Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica

UN METODO DI ANALISI STATICA DI QUALITÀ DEL SOFTWARE

Relatore:
Chiar.mo Prof.
Paolo Ciancarini

Presentata da:
Eleonora Pirri

III Sessione
Anno Accademico 2019/2020

Keywords:

SonarQube

Analisi Statica

Qualità del Software

Debito Tecnico

Indice

1 Introduzione **1**
1.1 Struttura della tesi 3

2 La qualità software **5**
2.1 L'importanza della qualità del software 5
2.2 Debito tecnico 6
2.3 Definizione qualità del software 8
2.4 Mantenere alta la qualità 10

3 Analisi statica e SonarQube **15**
3.1 Analisi statica 15
3.2 SonarQube 16

4 Analisi di un progetto con SonarQube **21**
4.1 Installazione e avvio di SonarQube 21
4.2 Analisi tramite SonarQube 22
4.3 Progetti analizzati 26
4.4 Risultati e conclusioni 28
4.5 Gli studenti e SonarQube 35

5 Conclusioni **37**

Questa pagina è lasciata intenzionalmente bianca.

Capitolo 1

Introduzione

La qualità del software è una tematica su cui è necessario soffermarsi perché influenza il costo, l'affidabilità e la longevità dei prodotti software. Conseguentemente è fondamentale avere degli strumenti e delle metriche per tenere sotto controllo la qualità durante l'intero ciclo di vita di un prodotto.
In questo elaborato viene presentato lo strumento SonarQube, ad oggi disponibile per ventisette linguaggi di programmazione.
SonarQube ha registrato un aumento di popolarità negli ultimi anni. È stato adottato da 120 000 utenti[1], tra cui più di 100 000 progetti open source[2].
L'uso di strumenti automatici per l'analisi statica permette di migliorare i fattori interni di qualità del software e rivela problemi del codice senza dover eseguire il programma.
Mediante l'analisi statica del codice sorgente misura i livelli di affidabilità, manutenibilità, sicurezza, complessità, copertura dei test e duplicazione del codice, assegnando un punteggio dipendentemente dalla severità dei difetti riscontrati. L'opportunità di usare uno strumento di immediata applicazione come SonarQube porta a una qualità più alta e a una produttività maggiore per i programmatori e agevola una sorta di "standardizzazione" del codice, fattore importantissimo dato che parti di prodotti software vengono condivise spesso e raggiungono persone con tipi di background

[1] https://www.sonarqube.org/
[2] https://sonarcloud.io/explore/projects

diversi e nel contesto odierno è molto comune la pratica del social coding. L'utilizzo di un tool come SonarQube inoltre si collega bene alla pratica della continua integrazione che include il controllo statico del codice dopo ogni cambiamento del codice [1]. La pratica della continua integrazione nasce nel contesto della metodologia agile.
La qualità del software è fortemente correlata alle metodologie di sviluppo adottate perché da queste dipendono le attività di testing, di progettazione e di realizzazione e la produzione della documentazione relativa alle diverse fasi del ciclo di vita del prodotto.

- Waterfall

Il metodo waterfall (o modello a cascata) prevede una sequenza di fasi in cui l'output di una fase diventa l'input della seguente. Ne esistono molte varianti che differiscono nel numero e nella natura delle fasi, ma tutte hanno in comune dei principi fondamentali, come il tipo di approccio predittivo (la tendenza alla pianificazione di gran parte del processo per intervalli di tempo molto lunghi, seguendo il principio per cui lo sviluppo software dovrebbe essere predittivo e ripetibile), una ampia documentazione prodotta ed è un tipo di sviluppo orientato ai processi e ai tool.
Il ciclo di vita di un software sviluppato attraverso il modello a cascata consta le seguenti fasi:

1. Requisiti: la descrizione del comportamento del sistema da sviluppare, stabilito con i clienti. I requisiti vengono raccolti, analizzati e viene prodotta la relativa documentazione, a cui si farà riferimento negli altri stadi dello sviluppo.

2. High Level Design: la formulazione di una implementazione per i requisiti raccolti. Viene determinata l'architettura del sistema, viene scelto lo schema concettuale del database, vengono definite le strutture dati da usare e steso il documento di progetto che identifica i moduli e le loro interfacce.

3. Codifica e integrazione: vengono implementati i moduli e i loro test e vengono integrati i moduli tra loro.

4. Testing: si effettuano test di sistema e si controlla che la soluzione software prodotta rispecchi davvero i requisiti originali. In questa fase vengono trovati i bug e i glitch nel sistema.

5. Manutenzione: le modifiche, i miglioramenti, la correzione degli errori eseguiti dopo il rilascio del prodotto.

Il processo a cascata è quindi pianificato, molto dettagliato, con un'ampia produzione di documentazione e si adatta a grandi organizzazioni gerarchizzate, ma è molto rigido, sono difficili modifiche ai requisiti e in generale ai prodotti delle fasi già superate. Inoltre la versione del software funzionante si ha solo alla fine del processo e il cliente è poco coinvolto nello sviluppo.

