UNIVERSITA' 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 di analisi statica su programmi Java: SpotBugs, PMD, Checkstyle, SonarLint

Anno Accademico 2016/2017

Candidato:

Marco d'Orso

matr. N46002610

Indice

Introduzione

Alla fine di ogni stadio dello sviluppo di un software, è di norma l'utilizzo di una serie di processi di *verifica e convalida*, cioè dei procedimenti di controllo ed analisi.

In questi processi, il software viene analizzato sia verificando l'aderenza ai requisiti sia attraverso la revisione dei progetti, l'ispezione del codice ed il test del prodotto.

Possiamo dividere le tecniche di verifica e convalida in 3 tipi:

- *Analisi statica*: processo di valutazione di un sistema o di un suo componente senza che esso sia eseguito;
- *Analisi dinamica:* processo di valutazione di un software basato sull'osservazione del suo comportamento a tempo di esecuzione;
- *Analisi formale*: utilizzo di tecniche matematiche per l'analisi degli algoritmi.

1.1 Analisi statica

L'**analisi statica** è un metodo di revisione del software che consente di analizzarlo senza il bisogno di eseguirlo , svolgendo l'analisi sulla sua forma, struttura, contenuto, documentazione, comunque analizzando non soltanto il codice sorgente, ma tutte le sue possibili astrazioni.

Ci possono essere diverse tipologie di analisi statica: un primo esempio è il compilatore, che effettua una verifica del programma in modo tale che rispetti

i criteri del linguaggio e poter generare il codice oggetto, oppure l'ispezione del codice da parte di uno o più sviluppatori, che cercano di capire il codice e scovarne i difetti.

Quest'ultimo può essere un lavoro molto lungo e faticoso che, su grandi progetti, può anche non portare ai risultati sperati.

Per questo motivo sono stati sviluppati degli **strumenti automatici** di analisi statica che esulano lo sviluppatore da questo compito.

Questi strumenti possono scandire il codice sorgente del software ed individuare possibili errori ed anomalie, riconoscendo tutte le istruzioni e riuscendo a ricavare anche il flusso di controllo delle funzioni e tutti i possibili valori per i dati del programma.

I risultati ottenuti da questi tool devono, però, essere filtrati: le anomalie riscontrate non sono sempre errori, ma potrebbero essere soluzioni volute che non hanno alcun effetto negativo sull'esecuzione, e quindi rappresentare casi di *falsi positivi*.

In generale, l'analisi mediante strumenti automatici non può sostituire l'ispezione diretta del codice, perché ci sono espressioni che soltanto un operatore umano può valutare, ma gli attuali strumenti, consentendo l'implementazione di regole personalizzate dall'utente per la ricerca degli errori, si propongono per diminuire il lavoro del programmatore in questo frangente.

Questo lavoro punta ad analizzare quattro strumenti automatici di analisi statica, che sono SpotBugs, PMD, Checkstyle e SonarLint, sia dal punto di vista analitico sia in esecuzione, per valutarne i risultati e confrontarli, in modo tale da avere una visione chiara su ogni tool.

Inoltre, si analizzerà l'esecuzione su un software Java di grandi dimensioni,

quale **SpringBoot**, così da avere un riscontro dei quattro strumenti analizzando una grande quantità di codice.

SpotBugs

SpotBugs è un software gratuito ed open source che, tramite l'analisi statica di codice Java, effettua la ricerca di bug ed errori. É anch'esso un software Java che necessita quindi della *Java Virtual Machine* (JVM) per la sua esecuzione.

SpotBugs deriva da un altro strumento simile, FindBugs, il cui progetto è stato però abbandonato nel 2015.

Questo software è disponibile sia nella versione autonoma (con interfaccia grafica o da riga di comando) sia sotto forma di plugin per *Eclipse*, *Maven* e *Gradle*, e come task integrato per le build in *Ant*. Inoltre è estendibile tramite plugin per allargare il campo di ricerca.

2.1 Funzionamento

L'esecuzione richiede la compilazione del codice sorgente perché analizza direttamente il bytecode del programma: quindi riesce ad accedere direttamente a tutte le classi del software analizzato.

SpotBugs permette di inserire dei *Filter Files* in formato XML per includere o escludere dall'analisi classi, metodi, variabili, ma anche bug che scendono sotto una determinata soglia di priorità.

Inoltre, si possono regolare la precisione e l'accuratezza della ricerca tramite i flag delle *Analysis Properties* e il costo computazionale dell'analisi modificando i flag di *Effort*, avendo però un'analisi ovviamente meno approfondita e precisa.

SpotBugs esegue la ricerca di errori tramite ”bug pattern”, cioè il confronto di parti di codice con esempi difettosi che il programma ha nella propria lista di pattern.

La ricerca avviene tramite moduli *Detector*, i quali possono ereditare da 3 diverse classi a seconda del suo scopo:

1. *AnnotationDetector*: detector che analizza le annotations su classi, metodi, parametri dei moduli;
2. *BytecodeScanningDetector*: detector che analizza il bytecode nei file delle classi;
3. *OpcodeStackDetector*: sottoclasse di BytecodeScanningDetector, scansiona il bytecode usando lo stack degli operandi.

