blocca), o perdite di risorse.

I difetti del software che emergono improvvisamente possono causare danni estremamente ingenti, in particolare se trovati alla fine del ciclo di sviluppo o, peggio ancora, dopo il rilascio del software. Molti semplici difetti nei programmi sono identificati direttamente dai moderni compilatori, ma il metodo predominante per trovare i difetti è il testing “a scatola chiusa”. I test hanno il potenziale di trovare la maggior parte di difetti, tuttavia, i test sono costosi e nessuna quantità di test troverà tutti i difetti. Anche il test è problematico perché può essere applicato solo all'eseguibile del codice, cioè piuttosto tardi nel processo di sviluppo. Alternative al test, come ad esempio analisi dei flussi di dati e verifica formale, sono noti fin dagli anni '70, ma non hanno ottenuto un'accettazione diffusa al di fuori del mondo accademico, almeno fino a una decina di anni fa.. Ultimamente sono emersi diversi strumenti per il rilevamento delle condizioni di errore a runtime e a tempo di compilazione. Gli strumenti si basano sull'analisi statica e possono essere utilizzati per individuare errori di runtime, codice morto, codice duplicato, risorse non utilizzate o utilizzate male, il tutto senza eseguire il codice.

Questo lavoro di tesi descrive e mette a confronto tre strumenti di analisi statica Open

Source: PMD, Checkstyle e Spotbugs.

L'obiettivo principale di questo studio è la sperimentazione dei tre software applicata a tipici programmi degli scritti dagli studenti, sotto l'ambiente di sviluppo Eclipse. L'obiettivo non è quello di fornire una classifica degli strumenti né di fornire una panoramica completa di tutte le funzionalità degli strumenti. Nel capitolo 1 trattiamo in generale l'analisi statica del codice e le maggiori tecniche utilizzate. Nel capitolo 2-3-4 descriviamo le funzionalità di base fornite dai tre strumenti: PMD, Chekstyle, SpotBugs in particolare quando utilizzati a supporto di codice sorgente Java sviluppate tramite l'ausilio di Eclipse:

Capitolo 1: Analisi Statica del Codice

Quasi la totalità del software (se non tutto) contiene dei *difetti*. Alcuni vengono individuati molto semplicemente, dopo una attenta lettura del codice, o come segnalazioni dei processi di compilazione e linking. Altri invece, risultano più insidiosi e sono difficilmente individuati prima del rilascio e della fase di esercizio del software. Altri ancora, emergono raramente o potrebbero non essere mai scoperti. Addirittura, alcuni difetti potrebbero passare inosservati non perchè mai riscontrati, ma perchè nella maggior parte dei contesti potrebbero non essere percepiti come errori (o come difetti *sufficientemente* severi).

A seconda della gravità e della natura dei *difetti*, delle condizioni di esecuzione e delle specifiche del software, i difetti possono o meno generare *errori*, che possono portare anche ad effetti catastrofici nel caso in cui il software sia eseguito su sistemi critici. In questo caso è chiaro che l'individuazione dei difetti sul software e dei possibili loro effetti deve essere parte integrante di ogni processo di sviluppo del software.

Generalmente, l'approccio più utilizzato per l'individuazione di difetti nel software, è il *Testing*. Le procedure di testing (black box) cercano di individuare errori nel software, sottomettendo diversi input al sistema e verificando (tramite un"oracolo" che compara gli output ottenuti con quelli attesi) che gli output (esistano e) siano coerenti con le specifiche. Purtroppo le procedure di testing richiederebbero la verifica di tutti i possibili input per tutti i possibili componenti del sistema. Questa, oltre che essere una procedura costosa in termini di tempo e risorse (si stima che il testing occupi circa l'80% del tempo e delle risorse necessarie alla progettazione e sviluppo del software). Hanno inoltre la proprietà di individuare la *presenza* di errori nel software, ma non la loro totale assenza: è infatti stato dimostrato che il testing *esaustivo* non è attuabile su nessuna tipologia di software. Inoltre, il testing difficilmente rileverebbe una serie di difetti "subdoli", il cui comportamento non

era stato previsto e dettagliato all'atto della definizione delle specifiche del sistema.

Al Testing Black Box, spesso si affiancano altri tipi di analisi del software, tra cui si vuole annoverare l'Analisi Statica del Codice.

L'analisi statica del codice (spesso indirizzata con il nome di *Testing White Box*), analizzano un programma senza eseguirlo. Il termine "white box" deriva dal fatto che, in genere, l'analisi statica viene eseguita analizzando con specifiche tecniche il codice sorgente. Le proprietà e di difetti analizzabili sono simili a quelli che un "umano" potrebbe ritrovare durante le fasi di "code reviewing"[1]. L'analisi Statica del Codice è eseguita in maniera totalmente automatizzata e sfrutta alcune caratteristiche peculiari del processo di compilazione e linking del codice sorgente.

L'attuale e predominante architettura di compilatori prevede tre fasi principali, così come descritto in Figura 1

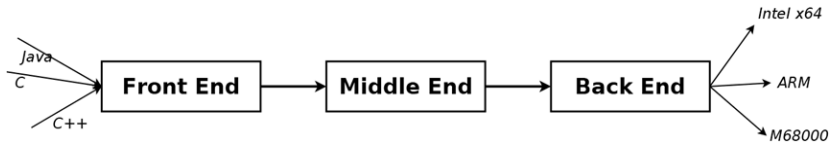


Figura 1: Fasi di Compilazione

La fase di *Front End* effettua verifiche di tipo, sintattiche e semantiche sul codice sorgente, includendo le fasi di analisi lessicale e sintattica. L'output di tale fase consiste in una *rappresentazione intermedia (RI)* del compilatore, che descrive il programma in un linguaggio indipendente dall'architettura target, e che viene poi utilizzata nelle fasi successive. Tra le strutture dati utilizzate in questa fase si ricorda l'albero di sintassi (*syntax tree*) e l'albero di sintassi astratto (*abstract syntax tree*)[1.1], che fornisce le basi per la costruzione della RI.

[1]Le procedure di "Code Review" richiedono l'ispezione diretta da parte di *umani* del codice sorgente al fine di identificarne punti di debolezza, difetti, vulnerabilità. Visto la caratteristica degli analizzatori statici di automatizzare alcuni processi di code reviewing, si tende ad interpretare l'analisi statica come una delle fasi del code reviewing.

La RI viene quindi utilizzata nella fase di Middle End, al fine di analisi e ottimizzazione. Per lo più, l'analisi statica del codice avviene in questa fase e, basandosi sulla RI della fase di front-end, produce una serie di rappresentazioni interne utili ai fini delle analisi del codice.

Le analisi svolte dagli analizzatori statici sono volte ad analizzare diverse proprietà del software, che vanno dal semplice controllo di indentazione del codice, alla ricerca di codice morto (*dead code*)[2], dall'analisi di comuni errori di programmazione che non risultano in errori di compilazione[3] , all'individuazione di complicati pattern di istruzioni che possono portare a gravi falle di sicurezza. Le strutture dati utilizzate a tal scopo, comprendono il Control Flow Graph, di cui parleremo in breve in seguito.

Fanno poi parte della fase di Middle End, anche le ottimizzazioni come la ridistribuzione delle istruzioni nei cicli o le propagazioni delle costanti.

La fase di Back End si occupa della traduzione delle rappresentazioni intermedie in codice macchina (sulla architettura target). Sono incluse in questa fase ulteriori ottimizzazioni che riguardano il processore target, come l'ottimizzazione dei registri macchina, della cache, delle pipeline etc.

1.1 Tecniche per L'Analisi Statica

Le principali proprietà analizzate durante i processi di analisi statica, avvengono tramite l'analisi dell'Abstract Sintax Tree, o grazie alla costruzione di appropriate strutture dati.

Alcune delle principali tecniche utilizzate nei sistemi di analisi statica sono [1.2]:

- Interpretazione Astratta: vista la complessità del sistema da analizzare (software e ambiente di esecuzione), si crea una rappresentazione astratta del sistema che ne

[2]Il Codice Morto è una porzione di codice sorgente che *non verrà mai eseguito*. Se non previsto, in genere l codice morto è sintomo di gravi difetti del codice. Altre volte il codice morto viene introdotto intenzionalmente, ad esempio quando si vuole disattivare delle funzionalità del codice, o quando si vuole produrre codice che sia compilabile su più architetture (si parla in questo caso di *codice disattivato*) . Si fa notare che nell'ultimo caso, non esiste traccia del codice sorgente nel codice oggetto generato dal compilatore.