- Metodologie agili

Le metodologie agili sono un insieme di processi di sviluppo in cui l'attenzione è posta sulla soddisfazione del cliente, la possibilità di cambiare i requisiti in qualsiasi momento del ciclo di vita del prodotto e i rapporti tra gli individui che prendono parte al processo di sviluppo [2]. Il software viene prodotto in modo incrementale attraverso dei cicli ripetuti iterativamente, detti *sprint* alla fine dei quali si ha un prodotto pronto per il mercato. Ogni incremento aggiunge funzionalità o migliora il software realizzato precedentemente.
I metodi agili sono adattivi, adeguandosi bene ai cambiamenti e sono orientati all'individuo, basandosi sull'idea che il ruolo di un processo di sviluppo è quello di supportare il team durante la sua attività.
Nel metodo a cascata la qualità è raggiunta grazie alla fase di sviluppo del testing. I metodi agili si combinano bene con delle pratiche che aiutano a mantenere alta la qualità come l'integrazione (e l'ispezione) continua.

1.1 Struttura della tesi

Questa tesi è strutturata come segue: Nel secondo capitolo viene introdotto il concetto di analisi statica e presentato SonarQube, uno strumento open source per la verifica della qualità. Nel terzo capitolo viene illustrato l'utilizzo di SonarQube e presentati i risultati di alcune analisi di progetti

open source di dimensioni diverse e scritti in linguaggi diversi.
Nell'ultimo capitolo vengono esposte le conclusioni e degli spunti per futuri lavori.

Capitolo 2

La qualità software

2.1 L'importanza della qualità del software

Il software è un prodotto che rapidamente si è diffuso con facilità, il più ampiamente usato della storia, ma è tra quelli che soffre maggiormente di malfunzionamenti dovuti alla sua bassa qualità. Proprio a causa della pervasività del software, i malfunzionamenti incidono sulla vita di tutti, non solo degli addetti ai lavori: aerei che arrivano in ritardo, vulnerabilità di sicurezza, macchinari medici che hanno guasti improvvisi, errori in sistemi di frenata. Nel 2012 il Knight Capital Group ha subito una perdita di 440 milioni di sterline in 45 minuti a causa di un errore nella distribuzione di un software sui propri server. Il servizio di ambulanze londinese a causa di upgrade ha registrato ritardi nel 2006 e nel 2017 l'intero sistema è andato in crash per sei ore. La presenza di *race condition* in un energy management system basato su Unix è stata la causa di un blackout dalla durata di 14 giorni che ha interessato l'Ontario e gli Stati Uniti d'America nord-orientali. Nella maggior parte degli ambiti è comune scegliere un compromesso tra qualità e costo, così si tende ad associare maggiore qualità a un costo maggiore. Questo non vale per la qualità del software, che quando è mantenuta alta riesce a garantire maggiore sicurezza e rende possibile abbassare i costi durante l'intero ciclo di vita del prodotto, diminuire il tempo di mantenimento e creare valore per produttori, clienti e finanziatori. L'alta qualità del software è stata correlata da [3] a:

- Contrazione dei tempi di testing e di consegna;
- Riduzione di più del 50% degli interventi di riparazione e modifica;
- Aumento della soddisfazione dei clienti;
- Riduzione, dopo il rilascio, dei costi di manutenzione;
- Riduzione dei contenziosi sui contratti dei progetti;
- Riduzione dei progetti cancellati;
- Aumento dell'affidabilità;
- Riduzione delle falle di sicurezza nelle applicazioni rilasciate.

2.2 Debito tecnico

L'espressione debito tecnico è una metafora per indicare le soluzioni non ottimali utilizzate in un progetto e così come nel corrispettivo finanziario, un debito tecnico non saldato con il tempo accumula interesse, rendendo più complicato implementare i cambiamenti.
Per [4] il debito tecnico è una collezione di costrutti che rappresentano stratagemmi nel breve periodo, ma creano un contesto in cui i cambiamenti futuri sono molto costosi. La presenza di debito tecnico costituisce un reale e contingente svantaggio il cui impatto è relativo a qualità interne del sistema, e in particolar modo, alla manutenzione e alla evolvibilità.
In [5] vengono definiti cinque componenti del debito tecnico:

- Code debt: il debito tecnico si manifesta spesso con delle forme errate nel codice scritto, per risolverlo è necessario fare refactoring;
- Design and architectural debt: risultato dell'adozione di soluzioni non ottimali o di pattern e tecnologie che sono state soppiantati;
- Environmental debt: il debito tecnico che dipende dall'ambiente (hardware, applicazioni di supporto, infrastrutture) dell'applicazione;

- Knowledge distribution and documentation debt: l'aumento dei costi a causa della mancanza di documentazione ben scritta o di conoscenza condivisa;

- Testing debt: costi maggiori sono associati alla mancanza di test automatici.

Un ulteriore componente del debito tecnico, individuato da [6], è quello rappresentato da fattori sociotecnici, che influenzano il benessere e la produttività dei membri dei team. Tra questi rientrano le modifiche alla struttura dell'organizzazione, il cambio del processo di sviluppo, la collaborazione a tutti i gradi dell'impresa e le decisioni organizzative che modificano il modo con cui i lavoratori interagiscono.
In [7] viene definito un metodo per stimare il debito tecnico, che prevede una prima fase di stima dello sforzo di riparazione e una seconda parte in cui viene considerato lo sforzo di mantenimento. Si presta attenzione quindi alla manutenibilità per ottenere software che possa essere facilmente modificato e migliorato nel tempo.
È possibile suddividere il debito tecnico in base alle ragioni per cui viene contratto [5].
Debito strategico: è un tipo di debito tecnico accumulato per ragioni associate a decisioni strategiche per l'impresa, come per anticipare il rilascio del prodotto sul mercato, rimandando il rimborso di questo tipo di debito al lungo periodo.
Debito tattico: viene contratto in una situazione di ritardo, ad esempio per rispettare una cadenza. Si concretizza con l'utilizzo di soluzioni non ottimali o rimandando l'implementazione di alcune funzioni e solitamente viene colmato nelle release successive. Così come il debito strategico è misurabile e visibile.
Debito incrementale: è il risultato di molte piccole scorciatoie adottare dagli sviluppatori, come la mancanza di commenti o l'utilizzo di nomi generici per variabili e sottoprogrammi. È utile fare refactoring per evitare che cresca troppo e sia difficile da tenere sotto controllo.
Debito involontario: è un tipo di debito che cresce non intenzionalmente. Le cause possono essere di sviste dei programmatori, cattive pratiche di

programmazione o problemi di comunicazione tra i componenti del team.