I Detector definiscono al loro interno i moduli:

- *BadCase*: modulo che definisce un esempio di bug;
- *GoodCase*: modulo che definisce un esempio di codice che non rappresenta un bug, utilizzato per evitare i casi di falsi positivi.

Una volta ottenuto il bytecode, vengono chiamati tutti i Detector inclusi nel file ”*findbugs.xml*” e, quando riscontra un bug, restituisce la sua descrizione.

Dal file ”*messages.xml*” vengono prelevate le definizioni più dettagliate del bug pattern riscontrato.

Per definire i propri bug pattern bisogna, quindi, implementare un Detector dalle classi sopra elencate, definire i propri BadCase e GoodCase, ed infine inserire il Detector nel file findbugs.xml con la descrizione del bug.

2.2 Bug Pattern

Nella sua versione base, SpotBugs possiede 10 categorie di bug differenti:

Bad practice : violazioni di pratiche essenziali nella scrittura del codice, come ad esempio l'uso di codice clonato, la cattiva gestione di eccezioni e il ritorno di una funzione con valore '*null*';

Correctness : possibile errore da parte dello sviluppatore che ha scritto codice che non avrà il risultato aspettato, ad esempio la mancata inizializzazione degli attributi senza valori di default in una classe, un apparente ciclo infinito oppure una funzione che ritorna sempre lo stesso valore;

Experimental : bug patterns che non sono stati ancora validati dalla community di SpotBugs;

Internationalization : errori che riguardano l'internazionalizzazione e la località, come ad esempio la supposizione di avere a disposizione la codifica di piattaforme di default;

Malicious code vulnerability : codice che è vulnerabile ad attacchi da parte di programmi non verificati, come la mancata dichiarazione di una funzione come '*final*' o una gestione errata dell'accesso a variabili o metodi interni di una classe;

Multithreaded correctness : bug riferiti alla gestione della concorrenza, quindi ad un uso errato di thread, lock e oggetti dichiarati '*volatile*';

Bogus random noise : pattern che mirano a migliorare esperimenti di data mining piuttosto che normali applicativi Java;

Performance : codice che non è necessariamente errato, ma che potrebbe essere inefficiente, come la dichiarazione di variabili non utilizzate, inutili chiamate a funzioni all'interno di cicli oppure la chiamata esplicita al Garbage Collector;

Security : utilizzo di input non attendibili che potrebbe creare una vulnerabilità utilizzabile da remoto, come l'accettazione di pacchetti dalla rete che non sono verificati;

Dodgy code : codice scritto in maniera confusa, anomala o poco chiara, che lo rendono poco leggibile e più soggetto ad errori, come l'uso di cast impliciti, molteplici controlli di valori '*null*' oppure la cattiva gestione di espressioni '*switch*'.

PMD

PMD è uno strumento open source, scritto in Java, che consente l'analisi statica del codice sorgente. É applicabile a codice sorgente scritto in Java, JavaScript, Salesforce.com Apex e Visualforce, PLSQL, Apache Velocity, XML, XSL.

É possibile utilizzarlo sia come applicativo autonomo da linea di comando che come plugin per vari IDE, come Eclipse e IntelliJ.

Essendo open source, permette di implementare nuove regole per la ricerca di bug sia per un linguaggio già supportato sia per nuovi linguaggi aggiunti dall'utente.

3.1 Funzionamento

PMD si basa sull'analisi dell'*Abstract Syntax Tree* (AST), cioè l'albero che rappresenta la struttura sintattica astratta del codice sorgente.

L'indagine viene sviluppata nel modo seguente:

- PMD riceve il nome del file da analizzare;
- passa il file al JavaCC che svolge la funzione di generatore di parser, il quale forma e restituisce un riferimento all'Abstract Syntax Tree;
- viene passato l'AST alla tabella dei simboli, che definisce gli scopes e trova le definizioni;

- qualora esistessero bug rules che richiedono una *Data Flow Analysis* (DFA), l'ADT viene passato al livello del DFA dove vengono definiti i grafi dei flussi di controllo e i nodi del flusso dei dati;
- ogni bug rule attraversa tutto l'AST per rilevare problemi e, eventualmente, richiamando la tabella dei simboli ed il livello del DFA se richiesto;
- infine viene generato il report con tutti i bug trovati.

3.2 Java Rulesets

PMD utilizza una serie di ruleset, cioè un insieme di esempi di codice che vengono confrontati per identificare i bug.

I ruleset cambiano a seconda del linguaggio utilizzato, e per Java le categorie principali sono le seguenti:

Basic : insieme di regole relative a buone pratiche di programmazione da seguire, come l'utilizzo di un ciclo "*for*" invece di un ciclo "*while*" oppure, la semplificazione delle condizioni negli "*if*";

Code Size : insieme di regole sui problemi relativi alla grandezza del codice e alla sua complessità. Di rilievo, tra le altre, ci sono le regole sulla complessità aciclica dei path (di norma inferiore a 200) oppure quelle sulla complessità delle classi, che verificano la quantità di classi e metodi;

Controversial : insieme di regole che sono considerate controverse. Servono maggiormente per evidenziare eventuali problemi e contrasti nei ruleset creati dall'utente;

Coupling : insieme di regole che trovano oggetti e package che hanno un livello di accoppiamento troppo alto;

Design : insieme di regole che evidenziano errori in progettazione, suggerendo inoltre soluzioni alternative, ad esempio regole sugli attributi o metodi statici;

Empty Code : insieme di regole che rilevano la presenza di istruzioni vuote, come ad esempio blocchi *try-catch* o metodi vuoti;

Optimization : insieme di regole che vengono utilizzate per ottimizzare il codice, ad esempio il miglioramento dell'utilizzo di array e cicli;

Security Code Guidelines : insieme di regole che controllano la sicurezza del codice secondo le linee guida pubblicate da Sun (`http://java.sun.com/security/seccodeguide.html#gcg`).