[3]come ad esempio, le istruzioni di assegnazione dei blocchi condizione degli *if*

approssimi il funzionamento (senza introdurne di nuovi), almeno per quanto riguarda le proprietà da analizzare

- Analisi Data-Flow: Una analisi che si basa sulla visita di opportuni grafi e reticoli. Dato ogni insieme di istruzioni che non possa essere interrotto da salti o altri tipi di variazioni del sequenziale flusso di controllo (*basic block*) si costruisce un grafo i cui nodi sono identificati dai basic block e dai loro effetti sulle variabili, mentre gli archi del grafo connettono due basic-block se esiste un percorso sul flusso di controllo (un "salto") che porti ad eseguire in sequenze (in genere condizionalmente) diversi basic-blocks. Questo grafo prende il nome di Control Flow Graph e la sua analisi è alla base per la valutazione di proprietà statiche come, ad esempio, l'analisi delle catene *definizione-uso* delle variabili, o per la valutazione di alcune *metriche del software* come il *numero ciclomatico* di McCabe.
- Sistemi formali che si basano su regole e logiche di Hoare[4]
- Model Checking: queste tecniche utilizzano delle rappresentazioni astratte dei sistemi (modelli), e su queste cercano di valutare in maniera esaustiva, se nell'inter spazio di stato del modello valgano o meno delle proprietà logiche (per esempio, la raggiungibilità di ogni istruzione di un programma rispetto ad ogni possibile esecuzione è una proprietà che viene esaustivamente verificata tramite queste procedure)
- Esecuzione Simbolica: tecniche che, in breve, cercano di determinare quali input servono per determinare l'esecuzione di (ogni) parte del programma.

[4]Le logiche di Hoare si basano triplette di tipo : {P} C {Q} dove P e Q sono predicati che definiscono in genere cosa succede ad un insieme di variabili quando si esegue il comando C dove P definisce l'insieme di variabili prima dell'esecuzione del comando, e Q ne descrive gli effetti. Le triplette possono poi essere composte tramite l'utilizzo di *regole* che definiscono i comandi più complessi attraverso operazioni più semplici. Le logiche di Hoare vengono comunemente utilizzate per descrivere le semantiche operazionali dei linguaggi di programmazione [1.3]

1.2 Pro e Contro

Sicuramente, tra i punti a favore dei sistemi per l'analisi statica del codice, c'è la loro alta *scalabilità*: le procedure di analisi del codice si adattano facilmente a software di bassa o alta complessità. Una volta definita una tecnica per l'individuazione di difetti del software, questa può essere riapplicata ad ogni modifica del software. Infatti, la maggior parte dei tools per l'analisi statica sono integrati nei framework per lo sviluppo del software, ed eseguono la maggior parte delle analisi ad ogni ricompilazione (cosa che nei moderni strumenti di sviluppo avviene praticamente ad ogni modifica o aggiunta di linea di codice nel software).

Alcuni strumenti sono sufficientemente maturi da avere un *elevato livello di affidabilità* sull'individuazione dei più comuni difetti del software: buffer overflow, SQL injection, valutazione di alcune metriche del software, evidenti applicazioni di errati pattern di programmazione.

I principali problemi dei metodi di analisi statica, risiedono nella loro caratteristica fondamentale: l'automazione dei processi di analisi. Alcuni difetti che incidono nella sicurezza dei sistemi software (protocolli e sistemi di autenticazione, crittografia etc.) sono così complessi da analizzare che sono difficilmente automatizzabili, o lo sono nella loro applicazione generale.

Altro problema è che tutti i difetti dipendenti dalla configurazione del software non vengono individuati dai sistemi di analisi statica.

Ancora, ma cosa più importante, i sistemi di analisi statica lavorano a valle delle prime fasi di compilazione, quindi risulta estremamente complicato (o addirittura impossibile) analizzare software che non viene compilato, o che assume configurazioni diverse in funzione dei parametri di configurazione, come ad esempio accade per i Template di C++ (e i generics di Java), o nei linguaggi che non vengono compilati o che, al limite, supportano

solo quella che viene chiamata compilazione“Just in Time” (come accade ad esempio per il Python ed altri linguaggi interpretati).

Infine, bisogna ricordare che i sistemi di analisi statica, specialmente quelli meno maturi o quelli che operano su più linguaggi (e quindi lavorano su astrazioni a livello più alto), soffrono del problema di rilevare un discreto numero di *falsi positivi* e *falsi negativi*. La notifica di un falso positivo equivale a identificare un difetto che in realtà non esiste, o di identificare un problema che in realtà non è tale (ad esempio: codice che *volutamente* viene inserito nel software e che rispetta tutte le caratteristiche di un *difetto*). I sistemi di analisi statica che analizzano codice che viene integrato con componenti “closed source” hanno in genere un’alta percentuale di falsi positivi dovuti al fatto che non possono analizzare alcune parti del software.

I falsi negativi si verificano quando uno strumento di analisi statica non rileva un difetto che in realtà esiste nel software. In genere questo è dovuto all'alta specializzazione dei singoli strumenti verso l’individuazione di determinati difetti e pattern (oltre al fatto, ovviamente, che non esistano ancora regole e metodi ben definiti in uno strumento per l’individuazione di un tipo di difetto o vulnerabilità scoperto recentemente). In questo caso è cruciale la scelta dello strumento di analisi che meglio si accoppia alle proprietà da studiare e ai difetti da individuare.

Capitolo 2: PMD

PMD [2.1] è uno strumento Open Source per l’analisi statica del codice sorgente. Come ammesso dagli sviluppatori, la sigla non corrisponde a nessun acronimo, benché gli stessi autori suggeriscano diversi “backronyms” per la sigla (tra i quali vogliamo riportare almeno

la ricorsiva definizione: “PMD Meaning Discovery”).

Commentato [Porfirio1]: Io sapevo Programming Mistake Detector

Al momento della scrittura di questa tesi, l’ultima versione di PMD rilasciata è la 6.0.0. Il framework supporta principalmente l’analisi di codice sorgente Java, benché l’ultima versione supporti 8 linguaggi (per lo più Java-based). L’interfaccia utente di PMD è a linea di comando e i sorgenti sono compilabili sia in ambiente Linux/Unix che Microsoft Windows.

Esistono dei progetti di supporto che forniscono una interfaccia grafica per le diverse funzionalità di PMD. In particolare si vuole ricordare: *Eclipse-PMD plugin* [2.2] e *PMD Rule Designer* [2.3], interfacce grafiche che verranno descritte più avanti in questo lavoro.

PMD è un analizzatore *basato su regole (Rule-Based)*. Le regole definiscono le occorrenze di pattern di costrutti di programmazione che corrispondono (secondo gli sviluppatori di PMD) a noti difetti di programmazione. In particolare, le classi di difetti che PMD individua sono:

- Possibili bug nell’utilizzo dei principali costrutti di programmazione. In questa categoria ricadono i blocchi try/catch, finally e switch vuoti; errori nei formati delle condizioni dei blocchi condizionali (if e while)
- Codice Morto. In questa classe ricade anche l’individuazione di variabili locali non utilizzate, la definizione senza uso di parametri dei metodi, la definizione senza utilizzo di metodi privati
- Espressioni eccessivamente complicate. Tra le espressioni individuate si ricordano l’utilizzo di *if* innestati inutili,for che potrebbero essere espressi come while etc.
- Scrittura di codice Subottimo. In questa classe ricade l’individuazione di cattivi utilizzi e gestione di stringhe e buffer
- Struttura di classi troppo complicata. L’individuazione di questa classe di errori prevede il calcolo della *complessità ciclomatica* dei metodi delle classi analizzate
- Codice Duplicato. PMD supporta uno strumento separato (CPD) per l’identificazione

di codice duplicato. Il codice duplicato proveniente da operazioni di “copia e incolla”, infatti, duplicherebbe anche l’esistenza di eventuali difetti della parte copiata.

2.1 Le regole di PMD

Avendo a che fare con un analizzatore *rule based*, è di fondamentale importanza capire come funziona il sistema di regole di PMD, sia per configurare il sistema per l’utilizzo di regole già esistenti, sia per poterne implementare di nuove.

Perchè PMD possa eseguire una analisi del codice, è necessario definire una lista delle regole che devono essere usate per riconoscere i vari difetti del software analizzato. Tale lista prende il nome di *Ruleset* e consiste praticamente in un file XML che elenca quali tra le regole (predefinite) di PMD utilizzare durante la fase di analisi.

Le *Regole Predefinite* di PMD sono divise in otto categorie:

- *Best Practice*: controllano che il codice sorgente sia conforme ad una serie di “buone norme” di programmazione.(come ad esempio, evitare variabili locali o campi privati mai usati nel loro scope)
- *Code Style*: controllano che venga seguito un determinato stile di codifica;
- *Design*: Regole che aiutano ad individuare errori di progettazione del software
- *Documentation*: controllano che vengano riportate le necessarie documentazioni all’interno del codice sorgente (ad esempio, le intestazioni JavaDoc delle classi e dei metodi)
- *Error Prone*: queste regole controllano che vengano evitati dei noti pattern di programmazione che portano ad errori, specialmente a run-time (ad esempio: dichiarazioni di variabili non inizializzate, Buffer non inizializzati ne’ limitati etc.)
- *Multithreading*: individuano problemi in blocchi multithread
- *Performances*: regole che identificano codice sub-ottimo
- *Security*: analizzano possibili falle di sicurezza nel codice (possibilità di buffer

overflow etc.)

Per definire un *RuleSet*, è necessario predisporre un file XML (*ruleset.xml*) che faccia riferimento a tutte le regole base utilizzate durante la fase di analisi da PMD. Appositi tag XML nel file ruleset dichiarano le regole base utilizzate (il tag: *rule*), il path (relativo) dove trovare la dichiarazione delle regole base (la proprietà *ref* del tag rule); altri permetteranno di escludere alcune regole da una delle categorie prima elencate (sotto tag *exclude* del tag rule). Infine, altri tag (*property*) e proprietà permettono di configurare alcuni parametri delle regole base.