2.3 Definizione qualità del software

La percezione della qualità è dipendente dal contesto di utilizzo del software e, insieme a delle caratteristiche quantitative, esistono delle caratteristiche soggettive e vincolate al ruolo dell'individuo rispetto al software, per esempio, la percezione di un utente è diversa da quella di uno sviluppatore. Un'altra caratteristica dei prodotti software è la velocità con cui sono necessarie modifiche. Il loro ciclo di vita coinvolge inoltre diversi ruoli: stakeholder, progettisti, sviluppatori, manutentori ed è necessario che la comunicazione tra le parti sia efficace e che per tutti sia chiara l'importanza della qualità. Per questa ragione è fondamentale una definizione di qualità del software condivisa e poco ambigua. Per [3] esistono sette criteri di cui si deve tener conto per dare una definizione di qualità del software affinché possa essere una definizione valida anche in contesti aziendali e per analisi economiche:

1. dovrebbe essere *prevedibile* prima dell'inizio del progetto;
2. dovrebbe essere *misurabile* durante e dopo il termine del progetto;
3. dovrebbe essere *dimostrabile* in caso di controversie;
4. dovrebbe essere *migliorabile* col passare del tempo;
5. dovrebbe essere *flessibile* e includere tutti i prodotti finali;
6. dovrebbe essere *estensibile* e coprire tutte le fasi e tutte le attività;
7. dovrebbe essere *espandibile* per poter essere applicata a nuove tecnologie.

I fattori che sono spesso tenuti in considerazione anche quando non vengono usati modelli standard si suddividono in attributi interni ed esterni.
Alcuni esempi di attributi interni, ovvero quelli correlati alla struttura del software e si riferiscono al modo in cui gli sviluppatori percepiscono la qualità del software sono:

- Manutenibilità: la possibilità di apportare modifiche a sistema realizzato. Si distingue la manutenibilità in correttiva (diagnosi e correzione degli errori), adattiva (modifiche apportate al sistema per far fronte ai diversi ambienti in cui verrà eseguito) e perfettiva (modifiche apportate alle caratteristiche funzionali come l'aggiunta di nuove funzioni);

- Flessibilità: la facilità di modificare il software;

- Portabilità: la capacità di un sistema di funzionare in ambienti diversi;

- Riusabilità: la possibilità di riutilizzare parti di un sistema per realizzare un nuovo prodotto;

- Leggibilità: la caratteristica di codice sorgente facilmente comprensibile;

- Testabilità: la realizzabilità di test è possibile a basso costo.

Esempi di attributi esterni, ovvero quelli legati alla percezione dell'utente:

- Correttezza: la conformità alla sua specifica dei requisiti;

- Usabilità: la facilità per l'utente di interagire con il prodotto;

- Scalabilità: la possibilità di adattare il sistema a contesti con forti differenze di complessità;

- Efficienza: il software usa le risorse in maniera ottimizzata, senza sprechi;

- Affidabilità: la probabilità che si verifichino malfunzionamenti;

- Robustezza: la capacità di un software di continuare a fornire servizi anche in situazioni impreviste (ad esempio guasti hardware, input scorretti).

Al fine di dare una definizione non ambigua e condivisa di qualità, sono stati definiti dei modelli di qualità attraverso i quali è possibile avere uno strumento per misurare e conseguentemente controllare la qualità di un prodotto software. L'obiettivo di questi standard è supportare la creazione di software di maggiore qualità e che possa essere riutilizzato e che ben sopporti i cambiamenti che di consueto interessano il software, come l'aggiunta di nuove funzionalità e la correzione delle funzionalità esistenti.
Secondo [8] è possibile seguire l'evoluzione dei modelli in "Basics Models" (1977-2001) il cui scopo è una valutazione globale del prodotto e in "Tailored Quality Models" (dal 2001 a oggi) che invece sono maggiormente orientati verso la valutazione dei componenti. Lo standard oggi in uso fa parte della famiglia ISO 25000, detta anche SQuaRE - System and Software Quality Requirements and Evaluation, che specifica un insieme di misure, estensioni e modelli di qualità del software.Lo standard ISO/IEC 25010, in particolare, definisce i modelli di qualità del software e la qualità in uso[1] .
Lo standard è nato nel 2007 come evoluzione del modello ISO 9126 e come questo mantiene tre modelli (tre diversi punti di vista) della qualità del software, proponendosi di sintetizzare le visioni e le esigenze di progettisti e utenti.

Lo standard definisce otto categorie di qualità del software: Functional Suitability, Reliability, Usability, Security, Performance Efficiency, Maintainability, Portability, Compatibility, e trentuno sotto-categorie.

2.4 Mantenere alta la qualità

I fattori principali che influenzano la qualità dei prodotti software sono la prevenzione dei difetti e il perfezionamento del processo di sviluppo. Per predire i potenziali difetti software bisogna prestare attenzione ai requisiti, alle metodologie usate per l'individuazione e la prevenzione dei difetti, la complessità ciclomatica del codice e i metodi di testing e la percentuale coperta dai test. Per mantenere alta la qualità occorre quindi agire su queste componenti.