3.3 CPD

CPD(Copy/Paste Detector) è un componente incluso in PMD che consente la ricerca di codice duplicato.

Questa è un'operazione che, soprattutto in progetti composti da molti file, può essere lunga e comportare grandi perdite di efficienza.

Può essere utilizzato sia insieme a PMD che separatamente, e supporta molti linguaggi oltre quelli di PMD, tra cui Fortran, Matlab, ObjectiveC, Php e Swift.

Checkstyle

Checkstyle è un tool che permette di controllare l'aderenza agli standard di programmazione di codice scritto in Java.

Può supportare ogni tipo di standard personalizzato dall'utente, ma offre di default le regole del *Sun Code Conventions* e di *Google Java Style*.

A differenza dei precedenti strumenti, questo software è specializzato per trovare errori di stile di programmazione più che bug, così da rendere il codice più leggibile.

4.1 Funzionamento

Checkstyle, a differenza degli strumenti precedenti, non ha bisogno né di compilazione né del supporto del JavaCC, ma riesce a calcolarsi autonomamente un proprio Abstract Syntax Tree per operare la ricerca.

É composto da vari moduli strutturati ad albero, la cui radice è il modulo *Checker* che ha la responsabilità di caricare i moduli successivi. Il secondo livello dell'albero può essere composto da:

- **FileSetCheck**, moduli che agiscono direttamente su un insieme di file e restituiscono gli errori;
- **Filters**, moduli che filtrano gli errori ricevuti dall'analisi del codice cercando di eliminare le istanze non rilevanti e i falsi positivi;
- **AuditListener**, moduli che gestiscono le occorrenze ricevute dall'analisi.

Di notevole rilevanza è il modulo *TreeWalker* dei FileSetCheck, che permette di accedere ad ogni singolo file del progetto Java e applicarvi le regole dei bug definite nel file di configurazione.

Un esempio del file di configurazione è il seguente:

```
<module name="Checker">
    <module name="JavadocPackage"/>
    <module name="TreeWalker">
        <property name="tabWidth" value="4"/>
        <module name="AvoidStarImport"/>
        <module name="ConstantName"/>
        ...
    </module>
</module>
```

4.2 Checks

Di default, Checkstyle fornisce numerosi check definiti per funzionalità, di cui alcuni di seguito:

Annotation : insieme di moduli che verificano la correttezza e la posizioni di commenti ed annotazioni;

Block : insieme di moduli che controllano tutto ciò che riguarda i blocchi, a partire dalle parentesi fino al controllo del contenuto effettivo dei blocchi;

Class Design :insieme di moduli riguardanti la progettazione delle classi, come il corretto utilizzo delle classi astratte oppure il controllo di visibilità di metodi o classi di utilità;

Coding : insieme di moduli che esaminano semplici errori di programmazione, come l'assenza del caso di default in uno switch;

Header : controlli sui file che devono avere una determinata intestazione;

Imports : insieme di moduli che verificano la presenza di errori nell'importazione di classi o package, come importazioni ridondanti oppure inutili;

Javadoc : moduli che controllano l'aderenza del codice alle regole definite nella Javadoc;

Metrics : insieme di moduli che misurano le caratteristiche del codice, come la complessità delle condizioni oppure l'accoppiamento tra moduli;

Naming Convention : tutti i moduli che controllano l'esatto utilizzo di una naming convention;

Size Violation : insieme di moduli che verificano che la lunghezza di file, classi, metodi, etc. sia corretta rispetto alle impostazioni inserite dall'utente.

Inoltre questo strumento consente all'utente di definire le proprie regole per la ricerca di bug semplicemente estendendo la classe astratta *AbstractCheck* fornita da Checkstyle e andando a ridefinire i suoi metodi, i quali utilizzano i *TokenTypes*: questi contengono le definizioni di ogni istanza che è possibile trovare nel codice, come la presenza di un determinato carattere oppure la definizione di un metodo o di una classe.

Una volta sviluppato il sorgente, bisogna includere il modulo come figlio del *TreeWalker*.

SonarLint

SonarLint è un plugin open source prodotto da Sonar, disponibile per gli IDE Eclipse, IntelliJ, Atom, Visual Studio e VS Code; consente di identificare bug o errori nel codice sorgente di un programma mediante bug pattern.

É possibile eseguirlo sia in versione autonoma sia in "connected mode" collegato a *SonarQube*, che consente di implementare nuove regole di ricerca di bug, aggiungere strumenti di terze parti nella stessa esecuzione e una gestione migliore dei report e delle statistiche sulla qualità del codice.

Questo plugin consente sia di analizzare il codice "on-the-fly", ossia controllando il codice continuamente mentre lo si scrive, oppure a comando, lasciando all'utente la scelta di quando eseguirlo.