Un esempio di file ruleset.xml è il seguente:

```
<?xml version="1.0"?>
<ruleset name="Custom ruleset"
		xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

	<description>
		Descrizione del File RuleSet
	</description>

	<!-- Tutte le regole utilizzate -->
	<rule ref="category/java/bestpractices.xml/UnusedLocalVariable"/>
	<rule ref="category/java/bestpractices.xml/AvoidPrintStackTrace"/>

	<!-- Aggiunta di una regola e ridefinizione di proprieta -->
	<rule ref="category/java/design.xml/CyclomaticComplexity">
		<properties>
			<property name="reportLevel" value="5"/>
		</properties>
	</rule>

	<!--Importa TUTTO da una categoria, eccetto la regola x -->
	<rule ref="category/java/codestyle.xml">
		<exclude name="WhileLoopsMustUseBraces"/>
	</rule>
</ruleset>
```

Figura 2.1: Esempio di File *ruleset.xml*

Come si può notare in Figura 2.1, il ruleset (per il linguaggio java) fa riferimento a tre categorie diverse di regole predefinite (*bestpractice, codestyle* e *design*). Dalla categoria *bestpractice* verranno controllate solo le regole *UnusedLocalVariable* e *AvoidPrintStackTrace*; La categoria *codestyle* verrà utilizzata nella sua interezza tranne che per la regola *WhileLoopsMustUseBraces* che verrà esclusa dal controllo.

Esistono poi due modi per scrivere una regola che PMD possa poi verificare:

- Implementando una apposita classe Java
- Scrivendo una espressione XPATH

In realtà, la scrittura di una regola richiede la conoscenza dell'architettura di base di PMD e dell'algoritmo con cui l'analizzatore esegue le sue analisi. Comunque, sia che si implementi la regola in Java, o che lo si faccia tramite una espressione XPATH, la regola deve comunque essere dichiarata in un file XML che abbia il nome della categoria in cui ricade la regola da controllare.

```
    <rule name="UnusedLocalVariable"
          language="java"
          since="0.1"
          message="Avoid unused local variables such as ''{0}''."

class="net.sourceforge.pmd.lang.java.rule.bestpractices.UnusedLocalVariableRule"

externalInfoUrl="${pmd.website.baseurl}/pmd_rules_java_bestpractices.html#unusedlocalvari
able">
        <description>
Detects when a local variable is declared and/or assigned, but not used.
        </description>
        <priority>3</priority>
        <example>
<![CDATA[
public class Foo {
    public void doSomething() {
        int i = 5; // Unused
    }
}
]]>
        </example>
    </rule>
```

Figura 2.2: Regola *UnusedLocalVariable* di *bestpractice.xml*

Le regole hanno lo stesso formato. La Figura 2.2 riporta una regola di categoria *bestpractice* per il linguaggio java: Il tag *rule* definisce il nome e le proprietà della regola. Il linguaggio

per cui è stata scritta la regola è identificato dal valore della proprietà *language* (java in figura). La proprietà *message* definisce il messaggio che funge da tooltip per la regola. La regola in questione è definita da una classe Java: la proprietà *class* definisce il nome della classe che implementa la regola. Seguono quindi un link al manuale online di PMD, una descrizione, una priorità per l'applicazione delle regole, ed un esempio che chiarifica in che caso l'analizzatore verifica la regola in questione (nell'esempio la variabile *i* non viene mai utilizzata se non nella riga di inizializzazione).

La struttura della classe dichiarata nella regola, è funzione dell'architettura interna di PMD, che andiamo a descrivere nel prossimo sottoparagrafo.

2.1.1 Matching di regole in PMD

PMD lavora sulla base dell'output della fase di front-end di compilazione definita nel Capitolo 1. L'analizzatore infatti lavora analizzando l'output dell'*Abstract Syntax Tree (AST)* generato da un parser per il linguaggio in esame. Se ad esempio si vuole analizzare una regola di un ruleset definito per il linguaggio Java, PMD ha bisogno di costruire l'AST del modulo software da analizzare. A tal scopo, PMD utilizza un generatore di parser scritta in java: *JavaCC* [2.2] (il che non vuol necessariamente dire che PMD analizzi esclusivamente linguaggi java-based). Perchè possa generare un Parser per il linguaggio target JavaCC ha bisogno di avere a disposizione la definizione della grammatica di tale linguaggio e di definire le funzioni che il parser dovrà implementare una volta analizzato e ritrovato un costrutto grammaticale valido.

E' possibile recuperare tale definizione della grammatica dai sorgenti di PMD disponibili su github[5]. Il parser generato da JavaCC costruisce sempre l'AST del programma analizzato, e quindi invoca le apposite classi Java per il controllo della regola che, nella maggior parte

[5]Ad esempio, nella versione 6.0.0, la grammatica per il linguaggio Java, nei sorgenti, è reperibile nel path: pmd-src-6.0.0/pmd-java/etc/grammar

dei casi, si riconduce al controllo di esistenza nell'AST di una determinata struttura.

L'algoritmo implementato dall'analizzatore è il seguente:

1. Parserizza i comandi da linea di comando
2. Individua linguaggio, ruleset e files su cui eseguire l'analisi
3. Prepara il *SourceCodeProcessor* (`net.sourceforge.pmd.SourceCodeProcessor`) in funzione della configurazione
4. Parserizza il Codice Sorgente => Crea un AST per il codice parserizzato
5. Visita l'AST per realizzare una DataFlow Analysis
6. Crea il Control Flow Graph (CFG) del Programma e i nodi di data Flow (DFN)
7. Visita il CFG per l'identificazione della regola
8. "Renderizza" L'output nel formato voluto (XML, testo, HTML etc.)

Figura 2.3: Passi per la verifica di una regola in PMD

2.1.2 Definizione di regole in PMD

Per implementare una nuova regola in PMD in Java, basta tener presente il funzionamento di massima di PMD descritto in Figura 2.3. Si ricorda che tutto dipende dalla costruzione dell'AST del codice sorgente.

Supponiamo di voler scrivere una regola che segnali l'esistenza di Blocchi FOR che non abbiano corpo (come ad esempio mostrato in Figura 2.4)

```
class EsempioFor{
      void funzione() {
            for (;;);
      }
}
```

Figura 2.4: Esempio di for senza blocco.

Utilizzando l'opportuno strumento di PMD[6] per la stampa dell'AST relativo al blocco in

[6] bin/designer.sh

Figura 2.4, si ottiene l'AST mostrato sulla parte destra della Figura 2.5

Figura 2.5: AST del blocco in Figura 3.4

Risulta evidente la presenza di un ForStatement, seguito da uno Statement che è un EmptyStatement[7] .

Per definire in PMD una classe che realizzi la regola sopra descritta, bisogna:

1. Implementare una classe che erediti la classe `net.sourceforge.pmd.lang.java.rule.AbstractJavaRule`
2. Fare in modo che la classe creata abbia lo stesso nome della regola che si vuole implementare
3. Ridefinire il metodo *visit* che si occuperà della costruzione dell'AST e della sua visita (ereditando il metodo della class AbstractjavaRule)
4. Richiamare esplicitamente il metodo *addViolation* se, una volta visitato un ForStatement nell'AST, questi non ha figli che siano "Blocchi" validi (ovvero identificati da un Empty Statement)

[7]La definizione degli elementi ForStatement, Statement e EmptyStatement fa parte della grammatica del linguagigo in esame. La grammatica è espressa in EBNF (Extendend Backus - Naur Form) e contenuta nei file di grammatica di JavaCC.

```
import net.sourceforge.pmd.lang.ast.*;

import net.sourceforge.pmd.lang.java.ast.*;

import net.sourceforge.pmd.lang.java.rule.*;



public class ForNotEmpty extends AbstractJavaRule {

    public Object visit(ASTWhileStatement node, Object data) {

        Node firstStmt = node.jjtGetChild(1);

        if (hasEmptyAsFirstChild(firstStmt)) {

            addViolation(data, node);

        }

        return super.visit(node,data);

    }

    private boolean hasEmptyAsFirstChild(Node node) {

        return (node.jjtGetNumChildren() != 0 && (node.jjtGetChild(0) instanceof
ASTStatement) && (node.jjtGetChild(0).getChild(0) instanceof ASTEmptyStatement));

    }

}
```

Figura 2.6: Implementazione della regola ForNotEmpty

L'implementazione della regola si chiude scrivendo l'apposita *rule* nel file ruleset come mostrato in precedenza.

2.2 Eclipse PMD

Sul MarketPlace di Eclipse[8] esiste un plugin per eclipse che integra PMD all'interno dell'IDE di Eclipse e che permette la visualizzazione dell'output di PMD come se i messaggi di individuazione di difetti fossero dei warning di compilazione.

La configurazione del plugin è molto semplice e richiede la sola installazione dal marketplace di Eclipse di *Eclipse-PMD*, e la creazione di un file ruleset.xml.

Una volta creato un nuovo progetto Java, per abilitare PMD per quel progetto, basta cliccare con il tasto destro del mouse sull'icona del progetto in Project Explorer, andare in *Properties*, selezionare *Java* dal menu a sinistra, e cliccare su PMD per accedere alla finestra di configurazione di PMD. L'unica cosa che resta da fare, è quella di richiamare il path del file ruleset.xml che si vuole usare nel progetto, così come descritto nei paragrafi precedenti.