[1] http://www.iso25000.it/styled/

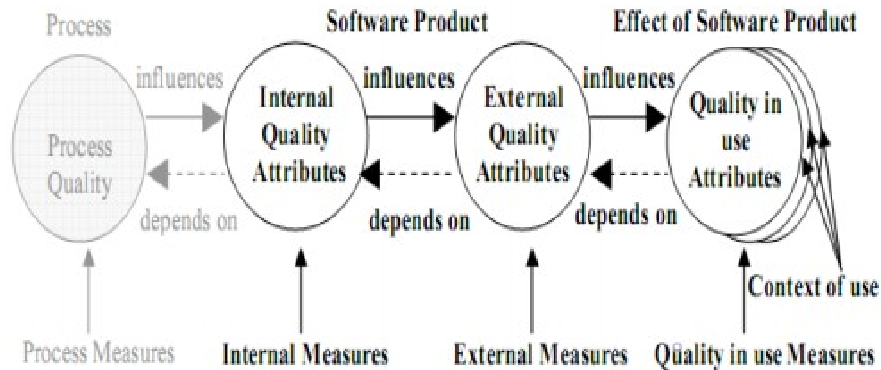

Figura 2.1: Diagramma ISO 9126

Prevenzione dei difetti

È necessario adottare delle strategie di prevenzione dei difetti, e dato che i difetti possono essere di varia natura è consigliabile l'utilizzo di più metodologie adeguatamente combinate tra loro. È importante riconoscere i difetti nelle prime fasi dello sviluppo affinché sia possibile attuare strategie di prevenzione e rimozione dei difetti. Alcune tecniche di prevenzione dei difetti sono:
Pattern e certificazione di materiali riutilizzabili: è comune il riutilizzo di codice sorgente, documentazione utente, design, requisiti e test anche in progetti estremamente diversi rispetto a quelli per cui sono stati ideati originariamente. È un buon metodo di prevenzione dei difetti quando il materiale riutilizzabile è certificato da un terzo ente.
Previsioni automatizzate della qualità: è possibile ottenere una stima generale della qualità aggregando le misure delle caratteristiche dei singoli componenti del sistema, attività generalmente costosa, ma è possibile sveltire questo tipo di misurazione attraverso dei tool automatici come *SonarQube* che classifica i code smell e propone strategie per la loro risoluzione.
Analisi statica: un metodo di valutazione per la verifica di caratteristiche come la struttura sintattica e l'identificazione di falle di sicurezza.

Requisiti

I requisiti sono spesso incompleti e ambigui, spesso perché i clienti non hanno una visione chiara dell'applicativo richiesto. Requisiti mal formulati o vaghi possono introdurre errori concettuali sul software prodotto, difficili e costosi da riparare nelle altre fasi del ciclo di vita del software. È dunque fondamentale definire un metodo di analisi per trovare errori nelle specifiche e definire dei criteri formali per la completezza, la consistenza e la sicurezza dei requisiti.
I requisiti si dicono consistenti quando due o più requisiti non si contraddicono a vicenda. Si parla di uso consistente dei termini quando gli stessi termini usati in diversi requisiti hanno sempre lo stesso significato. Nelle specifiche devono essere evitati requisiti mutualmente esclusivi e conflitti tra termini.
Si parla di completezza dei requisiti quando non ci sono informazioni implicite o da definire. Si parla di completezza esterna quando non ci sono informazioni mancanti dalla documentazione e assicura che tutte le informazioni possono essere reperite nelle specifiche.

Complessità del codice

Una delle misure per valutare la complessità del codice è la *complessità ciclomatica*, che indica il numero di cammini linearmente indipendenti attraverso il codice sorgente. È definita in riferimento a un grafo che ha come nodi i blocchi del programma uniti da un arco se il controllo può passare da un blocco a un altro. È definita come

$$v(g) = e\breve{\ } n + 2p$$

dove
$v(g)$ è complessità ciclomatica del grafo G,
e è il numero di archi del grafo,
n è il numero di nodi del grafo,
p è il numero di componenti connesse.
Quando la complessità ciclomatica è bassa, il programma è più facilmente

comprensibile e testabile.
Altri tipi di complessità usati sono:

- Complessità algoritmica: dipende dall'algoritmo usato;
- Complessità combinatoria: riguarda il modo in cui i moduli software vanno combinati;
- Complessità computazionale: influenzata dal costo delle istruzioni necessarie;
- Complessità entropica: misura la velocità con cui decade la struttura originaria del codice;
- Complessità organizzativa: dipendente dalla struttura organizzativa all'interno dell'azienda.

Al fine di ridurre i difetti del codice è necessario ridurre la complessità del codice.