SonarLint è estensibile attraverso plugin scritti dall'utente, ma soltanto nella modalità connessa a SonarQube.

Permette di trovare problemi riguardanti:

- codice duplicato;
- aderenza agli standard sulla scrittura del codice;
- test di unità;
- codice troppo complesso;
- potenziali bug;
- commenti;

- progettazione ed architettura del programma.

SonarLint include nativamente tutti i code analyzer forniti da Sonar per un totale di 20 linguaggi, tra cui SonarJava, SonarCFamily, SonarPython e SonarSwift.

5.1 SonarJava

SonarJava è il code analyzer di Sonar per il linguaggio Java.

Per eseguire la ricerca, SonarJava ha bisogno del bytecode del programma, quindi prima della sua esecuzione il codice sorgente deve essere compilato.

La differenza tra SpotBugs e SonarJava è, però, nell'applicazione: il primo viene applicato soltanto sul bytecode, mentre il secondo viene applicato sia sul codice sorgente che sul bytecode.

SonarJava ricava dal bytecode un *modello semantico* del codice, il quale fornisce informazioni riguardanti tutti i simboli che sono stati manipolati, a partire dai metodi fino ad arrivare alle singole istanze delle classi.

Attraverso le proprie funzionalità, rese pubbliche dalle API di Sonar, e che sono adoperate per la scrittura dei propri bug pattern, si possono ottenere tutti i dati riguardanti i simboli analizzati, come il proprietario di un metodo, le sue funzionalità, le eccezioni che può lanciare ed il tipo di ritorno.

5.2 Bug Rules

SonarJava definisce di default più di 403 regole per il riconoscimento dei bug, le quali sono definite nelle seguenti tre tipologie:

Bug : errori che possono portare ad un comportamento inatteso durante l'esecuzione, alcuni esempi possono essere l'uso errato della funzione *wait()* con i thread oppure la mancata chiusura di uno Stream Input/Output;

Code Smells : debolezze di progettazione del software che rendono più probabile la presenza di bug, come ad esempio la presenza di codice duplicato oppure morto, la definizione di classi troppo lunghe o espressioni troppo complesse;

Vulnerabilities : difetto del codice che diminuisce il livello di sicurezza del programma sviluppato, rendendolo quindi vulnerabile ad attacchi malevoli, ad esempio un uso scorretto dei modificatori di accesso oppure l'utilizzo di pacchetti non verificati ricevuti dalla rete.

Inoltre definisce anche una serie di standard ai quali verifica la compatibilità, i quali sono:

- CWE;
- SANS TOP 25;
- OWASP;
- MISRA;
- CERT.

Applicazione su programma Java: Spring Boot

SpringBoot è un software open source per la configurazione di programmi basati su *Spring*, una piattaforma per lo sviluppo di applicazioni enterprise web e stand-alone in Java.

Questo framework implementa il design pattern *Dependency Injection*, il quale facilita lo sviluppatore in vari compiti, come l'accesso ad un database, il test del prodotto oppure la gestione delle dipendenze.

SpringBoot consente di configurare automaticamente le applicazioni per agire con Spring, fornisce gli strumenti per una configurazione rapida con *Maven* e permette di creare e compilare facilmente applicazioni stand-alone di qualità elevata sempre basate su Spring.

In questo lavoro, ho analizzato il codice sorgente di SpringBoot, scaricato da *GitHub*, con i quattro strumenti precedentemente indicati, e ho confrontato i risultati ottenuti notando similarità e differenze.

La versione di SpringBoot prelevata da GitHub è quella del head branch *master* (v2.0.0.M6) del 5 novembre 2017.

Ho utilizzato i tool nelle loro versioni di plugin per l'IDE Eclipse, prelevandone gli output dalla console e dai report da loro generati.

Considerata l'elevata mole dei dati ricevuti, ho realizzato dei semplici programmi Java che sintetizzassero i risultati ottenuti da ogni singolo strumento, per rendere