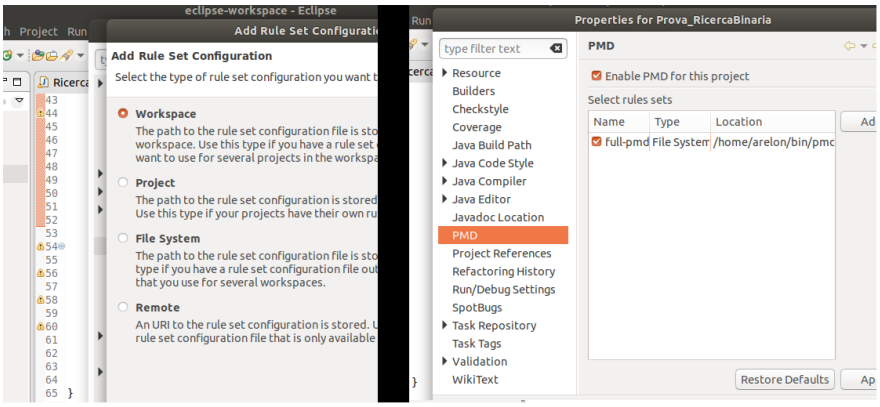


Figura 2.7: Configurazione di Eclipse - PMD

2.3 Esempio

A titolo di esempio, si riporta un file sorgente di prova per testare Eclipse-PMD (Figura 2.8).

[8] In questo lavoro è stato usata la release di Eclipse *Oxygen*

La Figura 2.9 riporta un esempio di come Eclipse-PMD mostri i difetti identificati da PMD.

Infine in Figura 2.10 si riporta una possibile soluzione ai problemi segnalati da PMD.

```
1    package org.unina.moscato.algoritmi;
2
3    public class RicercaBinaria {
4
5
6
7       private static int ricercaBinaria (int vett[], int sin, int dex, int
elem)
8       {
9               int retval=-1;
10
11              if (sin>dex || sin<0 || dex<0 || dex>vett.length)
12
13                      return retval;
14
15              else if(elem==vett[sin]) {
16
17                      retval=sin;
18
19              }
20
21             else if(elem==vett[dex]) {
22
23                      retval=dex;
24
25              }
26
27             else {
28
29                int medio=(dex+sin)/2;
30
31               if(elem==vett[medio]) {
32                      retval=medio;
33
34               }
35               else if(elem>vett[medio]) {
36                      retval = RicercaBinaria.ricercaBinaria(vett,medio+1, dex,
elem);
37
38               }
39               else if(elem<vett[medio]) {
40                      retval = RicercaBinaria.ricercaBinaria(vett,sin, medio-1,
elem);
41
42               }
43               else
44                      retval = -1;
45
46             }
47               return retval;
48
49
50
51
52      }
53
54      public static int cerca(int vettore[], int elemento) {
55
56              int len = vettore.length;
57
58              int retval = RicercaBinaria.ricercaBinaria(vettore, 0, len-1,
elemento);
59
60              return retval;
61
62      }
63
64
65   }
```

Figura 2.8: Sorgente con Difetti

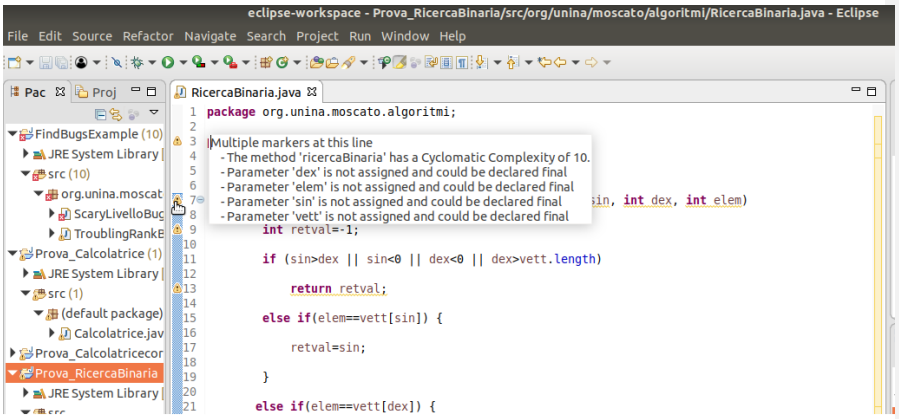


Figura 2.9: Segnalazione dei difetti in Eclipse PMD

Di seguito si riporta una tabella riassuntiva dei difetti identificati da PMD:

Linea di Codice	Tipo di Errore	Messaggio di Errore
3	*Best Practice*	-All method are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
3	*Design*	-The class “Ricerca Binaria” has a cyclomatic complexity of 6 (Highest =10)
3	*Documentation*	-headercommentrequired Required.
7	*Design*	-The method “ricercaBinaria” has a Cyclomatic Complexity of 10.
7	*Best Practice*	-Parameter “dex” is not assigned and could be declared final.
7	*Best Practice*	-Parameter “elem” is not assigned and could be declared final.

7	*Best Practice*	-Parameter “sin” is not assigned and could be declared final.
7	*Best Practice*	-Parameter “vett” is not assigned and could be declared final.
9	*Error Prone*	-Found “DD” anomaly for variable ‘retval’ (lines ‘9’-‘17’).
13	*Error Prone*	-Avoid using if...else statement without using curly braces.
13	*Design*	-A method should have only one exit point, and that should be the last statement in the method.
29	*Best Practice*	-Local variable ‘medio’ could be declared final.
44	*Error Prone*	-Avoid using if...else statement without using curly braces.
54	*Best Practice*	-Parameter “elemento” is not assigned and could be declared final.
54	*Best Practice*	-Parameter “vettore” is not assigned and could be declared final.
54	*Documentation*	-publicmethodcommentrequirement Required
60	*Design*	-Consider simply returning the value vs storing it in local variable ‘retval’.

```
 1 package org.unina.moscato.algoritmi;
 2
 3 /**
 4 * RicercaBinaria
 5 * Utility Class per RicercaBinaria su vettore di interi.
 6 *
 7 * @author  Claudio Moscato
 8 * @version 0.1
 9 * @since   2017-10-31
10 */
11
12
13 public final class RicercaBinaria {
14
15         private  RicercaBinaria() {
16                 // Per implementare una Utility Class
17         }
18
19         private static boolean controlloerrori (final int sin, final int dex,
```

```
final int leng) {
    20
    21                  return sin>dex || sin<0 || dex<0 || dex>leng ?true:false;
    22          }
    23
    24
    25          private static int ricercaBinaria (final int vett[], final int sin, final
int dex, final int elem)
    26          {
    27                  int retval;
    28
    29                  if (RicercaBinaria.controlloerrori(sin,dex,vett.length)) {
    30
    31                          retval=-1;
    32                  }
    33
    34                  else if(elem==vett[sin]) {
    35
    36                          retval=sin;
    37
    38                  }
    39
    40          else if(elem==vett[dex]) {
    41
    42                          retval=dex;
    43
    44                  }
    45
    46          else {
    47
    48           final int medio=(dex+sin)/2;
    49
    50           if(elem==vett[medio]) {
    51                   retval=medio;
    52
    53           }
    54           else if(elem>vett[medio]) {
    55                   retval = RicercaBinaria.ricercaBinaria(vett,medio+1, dex, elem);
    56
    57           }
    58           else if(elem<vett[medio]) {
    59                   retval = RicercaBinaria.ricercaBinaria(vett,sin, medio-1, elem);
    60
    61           }
    62           else {
    63                   retval = -1;
    64           }
    65          }
    66                   return retval;
    67
    68
    69
    70
    71          }
    72
    73
    74          /**
    75           * Cerca un elemento all'interno di un vettore ordinato.
    76           * ritorna l'indice dell'elemento se trovato -1 altrimenti
    77           *
    78           * @param  vettore array ORDINATO su cui effettuare la ricerca
    79           * @param  elemento l'elemento da cercare
    80           * @return      l'indice dell'elemento se trovato; -1 altrimenti
    81           */
    82
    83          public static int cerca(final int vettore[],final int elemento) {
    84
    85                  final int len = vettore.length;
    86
    87                  return RicercaBinaria.ricercaBinaria(vettore, 0, len-1, elemento);
    88
    89          }
    90
    91
    92  }
```

Figura 2.10: Soluzione ai Difetti segnalati da PMD

Capitolo 3: CheckStyle

CheckStyle [3.1] è uno strumento di analisi statica del codice ideato per la verifica della conformità del codice sorgente a svariati standard di codifica. Sebbene sembri sufficiente, per queste operazioni di verifica, un semplice matching di pattern di stringhe, CheckStyle implementa la verifica come un vero e proprio sistema di analisi statica[3.2], risultando uno strumento altamente affidabile, scalabile, facilmente adattabile a diverse esigenze e espandibile per la scrittura di nuove regole di verifica.