Testing

La verifica di un progetto software può essere di tipo analitico o sperimentale. Il primo approccio consiste nell'analizzare il prodotto e la documentazione relativa, è quindi basato su l'analisi di modelli statici del prodotto, l'altro consiste nello sperimentare (testare) il progetto software, richiedendo l'esecuzione del prodotto. Le due tecniche sono complementari.
Le soluzioni proposte fin qui rientrano nell'ambito della verifica analitica, a seguire ci saranno dei suggerimenti su tecniche dinamiche di verifica.
Il termine testing indica un'ampia serie di attività legate all'esecuzione del software da testare e il confronto dei risultati a metriche prestabilite, ad esempio i risultati attesi per misurare la correttezza del software. È possibile suddividere i test basandosi sulla porzione di codice coinvolta:

- **Test in piccolo**

Test utilizzato per l'analisi di singoli componenti software, come moduli o sottoprorgammi ed è trattabile con due approcci: il test *white-box* e il test *black-box*.
Il test white-box (test a scatola bianca o trasparente) prevede la conoscenza del codice, ma non è necessario conoscerne la specifica. È detto anche test *strutturale* perché utilizza la struttura interna del programma per ricavare i dati del test, un esempio di strategia di test white-box è quello di scegliere i casi in cui tutte le istruzioni della porzione di codice vengono analizzate (criterio della copertura delle istruzioni). Il principio di copertura completa può essere esteso anche alla struttura del programma, descrivendola come una rappresentazione grafica del flusso di controllo e verificando che ogni arco del grafo venga percorso almeno una volta (criterio della copertua degli archi.
Il test black-box (test a scatola nera) si effettua senza conoscere il modo in cui il software è scritto o organizzato, ma conoscendo la specifica per sviluppare i casi di test e analizzare i risultati. È detto anche test *funzionale* perché è basato su ciò che deve calcolare il frammento di programma testato. Alcune delle tecniche usate per i test black-box sono: la tecnica dei grafi-causa effetto, il test basato sulle tabelle di decisione, il test delle condizioni di confine.

- **Test in grande**

Per progetti di grandi dimensioni i test in piccolo risultano insufficienti. Le tecniche di testing dovrebbero riflettere l'organizzazione dell'attività progettuale, per questa ragione buone tecniche di progettazione portano a ottenere software più facilmente testabile e quindi in cui è possibile catturare ed eliminare errori. In questa categoria di test rientrano il test di integrazione bottom-up e top-down(si verifica la corretta integrazioni tra le diverse componenti software) e il test di sistema (si verifica il comportamento del sistema nel suo complesso).

Capitolo 3

Analisi statica e SonarQube

3.1 Analisi statica

Il controllo o analisi statica è una tecnica che consente di analizzare il codice di un programma attraverso un processo di valutazione basato sulla struttura, forma e contenuto. Non richiede dunque l'esecuzione del programma che si sta esaminando. Linguaggi diversi presentano problematiche differenti. Si concentra su diversi aspetti:

- Flusso di controllo: si analizzano le sequenze di esecuzione possibili al fine di accertare che il codice sia ben strutturato e localizzare blocchi di codice non raggiungibile;
- Flusso dei dati: si analizza l'uso e l'evoluzione di variabili e costanti, rileva anomalie quali l'utilizzo di variabili non inizializzate o la presenza di scritture successive senza letture intermedie;
- Esecuzione simbolica: è un metodo di analisi in cui si assume di avere degli input simbolici e si procede con l'esecuzione del programma in cui non vengono elaborati valori ma formule, ottenendo un execution tree che contiene le informazioni su tutti i possibili esiti;
- Verifica formale del codice: la prova della correttezza del codice rispetto alla specifica con l'ausilio di strumenti matematici, per avere una prova di correttezza totale bisogna provare anche la terminazione;

- Verifica di limite: vengono analizzati dati del programma in relazione al loro tipo e precisione, controllando casi di overflow, underflow ed errori di arrotondamento e facendo range checking;
- Inferenza: si verifica l'assenza di side-effect tra parti isolate del sistema;
- Codice oggetto: si verifica la giusta conversione da codice sorgente a codice oggetto e che non siano stati introdotti errori durante la compilazione.

Il modo più informale di effettuare questo tipo di analisi, e generalmente il primo tipo di analisi svolta, è il desk check o code reading. La tecnica permette di riscontrare difetti come loop infiniti, incoerenza tra parametri formali e la chiamata dei sottoprogrammi, incoerenza tra tipi di dati, accesso incorretto a strutture dati. È possibile che venga eseguita un'esecuzione simulata del codice volta a individuare errori algoritmici (walkthrough). È possibile eseguire una verifica statica usando strumenti automatici, che permettono di ridurre tempi e costi di testing.

3.2 SonarQube

SonarQube è uno strumento di analisi statica del codice utile per identificare vulnerabilità, code smell e bug. L'idea principale è avere uno strumento per mantenere traccia dei progressi e dei risultati rispetto alle metriche di qualità al fine di ridurre la crescita del debito tecnico, soprattutto in contesti di sviluppo con numerosi team coinvolti.
SonarQube nasce come la fusione di due tool di analisi del codice: Check Style e PMD [9]. Può essere usato dalla piattaforma `sonarcloud.io` o scaricato ed essere usato su un server privato.
Utilizza il modello SQALE[1] (Software Quality Assessment based on Lifecycle Expectations) un modello per la valutazione della qualità che permette di definire cosa crea debito tecnico, stimarlo correttamente e offrire strategie basate sulla priorità per stabilire un piano di riborso del debito tecnico.