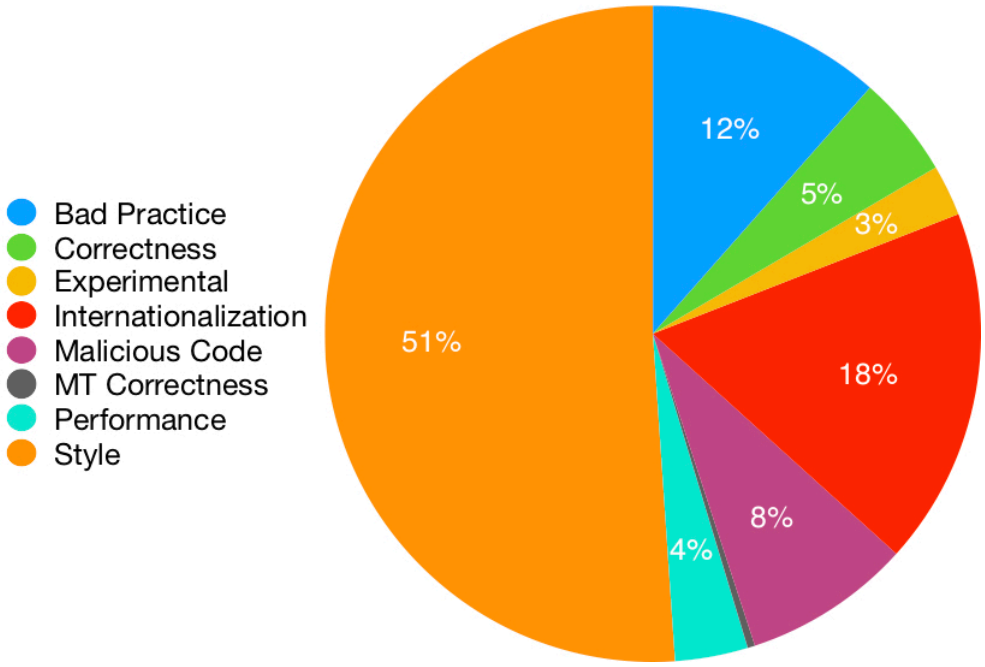

Figura 6.1: Risultati SpotBugs

la successiva analisi più semplice e lineare: tali tool sono discussi nella **Appendice A**.

6.1 SpotBugs

Per l'analisi con SpotBugs sono state considerate tutte le categorie di bug pattern.

Sono stati identificati **278** bug in tutto il codice sorgente, di cui 38 di tipologie diverse.

I risultati dell'analisi sono disponibili in **figura 6.1**.

Come si evince dal grafico, SpotBugs ha rilevato per il 51% delle occorrenze bug della categoria *Style*, cioè i bug pattern Dodgy Style, tra cui il più rilevato è il bug **NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE**, che indica il

mancato controllo del valore ”*null*” come valore di ritorno di una funzione, che potrebbe quindi sollevare un’eccezione ”*NullPointerException*”.

Inoltre, sono stati rilevati molti errori della categoria *Internationalization*, come il bug **DM_DEFAULT_ENCODING**, relativo a possibili errori di codifica dei byte facendo il cast a String.

Il bug con priorità più alta è **RV_EXCEPTION_NOT_THROWN**, della categoria *Correctness*, che rappresenta la creazione di un’eccezione che non è neanche lanciata: analizzando il codice, il bug si trova in una classe di test che va a provare un’eccezione scritta dagli sviluppatori, quindi possiamo considerare l’errore come non rilevante nel caso specifico.

Le anomalie riscontrate appartenenti alla categoria *Malicious Code* sono di priorità più bassa, poiché SpotBugs ha rilevato errori sulla visibilità di riferimenti a variabili interne delle classi.

6.2 PMD

PMD, analizzando il codice con tutti i rule set di default, ha rilevato **29763** errori nel codice, di cui 3210 bug diversi.

Tra questi, ha identificato 8173 bug di *LawOfDemeter*, del rule set *Coupling*, che misura l’accoppiamento tra classi ed oggetti: in particolare, la legge di Demetra definisce che un oggetto può chiamare soltanto le funzioni proprie, dei suoi parametri, degli oggetti che crea e dei suoi componenti diretti, quindi una sua violazione comporterebbe un elevato accoppiamento tra i componenti considerati.

Un altro errore molto comune, con 1783 occorrenze, è il *CommentSize* del rule set *Comments*, che identifica commenti la cui grandezza è eccessiva, rendendo quindi il codice più complesso da leggere.

Più interessanti sono i bug *DataflowAnomalyAnalysis* del rule set *Controversial*, che vanno ad analizzare il dataflow, controllando definizioni e riferimenti: tra le 386 anomalie riscontrate, 99 si riferiscono a riferimenti a variabili non definite che rappresentano un bug che può portare ad un errore a tempo di esecuzione.

Tra le altre anomalie, ne sono state trovate molte riguardanti i rule set di *Design*, tra cui errori su variabili dichiarate o dichiarabili "*final*", la mancanza di *break* nelle istruzioni *switch* e l'uso di variabili il cui nome è troppo lungo.

Invece, il software integrato per la ricerca di codice duplicato *CPD*, ricercando come da impostazioni predefinite pezzi di codice di 25 righe, non ha rilevato anomalie.

6.3 Checkstyle

Nell'analizzare SpringBoot con Checkstyle, utilizzando i check forniti dal rule set *Google Checks*, sono stati evidenziati **85465** errori.

La maggior parte è composta da errori di tipo *Indentation* della classe *Miscellaneous* (34708 occorrenze), che controlla l'indentazione del codice, e di tipo *FileTabCharacter* della classe *Whitespace*, che ricerca la presenza dei caratteri "tab".

Poiché questi errori rappresentano il gran numero delle occorrenze e non sono eccessivamente rilevanti, nelle analisi che seguono sono stati ignorati: quindi, di seguito nella **figura 6.2**, sono presenti i dati relativi all'analisi privata degli errori precedenti.