Analogamente a PMD, CheckStyle realizza il controllo di conformità con le regole di codifica desiderate analizzando l'*Abstract Syntax Tree.*

Anche CheckStyle viene distribuito come strumento a linea di comando, in licenza Open Source GNU LGPL-2.1+. Esistono poi diversi progetti per l'integrazione di CheckStyle nei vari strumenti IDE (approfondiremo in un paragrafo successivo le caratteristiche del Plugin di CheckStyle per Eclipse).

A discapito del nome, CheckStyle non viene utilizzato solo per stabilire la conformità di un codice sorgente con un particolare stile di codifica. La caratteristica di CheckStyle di essere uno strumento di analisi statica che verifica regole su un AST, permette di effettuare una ampia gamma di controlli sul codice. CheckStyle è inoltre facilmente estendibile in maniera molto simile a PMD tramite la scrittura di apposite classi Java.

I componenti principali di CheckStyle che permettono di implementare regole ed altri elementi per l'analisi statica del codice sono:

- i *Check*: un *Check* corrisponde ad una regola per il controllo di una certa caratteristica del codice sorgente (che sia sintomo di un difetto o che violi uno standard di codifica). Come verrà descritto in seguito, ad ogni Check corrisponde una classe Java che implementa un controllo sull'AST generato da un file Java.

- i *Moduli*: costituiscono lo strumento di configurazione di CheckStyle per l'attivazione di determinati Check, per la definizione di proprietà per i checker e, tra l'altro. per la definizione di filtri per l'output di CheckStyle (includendo formattazione degli avvisi, mascheramento dei check, definizione dei livelli di "logging" dello strumento etc.)
- i *Listner*: Monitorano il progresso di un modulo (detto *Checker*) attivo su un Check. Provvedono quindi ad effettuare il logging delle attività dei Checker durante la loro esecuzione.

3.1 I Check in CheckStyle

Come definito precedentemente, un Check corrisponde alla verifica di una proprietà sull'AST del codice sorgente di un file Java. I Check di default sono divisi in quattordici categorie:

- *Annotation:* Contiene tutti i Check sulle annotazioni del codice Java, come ad esempio il controllo sullo stile di annotazione, la presenza di annotazioni su cdice deprecato etc.
- *Block Check*: comprende i Check che controllano la presenza di Blocchi dopo gli appropriati costrutti (ad esempio dopo i for), la presenza delle parentesi graffe per l'identificazione dei blocchi etc.
- *Class Design*: i Check di questa categoria controllano determinate caratteristiche della struttura delle Classi Java, per valutare la bontà del progetto del software. Ad esempio, uno dei Check in questa categoria comprende la verifica che le Interfacce siano utilizzate solo per la definizione di Nuovi Tipi, che le Inner Class siano dichiarate solo alla fine delle classi che lo contengono etc.
- *Coding*: In questa categoria compaiono tutti i Check sulle buone norme di codifica (presenza opportuna di costruttori, Catch e Throws corretti, Switch senza default etc.)
- *Headers*: contiene i Checker per la verifica di correttezza delle intestazioni per classi

e metodi, nonché tutte le proprietà che permettono di ignorare un certo numero di linee (di commenti) nelle intestazioni delle classi

- *Imports*: Questi Checker controllano la validità delle operazioni di import, controllando ad esempio, se siano presenti import con wildcard (*); che non ci siano ridondanze o import non utilizzati etc.
- *JavaDoc Comments*: questa categoria contiene tutte i check per il controllo di un'opportuna presenza di commenti JavaDoc all'interno del codice.
- *Metrics*: qui vengono riportati i Check che implementa valutatori di metriche di complessità del software come: numero ciclomatico, Java NCSS, Complessità NPath; Complessità di Accoppiamento dei Dati, Fan Out delle Classi etc.
- *Miscellaneous*: questa categoria contiene i Checker che non sono stati inseriti nelle altre categorie
- *Modifiers*: questi checkers controllano la correttezza dei modificatori private, public, static, final, volatile, transient etc.
- *Naming Conventions*:i checkers in questa categoria controllano i nomi delle classi astratte, di quelle ereditate, delle variabili di conteggio dei cicli etc.
- *Regexp*:qui vengono controllati i blocchi con espressioni regolari per evitare classici pattern di errori nella loro definizione
- *Size Violation*: Questi checker controllano dimensioni di stringhe, buffers, files etc. Sono configurabili in funzione della proprietà che determina la dimensione massima.
- *WhiteSpaces*: Questi Checker controllano la presenza/assenza di "whitespaces" nei punti opportuni, come ad esempio assenza di linee vuote, presenza di uno spazio tra le parentesi tonde e graffe etc.

3.1.1 Definire un nuovo Check in CheckStyle

Come in PMD, per permettere la scrittura di un nuovo Check, anche in CheckStyle è utile

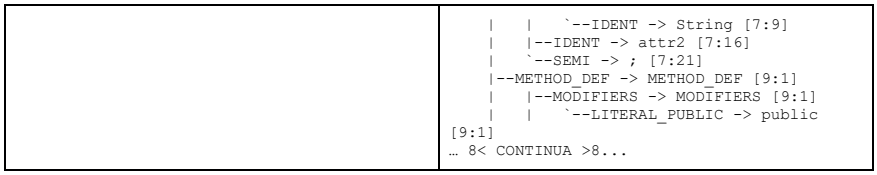
avere uno strumento per la stampa dell'AST generato dal tool per l'analisi del codice sorgente. La stampa dell'AST aiuta a comprendere come visitarlo per implementare il nuovo Check[9].

Se ad esempio si volesse scrivere un Checker per implementare un controllo che segnali la presenza di attributi di una classe superiore a un numero massimo fissato, si dovrebbe procedere nel seguente modo.

Per prima cosa conviene controllare l'AST di una classe che useremo come esempio. In Figura 3.1 viene riportata la classe di prova (MiaClass) sulla sinistra, mentre sulla destra viene riportato la parte dell'AST associato alla classe che riporta la struttura degli attributi (il resto viene omesso per brevità). In grassetto sono evidenziati gli elementi di interesse: in pratica, una volta identificato il primo elemento **OBJBLOCK** nell'AST, bisogna considerarne i figli con tag **VARIABLE_DEF** e contarli (nel nostro esempio non faremo altri tipi di controlli come quelli sull'identificatore di visibilità etc.)

Classe di Prova MiaClass	AST
/** * Mia class. * @see AbstractClass */ public class MiaClass { private int attr1; private String attr2; public int method1(int par1){ int a= par1+1; return a; } public float method2 (int par2){ float b = par2 /2; return b; } }	CLASS_DEF -> CLASS_DEF [5:0] |--MODIFIERS -> MODIFIERS [5:0] | `--LITERAL_PUBLIC -> public [5:0] |--LITERAL_CLASS -> class [5:7] |--IDENT -> MiaClass [5:13] `--**OBJBLOCK** -> OBJBLOCK [5:22] |--LCURLY -> { [5:22] |--**VARIABLE_DEF** -> VARIABLE_DEF [6:1] | |--MODIFIERS -> MODIFIERS [6:1] | | `--LITERAL_PRIVATE -> private [6:1]   | |--TYPE -> TYPE [6:9]   | | `--LITERAL_INT -> int [6:9] | |--IDENT -> attr1 [6:13] | `--SEMI -> ; [6:18]   |--**VARIABLE_DEF** -> VARIABLE_DEF [7:1]   | |--MODIFIERS -> MODIFIERS [7:1]   | | `--LITERAL_PRIVATE -> private [7:1] | |--TYPE -> TYPE [7:9]

[9]Checkstyle fornisce uno strumento a riga di comando ed uno grafico per la visualizzazione dell'AST relativo ad un file java:
(da linea di comando) : java -jar checkstyle-X.XX-all.jar -t <Classe>.javaoppure: java -jar checkstyle-X.XX-all.jar -T <Classe>.java nel caso si vogliano stampare anche le informazioni su letterali e commenti
(strumento grafico): java -cp checkstyle--X.XX-all.jar com.puppycrawl.tools.checkstyle.gui.Main



```
    |   |   `--IDENT -> String [7:9]
    |   |--IDENT -> attr2 [7:16]
    |   `--SEMI -> ; [7:21]
    |--METHOD_DEF -> METHOD_DEF [9:1]
    |   |--MODIFIERS -> MODIFIERS [9:1]
    |   |   `--LITERAL_PUBLIC -> public
[9:1]
… 8< CONTINUA >8...
```

Figura 3.1: Classe di esempio e AST

Per scorrere l'AST, CheckStyle mette a disposizione una classe astratta *Visitor*, che bisogna estendere per realizzare il checker, così come riportato in Figura 3.2.

```
package it.unina.moscato.checkstyle.checks;
import com.puppycrawl.tools.checkstyle.api.*;

public class VarsLimitCheck extends AbstractCheck
{
    private static final int DEFAULT_MAX = 30;
    private int max = DEFAULT_MAX;

    @Override
    public int[] getDefaultTokens()
    {
        return new int[]{TokenTypes.CLASS_DEF, TokenTypes.INTERFACE_DEF};
    }

    @Override
    public void visitToken(DetailAST ast)
    {
        // cerca il primo nodo  OBJBLOCK al di sotto di un nodo CLASS_DEF/INTERFACE_DEF
        DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);

        // conta il numero di figli di OBJBLOCK
        // che hanno il tag  VARIABLE_DEFS
        int varsDefs = objBlock.getChildCount(TokenTypes.VARIABLE_DEF);

        // Stampa un messaggio se il numero di variabili è maggiore di quello consentito
        if (varsDefs > this.max) {
            String message = "TROPPI ATTRIBUTI: sono consentiti solo " + this.max + "
variabili";
            log(ast.getLineNo(), message);
        }
   }
            //configurato tramite le "Property" nel file XML del Check
      public void setMax(int limit)
    {
        max = limit;
    }
}
```

Figura 3.2 : Implementazione del Checker

Dopo aver compilato il checker, bisogna integrarlo in Checker,, bisogna integrarlo in CheckStyle. Le funzionalità di checkstyle, sono integrate in Moduli che possono a loro volta funzionare come contenitori di altri moduli. Il modulo radice di CheckStyle implementa *l'interfaccia FileSetCheck*, che viene utilizzata per leggere dei file in input e per stampare

dei messaggi in output. Una delle implementazioni di default in CheckStyle di FileSetCheck è la classe *TreeWalker* che viene utilizzata per realizzare moduli con Visitor di AST e che supporta qualsiasi modulo abbia ereditato la classe astratta *AbstractCheck* (così come è avvenuto per il nostro checker in Figura 3.2).