[1] http://www.sqale.org/details

SonarQube è una piattaforma open source largamente utilizzata che supporta ventisette linguaggi e supporta automaticamente l'integrazione di Git e Apache Subversion e attraverso plugin addizionali sono supportati anche altri SCM provider. Dalla versione di SonarQube 8.0 è stata aggiunta l'integrazione di GitLab[2]. Essendo scritto in Java, SonarQube necessita di una Java Virtual Machine per essere lanciato.
La piattaforma si basa su quattro componenti[3]:

1. Un SonarQube server che lancia tre processi: un server web per accedere ai risultati dei controlli precedenti e per la configurazione dell'istanza di SonarQube, un motore di ricerca basato su Elasticsearch[4] e un motore per la creazione e il salvataggio dei report delle analisi;

2. Un database in cui vengono salvati i dati relativi all'istanza di SonarQube e i dati delle analisi;

3. Numerosi plugin installati sul server per i diversi linguaggi, per l'autenticazione e per l'integrazione di SCM;

4. Uno o più SonarScanner per analizzare i progetti.

SonarQube, per analizzare il codice, usa delle regole (rules) classificate in quattro tipi: code smell, bug, vulnerabilità e security hotspot.
Nella categoria *code smell* rientrano le caratteristiche che indicano un difetto di programmazione ma non rivelano degli errori e dunque non influiscono sull'effettiva correttezza del software, ma rendono il codice meno manutenibile. Degli esempi tipici sono l'utilizzo di *goto*, *switch* senza la clausola $default$, sottoprogrammi con troppi parametri, codice duplicato, classi troppo ampie, codice indentato male, nomi di variabili e attributi non chiari.
I *bug* sono errori da cui dipende un comportamento inaspettato o incorretto del programma, riducono l'affidabilità del programma. Tra i bug rientrano una cattiva gestione dei puntatori, il numero di parametri attuali diversi dal numero dei parametri formali, divisione per zero, classi senza un costruttore

[2]https://www.sonarqube.org/sonarqube-8-0/
[3]https://docs.sonarqube.org/latest/architecture/architecture-integration
[4]https://www.elastic.co/elasticsearch

o con il costruttore privato, confronti (<, >, <=, >=) tra booleani.
Le *vulnerabilità* sono delle debolezze che possono essere sfruttate per compromettere il livello di sicurezza del sistema. Le vulnerabilità dipendono spesso da chiavi crittografiche poco robuste (ad esempio una chiave per l'algoritmo RSA dovrebbe essere lunga almeno 2048 bit), da funzioni hash che non includono salt o includono salt troppo corti o non randomici (e quindi soggette facilmente ad attacchi a dizionario) e da protocolli che sono considerati come insicuri come il protocollo SSLv3.
I *security hotspot* sono delle debolezze che non sono problematiche, ma potrebbero diventare delle vulnerabilità. Degli esempi sono una cattiva configurazione dei cookie, l'utilizzo di algoritmi crittografici non standard e l'utilizzo di protocolli che non costruiscono connessioni sicure.
[5] segnala che per i code smell e i bug si aspettano zero falsi positivi, invece per le vulnerabilità e i security hotspot si aspettano meno del 20% di falsi positivi.
È possibile creare nuove regole attraverso la piattaforma di SonarQube, usando i template forniti. Attraverso l'uso dei profili di qualità è possibile scegliere un insieme di regole da associare a ogni linguaggio.
Durante l'analisi, vengono definiti degli issues ogni qualvolta un frammento di codice infrange una regola. A ogni issue è legato un livello diverso di gravità:

- *Blocker*: un bug che può modificare il comportamento del programma. Il codice è da modificare;
- *Critical*: un bug che poco probabilmente può modificare il comportamento del programma o una falla di sicurezza. Il codice è da revisionare;
- *Major*: difetti del codice che abbassano la produttività degli sviluppatori come codice ripetuto, parametri e variabili non utilizzate;
- *Minor*: difetti del codice che riducono lievemente la produttività degli sviluppatori come linee troppo lunghe e switch con meno di tre casi;

[5]https://docs.sonarqube.org/latest/user-guide/rules

- *Info*: non rappresenta un vero difetto del software, come i commenti che contendono “TODO” o l’uso di costrutti deprecati.

Un *issue*, una volta creato, può assumere cinque stati: subito dopo la sua creazione è aperto (*open*), diventa confermato (*confirmed*) quando l’utente indica che è un problema valido, risolto (*resolved*) quando l’utente indica che nella prossima analisi non dovrebbe essere considerato nuovamente perché sono state apportate modifiche, riaperto (*reopened*) quando il problema è stato indicato come risolto ma non è stato veramente corretto e chiuso (*closed*) quando SonarQube non lo riconosce più come un problema.
Un problema viene chiuso quando è stato corretto (*fixed*) o perché la relativa regola che lo riconosce non è più disponibile (*removed*).
Le metriche utilizzate per definire la qualità del codice sono la complessità,, la duplicazione, la manutenibilità, l’affidabilità, la sicurezza, la dimensione e la copertura dei test.
Alle metriche viene assegnata una valutazione (*rating*) come segue.
Rating affidabilità:
A = 0 bug
B = almeno un minor bug
C = almeno un major bug
D = almeno un critical bug
E = almeno un blocker bug

Rating sicurezza:
A = 0 vulnerabilità
B = almeno una minor vulnerabilità
C = almeno una major vulnerabilità
D = almeno una critical vulnerabilità
E = almeno una blocker vulnerabilità

Rating debito tecnico:
dipende dal "technical debt ratio" (TD ratio) che è il rapporto tra il debito tecnico e il tempo stimato dello sviluppo, calcolato come $LOCx30min$.
A = TD ratio <= 5%
B = TD ratio tra il 6 e il 10%

C = TD ratio tra il 10 e il 20%
D = TD ratio tra il 21 e il 50%
E = TD ratio >= 50%

I linguaggi di programmazione coperti dalla piattaforma sono: Java, T-SQL, ABAP, PL/SQL, Swift, Apex, COBOL, TypeScript, Kotlin, Ruby, CSS, XML, JavaScript, C#, VB.Net, C, C++, RPG, VB6, Objective-C, PL/I.