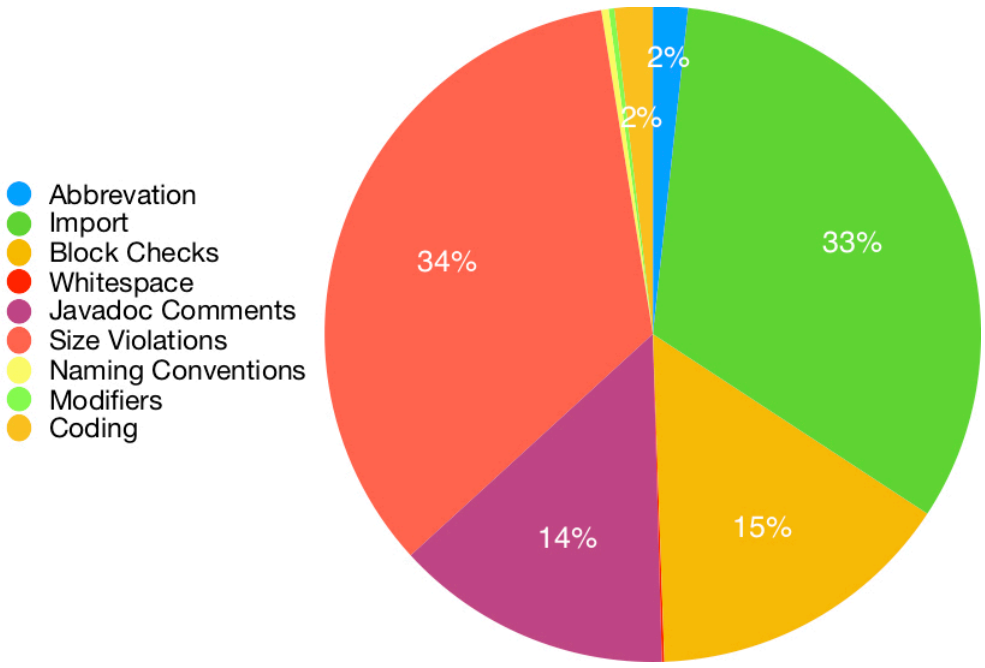

Figura 6.2: Risultati Checkstyle

Dal grafico si nota che la maggior parte degli errori appartiene ai check di *Size Violation*, come ad esempio la grandezza eccessiva di una funzione, di una classe o della stessa riga di codice.

L'analisi ha restituito anche problemi sulle annotazioni della Javadoc e sui blocchi, come la posizione delle parentesi graffe nel codice.

Inoltre, ci sono molti risultati dei check *Import*, che riguardano l'ordine delle importazioni nelle diverse classi.

6.4 SonarLint

SonarLint, attraverso i bug pattern di default di SonarJava, ha evidenziato la presenza di **440** errori, di cui 230 istanze diverse.

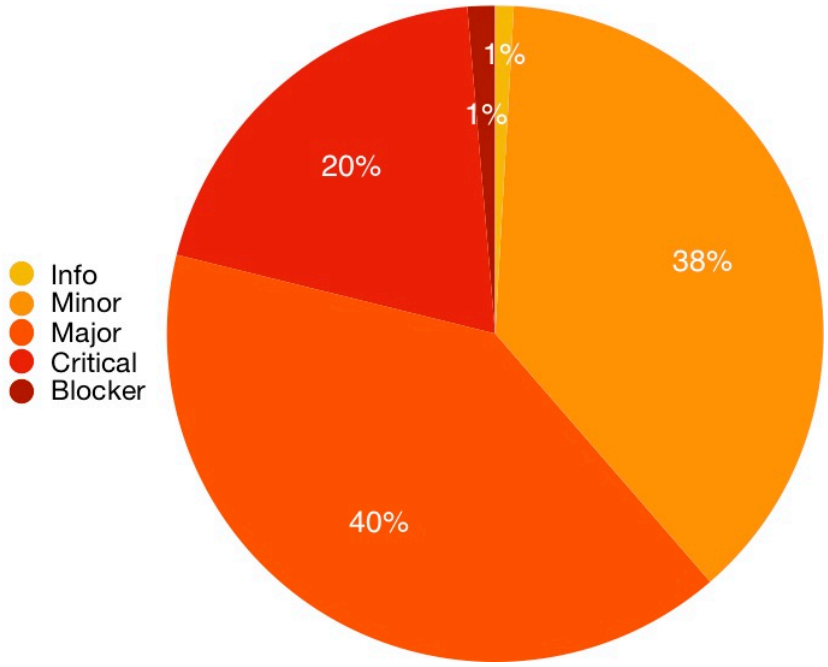

Figura 6.3: Risultati SonarLint

Su 440 anomalie riscontrate, 40 rappresentano Bug, 8 Vulnerabilities e le restanti Code Smells.

In **figura 6.3** sono presenti le proporzioni degli errori a seconda dei livelli di priorità riscontrati da SonarLint.

Gli errori più gravi sono quelli marcati come *Blocker*, perché anomalie che possono bloccare l'esecuzione del software; sono stati rilevati 6 di questi errori, di cui 3 diversi:

- *Methods and field names should not be the same or differ only by capitalization*: appartiene alla classe dei Code Smells, indica il nome di un'istanza di una classe che differisce dal nome della classe soltanto per le lettere maiuscole;

- *Resources should be closed*: appartiene alla classe dei Bugs, indica che oggetti come connessioni, stream, ecc., non sono stati chiusi;
- *Child class fields should not shadow parent class fields*: appartiene alla classe dei Code Smells, indica una classe che definisce una proprietà con lo stesso nome di una della classe padre.

Andando a ricercare nel codice sorgente le parti di codice che hanno scatenato tali anomalie, è evidente come in tutti i 6 casi SonarLint ha identificato correttamente gli errori nel codice, e questi rappresentano dei bug abbastanza rilevanti nel contesto dell'applicazione.