A questo punto, quindi, bisogna:

1. Compilare il checker e inserire i .class compilati in un jar, ad esempio *unina.jar*
2. Scrivere un file di configurazione per il nuovo modulo (config.xml) che appaia come in Figura 3.3. Si noti come venga definito l'intero nome del package (in grassetto) e di come la definizione del tag *property* permetta di settare la variabile *max* del checker.
3. Bisogna infine avere cura di lanciare CheckStyle con gli opportuni parametri, includendo nel classpath anche il jar creato al punto 1[10]

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
    "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
    "http://checkstyle.sourceforge.net/dtds/configuration_1_3.dtd">
<module name="Checker">
  <module name="TreeWalker">
         <!-- Moduli standard da utilizzare -->
    <module name="MethodLength"/>
         <!-- Modulo implementato dal Checker  -->
    <module name="it.unina.moscato.checkstyle.checks.VarsLimitCheck">
      <property name="max" value="1"/>
    </module>
  </module>
</module>
```

Figura 3.3: config.xml per il nuovo Checker

3.2 CheckStyle Eclipse plugin

Il plugin per Eclipse che integra Checkstyle all'interno dell'IDE, ha una interfaccia del tutto simile a quella di PMD. Il Plugin è disponibile sul Marketplace di Eclipse. Esattamente per come accadeva per il plugin di PMD, le linee in cui vengono identificati i difetti appaiono

[10] checkstyle verrà chiamato con parametri simili ai seguenti:
`java -classpath unina.jar:checkstyle-X.XX-all.jar com.puppycrawl.tools.checkstyle.Main -c config.xml <Progetto>`

contrassegnati da un'icona a forma di lente di ingrandimento e, posizionando il puntatore del mouse sull'icona, il log del checker che ha identificato il difetto appare sotto forma di tooltip.

Questa volta, pero', la configurazione di CheckStyle avviene per l'intero IDE, e non per singolo progetto. Per avviare la configurazione, bisogna accedere al menu:

Window->Preferences-> CheckStyle

Comparirà il menu di configurazione di CheckStyle, dove è possibile definire quali Checker abilitare o meno per un determinato stile di codifica, così come mostrato in Figura 3.4.

Si fa notare che il plugin di Eclipse ha già preparati moduli completi per gli stili di codifica standardizzati da Google e da SUN Enterprise.

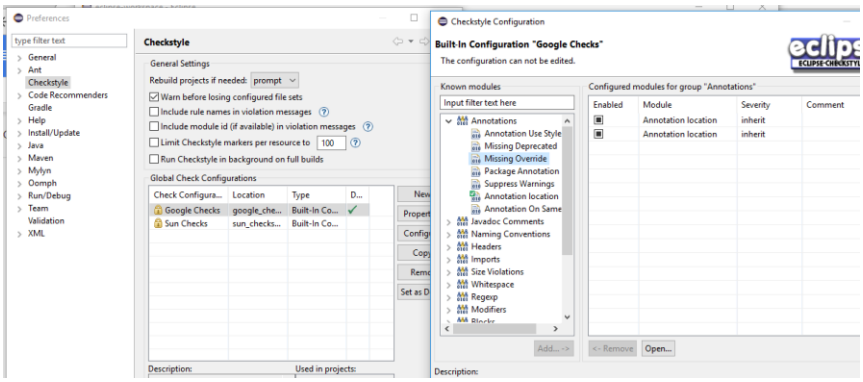


Figura 3.4: Configurazione di CheckStyle

Per poter poi utilizzare CheckStyle nei progetti, basterà creare un nuovo progetto e questa volta, selezionando il progetto dal Project Explorer e cliccando con il tasto destro del mouse, dal menù comparirà direttamente una voce *CheckStyle* che eseguirà i Check definiti in fase di configurazione, così come mostrato nell'esempio successivo.

3.3 Esempio

Analogamente per quanto fatto per PMD, riportiamo un esempio di segnalazione di difetti da parte di CheckStyle. Il sorgente utilizzato in questo esempio è lo stesso di quello dell'esempio del paragrafo 2.3 e riportato in Figura 2.8. Nell'esempio, CheckStyle è stato configurato per gestire le regole per controllare la conformità allo stile di codifica di Google. La Figura 3.5 mostra come appare la notifica delle segnalazioni di CheckStyle nel plugin per eclipse, per il file di esempio.

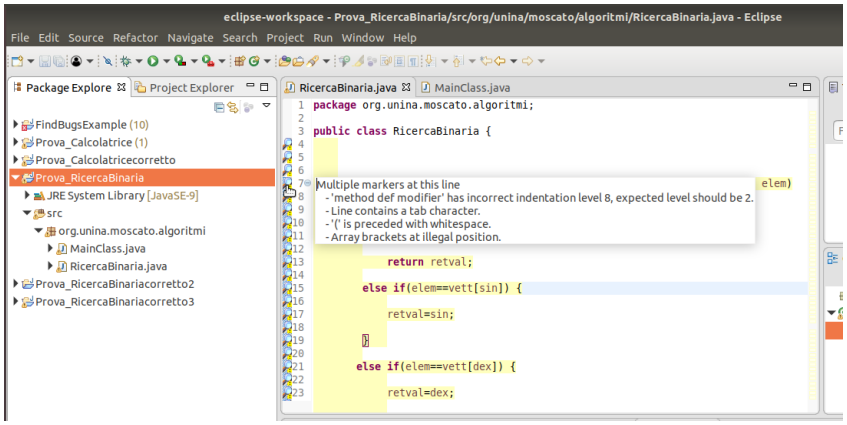


Figura 3.5: Segnalamento dei difetti in Eclipse CheckStyle

Messaggi di Errore di Checkstyle.

Linea di Codice	Messaggio di Errore
Ogni Linea	Line contains a tab character.
7	-'method def modifier' has incorrect indentation level 8, expected level should be 2. -'(' is preceded with whitespace -Array brackets at illegal position
9	-'method def' has incorrect indentation level 16, expected level should

	be 14. -Whitespacearound: '=' is not preceded with whitespace. -Whitespacearound: '=' is not followed by whitespace. Empty block may only be represented as {} when not part of a multi-block statement (4.1.3).
29	-Whitespacearound: '+' is not preceded with whitespace. -Whitespacearound: '+' is not followed by whitespace. Empty block may only be represented as {} when not part of a multi-block statement (4.1.3). -Whitespacearound: '/' is not preceded with whitespace. -Whitespacearound: '/' is not followed by whitespace. Empty block may only be represented as {} when not part of a multi-block statement (4.1.3). -'else' child has incorrect indentation level 10, expected level should be 6. -Whitespacearound: '=' is not preceded with whitespace. -Whitespacearound: '=' is not followed by whitespace. Empty block may only be represented as {} when not part of a multi-block statement (4.1.3).

In figura 3.6, si riporta una possibile soluzione ai difetti segnalati da CheckStyle.