Capitolo 4

Analisi di un progetto con SonarQube

4.1 Installazione e avvio di SonarQube

Sulla macchina in cui viene installato SonarQube deve essere presente una Java Virtual Machine. Le analisi svolte sono state condotte su Windows 10 usando SonarQube 7.9 e SonarQube 8.1. Entrambe le versioni richiedono Oracle JRE 11 o OpenJDK 11. È stata usata l'edizione Community e SonarScanner 4.2.
L'installazione prevede il download della cartella compressa dal sito `https://www.sonarqube.org/downloads/`. Dopo aver estratto i file, bisogna lanciare l'eseguibile per l'installazione che si trova nella cartella bin/[OS].

Per quanto riguarda l'installazione su windows, va lanciato il file batch InstallNTService.bat che si trova nella sottocartella windows-x86-64.
Nella cartella conf si trova il file sonar.properties, che va modificato nel caso in cui bisogna configurare impostazioni valide per il programma come l'utilizzo di un server esterno. SonarQube possiede un database integrato che però non è scalabile e non è supporta l'upgrade di SonarQube. Sono supportati database esterni come PostgreSQL, Microsoft SQL Server e il database di Oracle.
Per poter utilizzare SonarScanner va scaricato il file compresso ed estrat-

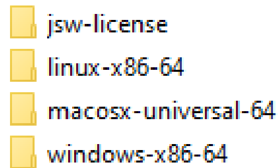

Figura 4.1: Cartella bin

to. È consigliato creare la variabile d'ambiente del sistema con nome SONAR_SCANNER_HOME e con valore la cartella in cui è stato salvato il file SonarScanner.
È possibile installare nuovi plugin scaricando il jar del plugin e spostandolo nella sottocartella extension \plugins.
Per le analisi svolte ho utilizzato, oltre ai plugin installati di default da SonarQube, i plugin C (Community) e C++ (Community), reperibili all'indirizzo `https://github.com/SonarOpenCommunity/sonar-cxx`. Il download di questi plugin è stato necessario perché i linguaggi C e C++ non sono inclusi tra i linguaggi della Community edition di SonarQube.

4.2 Analisi tramite SonarQube

Una volta conclusa l'installazione, va avviato il server di SonarQube attraverso l'istruzione StartSonar.bat presente in bin/windows-x86-64. È quindi possibile eseguire l'accesso all'indirizzo http://localhost:9000/.
Fatto il log in, viene mostrata la dashboard di SonarQube in cui è possibile creare un nuovo progetto, per cui va definita una Project Key univoca.

sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration

Create new project

Project key*

Up to 400 characters. All letters, digits, dash, underscore, period or colon.

Display name*

Up to 255 characters

Set Up

Figura 4.2: Nuovo progetto su SonarQube

Per poter essere scansionato, e quindi analizzato, va creato un file sonar-project.properties da posizionare nella cartella principale del progetto in cui vanno settati dei parametri obbligatori (Tabella 4.1) e opzionali (Tabella 4.2).

Tabella 4.1: Parametri obbligatori

Key	Descrizione	Default
sonar.host.url	L'URL del server	http://localhost:9000
sonar.projectKey	La chiave univoca inserita durante la creazione del progetto su SonarQube.	

Tabella 4.2: Parametri opzionali

Key	Descrizione	Default
sonar.projectName	Il nome del progetto mostrato dall'interfaccia di SonarQube	La projectKey
sonar.projectVersion	La versione del progetto	"not provided"
sonar.login	Il token di autenticazione dell'utente	
sonar.login	La password associata al'utente indicato da sonar.login	
sonar.ws.timeout	Il tempo massimo di attesa per la risposta di una chiamata al server in secondi.	60
sonar.projectDescription	La descrizione del progetto	
sonar.link.homepage	L'homepage del progetto	
sonar.links.scm	Il repository dei sorgenti del progetto	
sonar.sources	I percorsi che contengono i codici sorgente	
sonar.tests	I percorsi che contengono i sorgenti dei test	
sonar.sourceEncoding	La codifica dei file. La lista delle codifiche disponibili è dipendente dalla JVM usata	La codifica del sistema
sonar.externalIssuesReportPaths	I percorsi dei report	

Tabella 4.2: Parametri opzionali

Key	Descrizione	Default
sonar.projectDate	Assegnazione di una data all'analisi. Utile per creare una cronologia di analisi per progetti mai analizzati. Il formato è yyyy-mm-dd	La data corrente
sonar.scm.provider	Esplicitare quale SCM plugin usare per il progetto. Il valore di questa proprietà deve essere scritto solo in minuscolo.	
sonar.scm.forceReloadAll	Questa proprietà va impostata a true se è necessario caricare nuovamente anche i file che non hanno subito modifiche dall'ultima analisi.	Analisi dei soli file modificati dall'ultima analisi.
sonar.cpd.$language.minimumtokens	Il numero di caratteri consecutivi dopo i quali si considera il frammento di codice duplicato	100
sonar.cpd.$language.minimumLines	Il numero di linee consecutive dopo le quali si considera il blocco di codice duplicato	10

Tabella 4.2: Parametri opzionali

Key	Descrizione	Default
sonar.log.level	Il livello di informazioni nei log prodotte durante l'analisi. I valori possibili per questa proprietà sono: INFO, DEBUG, TRACE	INFO
sonar.verbose	Aggiunta di dettagli sia client-side sia server-side.	False

4.3 Progetti analizzati

Selezione dei progetti

Sono stati scelti nove progetti open source da analizzare (Tabella 4.3). Sette sono dei motori per il gioco degli scacchi, scritti in linguaggi diversi e con un numero di linee di codice variabile, il più breve conta circa 850 linee di codice, il più ampio più di 11 000. A questi si sommano la release 7.9 di SonarQube e la release 8.1 di SonarQube, scritte in Java, che sono state prese in considerazione perché si tratta di progetti molto grandi con molti contributori.
La dimensione è misurata in righe di codice (Lines Of Code), SonarQube considera come linea di codice una riga che ha almeno un carattere diverso da spazio, tabulazione e non sia commentata. Questa misura varia in base al linguaggio e allo stile di programmazione, ma risulta un buon compromesso perché facile da calcolare e intuitiva.