Gli altri bug di priorità più alta rilevati sono relativi all'uso di costanti invece di stringhe duplicate, la definizione di una variabile come "*Serializable*", metodi con corpo vuoto, variabili da rinominare oppure il tentativo di catturare una eccezione "*Exception*" invece della sua classe genitore "*Throwable*".

I rilevamenti di priorità minore si riferiscono per la maggior parte a funzioni marcate come *Deprecated* oppure ad ambiguità nel codice.

Conclusioni

In questo lavoro sono stati analizzati quattro strumenti di analisi statica, SpotBugs, PMD, Checkstyle e SonarLint, per poi applicarli su un programma Java di dimensioni abbastanza grandi, quale è SpringBoot.

Dopo l'applicazione dei software e l'analisi dei prodotti ottenuti, è possibile estrapolare una serie di risultati.

SpotBugs è riuscito ad identificare anomalie di vario genere, a partire da problemi relativi alla grammatica del codice fino ad arrivare ad errori più difficilmente rilevabili, come quelli relativi alle categorie Correctness ed Internazionalization: i risultati ottenuti sono in un numero sufficientemente piccolo da poter essere tutti analizzati ed approfonditi semplicemente per migliorare il codice.

PMD ha identificato molti errori relativi all'accoppiamento tra le classi e al dataflow di ogni istruzione, oltre ad anomalie legate alla lunghezza del codice, che lo rendono un valido strumento per migliorare la qualità generale del software: ha però restituito un numero molto alto di errori, che lo rendono dunque più utile qualora si riescano a filtrare gli errori per estrapolarne quelli più importanti e rilevanti.

Checkstyle ha rilevato moltissime anomalie che riguardano lo stile del programmatore, come l'indentazione del codice oppure l'uso dei "tab", ma anche il miglioramento dell'importazione dei vari package e dei blocchi di ogni funzione: il report generato, però, era composto da un numero molto grande di errori, che hanno reso l'analisi molto lunga e difficoltosa, considerando anche che molte anomalie rilevate

sono relative alla soggettività del programmatore e non intaccano la qualità del software.

SonarLint è stato in grado di individuare molti problemi differenti, sia di contorno al codice sia molto rilevanti, rendendo l'analisi dei report molto più semplice grazie alla suddivisione delle anomalie per priorità e facilitando la comprensione dei singoli bug mediante l'accesso rapido alle loro definizioni: i bug rilevati sono della quantità giusta per essere gestiti e compresi per adeguare il codice.

Dai risultati ottenuti, si nota come SpotBugs e SonarLint sono gli strumenti che hanno reso una descrizione più ampia degli errori del sistema, fornendo output differenziati e riguardanti vari aspetti del codice; PMD ha fornito risultati più accurati, soprattutto per quanto riguarda l'analisi dei dataflow e dell'accoppiamento tra le classi, anche utilizzando CPD per la ricerca del codice duplicato; Checkstyle, invece, ha fornito molti risultati che non sono di primaria importanza al fine di migliorare la qualità del prodotto finale, perché molto relativi allo stile di scrittura del codice e alla sua espressione formale, ma comunque non ha tralasciato evidenze importanti.

Esaminando i cambiamenti tra un'altra versione di SpringBoot (base branch 1.0.x) e la versione analizzata in questo lavoro, si può vedere come ci siano state correzioni da parte della community di SpringBoot che equivalgono ad alcune delle anomalie riscontrate applicando i SpotBugs, PMD, Checkstyle e SonarLint: tra queste ci sono rettifiche che riguardano l'indentazione del codice, problemi riguardanti i parametri delle funzioni, la gestione e il controllo dei valori "*null*" con le relative eccezioni; queste revisioni sono state applicate nel mese di novembre 2017.

In conclusione, questi quattro software sono tutti importanti per l'analisi statica del codice, ma possono essere usati in momenti differenti: idealmente, potrebbero essere utilizzati, soprattutto Checkstyle, durante lo sviluppo del software e non soltanto sul prodotto finito, perché restituiscono risultati che, corretti di volta in volta, possono generare un codice migliore sotto molti punti di vista e facilitare il compito agli sviluppatori; ciò non toglie che potrebbero essere applicati anche alla fine dello sviluppo di parti o dell'intero software, ed essere applicati come parte dei test di integrazione o del test del prodotto, così da avere un confronto di insieme e produrre un software di elevata qualità.

Analisi dei report

I risultati ottenuti dall'analisi con questi tool non sono sempre di semplice interpretazione: infatti, soprattutto per i report molto grandi come quello generato da Checkstyle, è difficile estrapolare informazioni rilevanti e utili per migliorare il codice.

Mantenendo l'esempio di Checkstyle, parte del report generato dal suo plugin per Eclipse, e copiato dalla sua console, è il seguente:

```
...
Abbreviation As Word In Name: Abbreviation in name 'XADataSourceWrapper'
must contain no more than '2' consecutive capital letters.
XADataSourceWrapper.java /spring-boot/src/main/java/
org/springframework/boot/jta line 31 Checkstyle Problem
Custom Import Order: Import statement for 'org.assertj.core.
api.Assertions.assertThat' is in the wrong order.
Should be in the 'STATIC' group, expecting not
assigned imports on this line. AbstractDataSourcePoolMetadataTests.java
/spring-boot/src/test/java/org/springframework/boot/jdbc/metadata
line 25 Checkstyle Problem
...
```

In questo caso, l'output è composto dal nome dell'errore riscontrato, una sua breve descrizione e la posizione del bug all'interno della classe indicando anche il path all'interno del progetto.