```
 1 package org.unina.moscato.algoritmi;
 2
 3 public class RicercaBinaria {
 4
 5
 6
 7   private static int ricercaBinaria(int[] vett, int sin, int dex, int elem) {
 8     int retval = -1;
 9
10     if (sin > dex || sin < 0 || dex < 0 || dex > vett.length) {
11
12       return retval;
13     } else if (elem == vett[sin]) {
14
15       retval = sin;
16
17     } else if (elem == vett[dex]) {
18
19       retval = dex;
20
21     } else {
22
23       int medio = (dex + sin) / 2;
24
25       if (elem == vett[medio]) {
26         retval = medio;
27
28       } else if (elem > vett[medio]) {
29         retval = RicercaBinaria.ricercaBinaria(vett,medio + 1, dex, elem);
30
31       } else if (elem < vett[medio]) {
32         retval = RicercaBinaria.ricercaBinaria(vett,sin, medio - 1, elem);
33
34       } else {
35         retval = -1;
36       }
37     }
38     return retval;
```

```
39
40    }
41
42    public static int cerca(int[] vettore, int elemento) {
43
44      int len = vettore.length;
45
46      int retval = RicercaBinaria.ricercaBinaria(vettore, 0, len - 1, elemento);
47
48      return retval;
49
50    }
51
52
53  }
```

Figura 3.6: Soluzione alle segnalazioni di CheckStyle

Capitolo 4: SpotBugs

SpotBugs [4.1] è un progetto che eredita l'ormai abbandonato FindBugs[4.2], celebre analizzatore statico sviluppato dall'Università del Maryland. SpotBugs ha aggiunto dal 2015 (anno dell'ultimo rilascio stabile di FindBugs) più di 100 regole di individuazione di possibili errori del codice sorgente. Attualmente SpotBugs supporta l'individuazione di circa 400 tipi di difetti del software.

Essendo un prodotto accademico, FindBugs è stato oggetto e strumento di numerosi studi e ricerche [4.3], e SpotBugs ne eredita tutte le caratteristiche.

SpotBugs è un analizzatore statico che lavora tramite *visite su un Abstract Syntax Tree* ma, a differenza di PMD e CheckStyle, non genera l'AST a partire dal codice sorgente dei file Java, bensì *costruisce un AST a partire dal* ***bytecode*** dei programmi java da analizzare.

SpotBugs viene rilasciato sotto licenza GNU LGPL.

I componenti fondamentali di SpotBugs, sono i *Detectors* che realizzano l'individuazione dei difetti. Come per PMD e CheckStyle, i Detectors implementano una visita sull'AST del bytecode del codice sorgente da analizzare. Alcuni detectors effettuano operazioni più complicate, come Data Flow Analysis, o persino analisi tramite motori inferenziali.

SpotBugs è uno strumento utilizzato a riga di comando, e direttamente integrato nei processi di compilazione del codice tramite ANT o MAVEN. Ne esiste comunque una interfaccia grafica integrata in Eclipse, che verrà descritta più avanti. Inoltre, SpotBugs mette a disposizione una serie di API proprio per lo sviluppo di plugin da inserire negli strumenti di sviluppo integrato.

Analogamente a CheckStyle, è possibile definire, in fase di configurazione, una serie di *Filtri* che permettono di attivare, disattivare e configurare i Detector che l'analista vuole utilizzare in un dato progetto.

Uno dei punti di forza di SpotBugs è l'analisi di difetti per codice multithreaded, non presente negli strumenti precedentemente utilizzati, oltre che ad una serie di Detector specializzati per l'individuazione di falle di sicurezza nel codice.

4.1 I Detector in SpotBugs

SpotBugs classifica i suoi Detectors in dieci categorie:

- *Bad Practice*: include i difetti dovuti a cattive pratiche di programmazione, come ad esempio: l'invocazione di metodi Swing al di fuori di Thread Swing; Finalizzatori vuoti; Classi Clonable senza il metodo clone(); Campi immutabili che non sono final etc.
- *Correctness* : includono difetti di implementazioni scorrette, la presenza di loop infiniti; Metodi che non controllano che i parametri siano non null etc.
- *Experimental* : In questa categoria vengono inclusi tutti i detector sperimentali
- *Internationalization*: include tutti la gestione dei difetti dovuti a problemi di internazionalizzazione delle codifiche
- *Malicious Code vulnerability*: Questa categoria include un enorme numero di detectors per l'individuazione di diverse vulnerabilità del codice che possono dare adito a utilizzo malizioso dei programmi: detectors che controllino che vengano utilizzati i giusti quantificatori per variabili e metodo, che vengano effettivamente gestiti oggetti immutabili etc.
- *Multithreaded correctness*: Questa categoria include la gestione dei difetti dovuti ad una scorretta implementazione di codice multi-threaded, in particolar modo riguardo alle classi che non implementano opportuni monitor (synchronized) per l'eventuale gestione condivisa delle risorse.
- *Bogus Random Noise*: i detectors di questa categoria gestiscono il management non deterministico di metodi, campi etc.

- *Performance*: In questa categoria rientrano tutti i difetti che ricadono in degradazioni di performance, come la gestione non ottima di buffers di grandi dimensioni, l'utilizzo inopportuno dei tipi wrapper etc.
- *Security*: Questa categoria racchiude i detector che analizzano falle di sicurezza nel software, in particolare nel caso di software *server-side* e *servlet*
- *Dodgy Code*: i detectors di questa categoria scovano difetti dovuti alla scrittura di codice ingannevole (ad esempio: assegnazioni utilizzate nei blocchi condizionali di if e while).

I Detectors in SpotBugs utilizzano la libreria *Byte Code Engineering* BCEL per la creazione di AST a partire dai bytecode e per implementare opportunamente i Visitor per l'AST. BCEL mette a disposizione svariati Visitors: Tutti i rilevatori di scansione bytecode sono basati sul pattern visitor. Un Visitor molto versatile è BetterVisitor, per i quali sono disponibili decine di metodi di utilità, fra i quali sicuramente verranno ereditati il metodo di visita dell'AST (*visit*) ed il metodo per analizzare i codici operativi del bytecode (*sawOpCode)*, ovvero il metodo per analizzare i singoli nodi dell'AST.
Comunque, la scrittura di un Detector è una operazione complicata, che richiede conoscenze riguardanti la decompilazione di bytecode Java, l'architettura di una java Virtual Machine, la gestione dei thread in java etc. Si ritiene pertanto di non poter trattare direttamente questo argomento in questo lavoro di tesi.

4.2 SpotBugs Eclipse plugin

Il plugin per Eclipse che integra SpotBugs all'interno dell'IDE, ha una interfaccia del tutto simile a quella di PMD e checkstyle. Il Plugin è disponibile sul Marketplace di Eclipse. La configurazione del plugin è praticamente identica a quella degli altri due plugin trattati in

questa tesi, e quindi non la approfondiremo qui. L'unica cosa da notare in questo plugin, è la presenza di *una perspective* vera e propria che, all'atto della individuazione dei difetti, apre una casella di testo in una barra laterale che descrive il difetto e i possibili effetti che può avere sul codice, oltre a proporre alcuni metodi per eliminare il difetto.

Inoltre, in fase di configurazione di SpotBugs, è possibile definire il comportamento dell'IDE nel caso in cui venga identificato un difetto: E' possibile fare in modo che, a seconda del livello di *gravità* del difetto (*Concern in SpotBugs*) Eclipse faccia risultare un semplice messaggio informativo (*Info*), un *Warning* o un vero e proprio errore (*Error*).

In Figura 4.1 è mostrato l'interfaccia di configurazione di SpotBugs Eclipse.

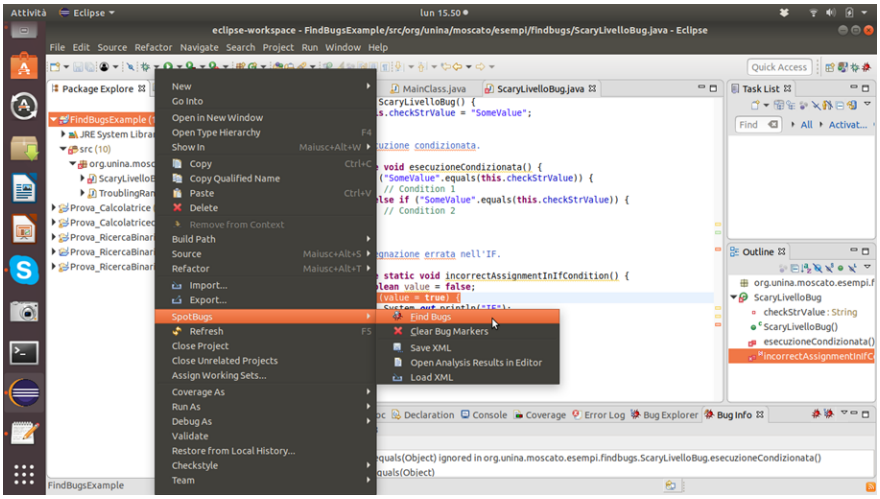


Figura 4.1: Interfaccia di Eclipse - SpotBugs

4.3 Esempio

Il codice di prova per SporBugs è riportato in Figura 4.2

```
1   package org.unina.moscato.esempi.findbugs;
2
3   public class ScaryLivelloBug {
4   /** The check str value. */
5   private String checkStrValue = null;
6   /**
7   * Costruttore.
8   */
9   public ScaryLivelloBug() {
10  this.checkStrValue = "SomeValue";
11  }
```

```
12              /**
13              * Esecuzione condizionata.
14              */
15              private void esecuzioneCondizionata() {
16              if ("SomeValue".equals(this.checkStrValue)) {
17                     // Condition 1
18              } else if ("SomeValue".equals(this.checkStrValue)) {
19                     // Condition 2
20              }
21              }
22              /**
23              * Assegnazione errata nell'IF.
24              */
25              private static void incorrectAssignmentInIfCondition() {
26              boolean value = false;
27              if (value = true) {
28                     System.out.println("IF");
29              } else {
30                     System.out.println("ELSE");
31              }
32              }
33
34               }
35
```

Figura 4.2: Codice di Prova per SpotBugs

I messaggi di Errore di SpotBugs sono riassunti nella seguente tabella

Linea di codice	Messaggio di Errore
16	Private method org.unina.moscato.esempi.findbugs.ScaryLivelloBug.esecuzioneCondizionata() is never called [Of Concern(20), Low confidence]
18	Return value of String.equals(Object) ignored in org.unina.moscato.esempi.findbugs.ScaryLivelloBug.esecuzioneCondizionata() [Scary(8), Low confidence]
26	Private method org.unina.moscato.esempi.findbugs.ScaryLivelloBug.incorrectAssignmentInIfCondition() is never called [Of Concern(20), Low confidence]
27	Dead store to value in org.unina.moscato.esempi.findbugs.ScaryLivelloBug.incorrectAssignmentInIfCondition() [Of Concern(17), Normal confidence]
27	IncorrectAssignmentInIfCondition() assigns boolean literal in boolean expression [Scary(5), High confidence]

L’editor e la perspective FindBugs appaiono come in Figure 4.3 e 4.4

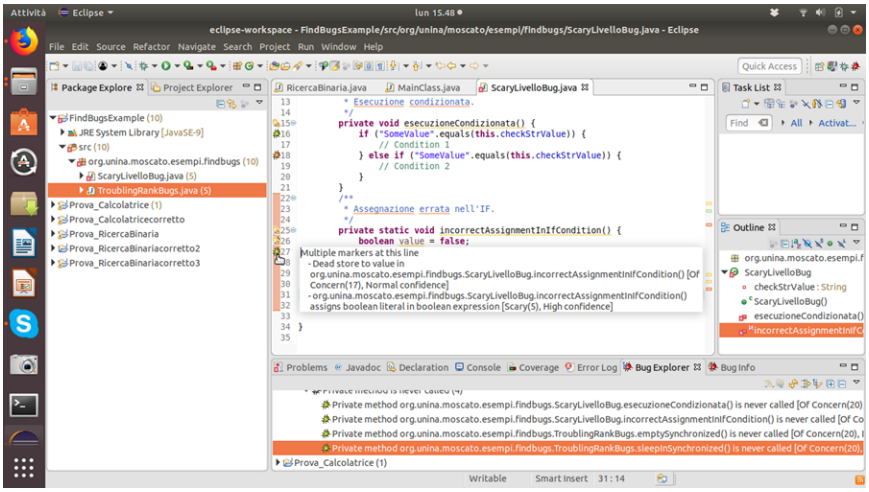


Figura 4.3: Errori in SpotBugs

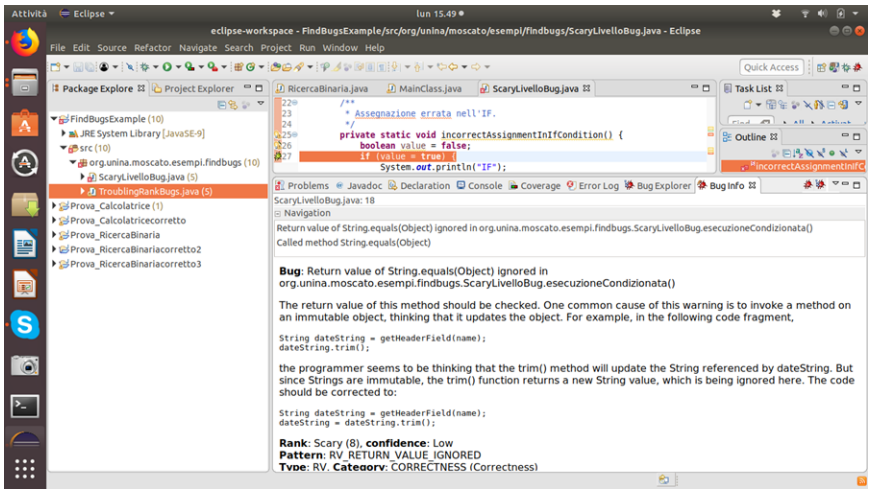


Figura 4.4:Perspective di Eclipse SpotBugs

Capitolo 5: Confronto degli Strumenti Analizzati

I tre strumenti analizzati, come risulta sicuramente chiaro, hanno molti punti in comune. Nelle prime versioni degli analizzatori, le differenze tra i diversi tools erano più marcate, sia per quanto riguarda le metodologie di analisi che i difetti evidenziabili. Ad esempio, ne' PMD ne' CheckStyle implementavano analisi data-flow, mentre già le prime versioni di FindBugs (il predecessore di SpotBugs) usava l'analisi data-flow per implementare alcuni dei suoi detector. Attualmente, tutti e tre gli strumenti hanno almeno un componente che utilizza analisi data-flow per l'individuazione di alcuni difetti. Ancora, CheckStyle non implementava alcun controllo di "corretto Design" delle Classi, cosa che invece nelle ultimi versioni fa.

Una prima differenza si può trovare nei *linguaggi supportati* dagli analizzatori. Mentre CheckStyle e SpotBugs supportano praticamente solo Java, PMD è uno strumento multi-linguaggio. Al momento supporta 8 linguaggi di programmazione (la maggior parte Java-Based), ma dando uno sguardo ai repository dei codici sorgenti, si può notare che sono in via di sviluppo analizzatori per molti altri linguaggi (inclusi C, C++ etc.).

Un'altra differenza tra i tre sta nel fatto che, mentre PMD e CheckStyle eseguono le analisi sul *codice sorgente Java*, SpotBugs analizza direttamente *il bytecode*. Questo porterebbe a pensare che i primi due analizzatori possono realizzare analisi anche di codice non compilabile (per qualche tipo di errore), mentre SpotBugs ha bisogno di analizzare codice spurio da errori di compilazione. In realtà sia PMD che CheckStyle, quando operano su codice che "non compila", hanno il problema di identificare numerosi falsi positivi e quindi è sempre bene usare gli analizzatori su codice che sia compilato.

Tutti e tre gli analizzatori hanno raggiunto un elevato livello di maturità. Su questo livello,

quindi, una metrica per differenziarli può' essere quella che c*onfronta le categorie di difetti e il numero di controlli effettuati* dai tools. Da questo punto di vista, SpotBugs è sicuramente lo strumento più completo, con oltre quattrocento tipi di analisi supportate, mentre PMD e CheckStyle supportano un numero sicuramente inferiore di controlli. D'altra parte, PMD e CheckStyle coprono meglio di SpotBugs i controlli sulla documentazione del codice, mentre SpotBugs è supporta un numero di controlli su codice multithreaded e relativi a problemi di sicurezza, sicuramente maggiore degli altri due strumenti. Ancora, solo CheckStyle supporta Cinque metriche per l'analisi della complessità del codice, a differenza di PMD che ne supporta solo una e di SpotBugs che, al momento, sembra non supportarne alcuna.

Un'altro tipo di differenza tra i vari strumenti, è nelle API che permettono di introdurre nuovi controlli negli analizzatori. Da questo punto di vista, PMD è lo strumento ad avere, a nostro avviso, il migliore insieme di API e il primato nella semplicità con cui si può implementare un nuovo controllo. CheckStyle è molto simile a PMD da questo punto di vista, ma ha lo svantaggio di avere un ridotto numero di classi da ereditare per implementare le visite dell'AST da analizzare. SpotBugs, per quanto riguarda l'espandibilità, è sicuramente lo strumento più complicato da estendere, non fosse altro perchè richiede la conoscenza anche delle strutture dei bytecode Java.

Conclusioni

In questo lavoro di tesi sono stati studiati tre sistemi di analisi statica del codice: PMD, CheckStyle e SpotBugs. Dei tre analizzatori è stata descritta l'architettura generale, sono state descritte alcune plug-in che ne integrano le funzionalità in uno strumento integrato di sviluppo come Eclipse.

Sono stati riportati degli esempi di utilizzo dei tre strumenti, evidenziando come vengano riportate le segnalazioni di presenza di difetti, e come sia possibile risolvere i difetti.

Essendo possibile estendere i tre analizzatori con ulteriori controlli, caso per caso sono stati descritti, con esempi, i processi con cui è possibile svilupparli ed integrarli nei tre strumenti.

Infine, è stato riportato un breve paragrafo che confronta le potenzialità e le caratteristiche dei tre strumenti.

Bibliografia

[1.1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques, and tools. Vol. 1-2.: Addison-Wesley, 2007.

[1.2] D'silva, Vijay, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27.7 (2008): 1165-1178.

[1.3] Winskel, Glynn, Franco Turini, and Ugo Montanari. La semantica formale dei linguaggi di programmazione: un'introduzione. MIT Press (IS), 1999.

[2.1] PMD Source Code Analyzer Project, https://pmd.github.io/pmd-6.0.0/index.html (Ultimo Accesso: 17/12/2017).

[2.2] JavaCC: The Java Parser Generator, https://javacc.org/ (Ultimo Accesso 28/12/2017)

[3.1] CheckStyle, http://checkstyle.sourceforge.net/ (Ultimo Accesso 02/01/2018)

[3.2] P. Louridas, "Static code analysis," in *IEEE Software*, vol. 23, no. 4, pp. 58-61, July-Aug. 2006.

[4.1] SpotBugs: https://spotbugs.github.io/ (Ultimo Accesso 03/01/2018)

[4.2] FindBugs: http://findbugs.sourceforge.net/ (Ultimo Acesso 03/01/2018)

[4.3] Foster, Jeffrey S., Michael W. Hicks, and William Pugh. "Improving software quality with static analysis." Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2007.