Per gestire il numero eccessivo di occorrenze, che nel caso specifico sono 85465, serve necessariamente un tool che possa sintetizzarne i risultati.

Dunque ho realizzato un semplice programma in Java, riportato di seguito:

```
public class Checker {
  public static void main(String[] args) {
    String fileName="Spring Boot - Checkstyle #1.txt";
    File file=new File(fileName);
    int occ, clOcc;
    HashMap<String, Integer> bugs=new HashMap<String, Integer>();
    HashMap<String, Integer> classi=new HashMap<String, Integer>();
    try {
      String[] bug, cl;
      BufferedReader br = new BufferedReader(new FileReader(file));
      BufferedWriter writer=new BufferedWriter(new
       FileWriter(new File("Checked - "+fileName)));
      BufferedWriter writer2=new BufferedWriter(new
       FileWriter(new File("Checked Classes - "+fileName)));
      String line;
      while ((line = br.readLine()) != null) {
        bug=line.split(Pattern.quote("."));
        cl=line.split(Pattern.quote(":"));
        if(bugs.containsKey(bug[0])) {
          occ=bugs.get(bug[0]);
          occ++;
          bugs.put(bug[0], occ);
        }else {
              bugs.put(bug[0], 1);
        }
        if(classi.containsKey(cl[0])) {
```

```
                clOcc=classi.get(cl[0]);
                clOcc++;
                classi.put(cl[0], clOcc);
          }else {
                classi.put(cl[0], 1);
          }
        }
        br.close();
        writer2.write("Occorrenze per classe:\n");
        Map<String, Integer> treeMap = new TreeMap<String, Integer>(bugs);
        Map<String, Integer> treeMap1 = new TreeMap<String, Integer>(classi);
        System.out.println(fileName+" - Bug totali= "+bugs.size()+"\n\n\n");
        int i;
        for(i=0; i<treeMap.size(); i++) {
          String key=treeMap.keySet().toArray()[i].toString();
          writer.write("BUG: "+key+"\n OCCORRENZE: "+treeMap.get(key)+"\n");
        }
        for(i=0; i<treeMap1.size(); i++) {
          String key1=treeMap1.keySet().toArray()[i].toString();
          writer2.write("\nCLASSI: "+key1+"\n OCCORRENZE: "+treeMap1.get(key1));
        }
        writer.close();
        writer2.close();
      }catch(IOException ex) {
        ex.printStackTrace();
      }
    }
}
```

Questa classe produce 2 file di output differenti: un primo file, *Checked*,

stampa ogni bug riscontrato con il suo nome, la sua descrizione e il numero di volte che esso è stato riscontrato.

Di seguito un estratto:

```
BUG: Indentation: 'block' child has incorrect indentation level 40,
 expected level should be 34
OCCORRENZE: 1
BUG: Indentation: 'block' child has incorrect indentation level 40,
 expected level should be one of the following: 20, 22, 24
OCCORRENZE: 3
BUG: Indentation: 'case' child has incorrect indentation level 24,
 expected level should be 8
OCCORRENZE: 10
```

Questo file è stato utile per identificare quali errori sono occorsi più volte nello stesso file e non solo.

Il secondo file, *Checked Classes*, invece, restituisce il numero di volte che un bug è stato rilevato *in tutta l'analisi*: questo file è stato molto utile perché, nel caso specifico di Checkstyle, ha consentito di individuare quali errori erano stati rilevati più volte, così da rendere l'analisi più semplice e snella.

Di seguito una parte del file:

```
CLASSI: Empty Catch Block
OCCORRENZE: 10
CLASSI: Empty Line Separator whitespace
OCCORRENZE: 3
```

```
CLASSI: File Tab Character whitespace
OCCORRENZE: 48259
CLASSI: Indentation Miscellaneous
OCCORRENZE: 34708
CLASSI: Javadoc Paragraph
OCCORRENZE: 203
```

Per gli altri strumenti, sono stati realizzati dei programmi molto simili a quello precedentemente descritto che generano lo stesso tipo di output, agendo però su report con formati leggermente differenti, così da rendere l'analisi dei report più semplice anche per gli altri tool.

Bibliografia

[1] Sommerville I., *Ingegneria del Software*, Pearson, 8 ed., 2015

[2] *SpotBugs*, `https://spotbugs.github.io`

[3] *PMD*, `https://pmd.github.io`

[4] *Checkstyle*, `http://checkstyle.sourceforge.net`

[5] *SonarLint*, `https://www.sonarlint.org`

[6] *SonarJava*, `https://www.sonarsource.com/products/codeanalyzers/sonarjava.html`

[7] *Spring*, `https://spring.io`

[8] *SpringBoot*, `https://projects.spring.io/spring-boot/`

[9] *SpringBoot*, `https://github.com/spring-projects/spring-boot`, head branch master, v2.0.0.M6