Tabella 4.3: Progetti analizzati

Progetto	Linguaggio	Dimensione (LOC)
Belofte[1]	C++, HTML, XML	6,896
Feeks[2]	Python	950
Gopher-Check[3]	GO	5,007
Python-Chess[4]	Python	866
RapChess[5]	Javascript	2,340
Roller[6]	JavaScript	11,431
ScalaChess[7]	Scala	1,354
SonarQube 7.9[8]	Java	537,556
SonarQube 8.1[9]	Java	502,048

Dopo le analisi la dashboard di SonarQube si popolerà con una panoramica dei dati risultanti (Figura 4.3).
Nella sezione di sinistra della dashboard sono presenti le categorie per cui è possibile filtrare i progetti. È possibile filtrare i progetti per i punteggi ottenuti per *quality gate* (progetti bocciati o approvati), affidabilità, sicurezza, manutenibilità, copertura (in riferimento alla percentuale coperta dai test), duplicazione, dimensione, linguaggio e tag (che l'amministatore può aggiungere ai singoli progetti).
Nella panoramica, per singolo ogni progetto viene mostrata una sintesi dei risultati per alcune metriche: numero dei bug, numero delle vulnerabilità, numero dei code smell, la percentuale della copertura dei test e la densità della duplicazione del codice

[1]https://sourceforge.net/p/belofte/gitrepo/ci/master/tree/dist/Release/
[2]https://github.com/flok99/feeks
[3]https://github.com/stephenjlovell/gopher_check/releases
[4]https://travis-ci.org/jrialland/python-chess
[5]https://github.com/Thibor/Chess-Engine-Rapchess/releases
[6]https://github.com/op12no2/roller
[7]https://github.com/exoticorn/scalachess/
[8]https://github.com/SonarSource/sonarqube/tree/branch-7.9
[9]https://github.com/SonarSource/sonarqube

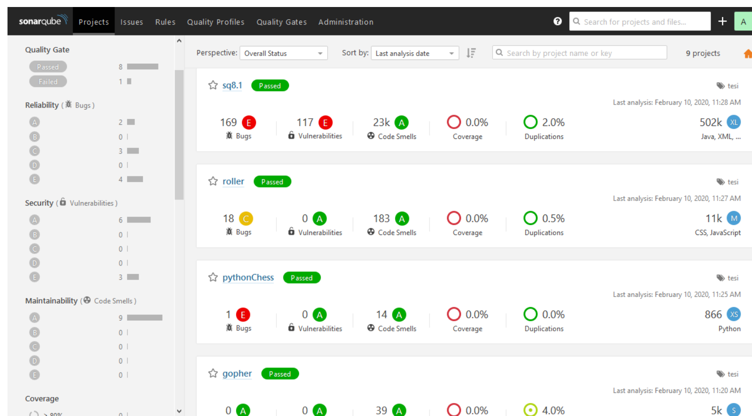

Figura 4.3: Dashboard SonarQube

4.4 Risultati e conclusioni

Nella Tabella 4.4 sono riportati i dati relativi all'affidabilità (numero di bug, il rating e lo sforzo per rimediare), nella Tabella 4.5 i dati relativi alla sicurezza (numero di vulnerabilità, il rating, lo sforzo stimato per raggiungere il rating A e il numero di security hotspot), nella Tabella 4.6 i dati relativi alla manutenibilità (il numero di code smell, il rating, il debito tecnico, il "technical debt ratio"), nella Tabella 4.7 i dati relativi alla duplicazione (la densità che è il rapporto tra le linee duplicate e le linee totali in percentuale e il numero di blocchi, righe e file ripetuti) e nella Tabella 4.8 i dati relativi alla dimensione in LOC, la percentuale dei commenti, la complessità ciclomatica e il numero di issue riportati.

Tabella 4.4: Reliability

Progetto	Bug	Rating	Effort
Belofte	23	C	1h 4min
Feeks	4	E	40min
Gopher-Check	0	A	0
Python-Chess	1	E	10min

RapChess	6	C	12min
Roller	18	C	3h
ScalaChess	0	A	0
SonarQube 7.9	180	E	3d 4h
SonarQube 8.1	169	E	3d 2h

Tabella 4.5: Security

Progetto	Vulnerability	Rating	Effort	Security hotspot
Belofte	6	E	6min	0
Feeks	0	A	0	5
Gopher-Check	0	A	0	0
Python-Chess	0	A	0	4
RapChess	0	C	0	2
Roller	0	A	0	23
ScalaChess	0	A	0	0
SonarQube 7.9	117	E	3d 6h	681
SonarQube 8.1	117	E	3d 6h	655

Tabella 4.6: Maintainability

Progetto	Code smell	Rating	Technical Debt	TC ratio
Belofte	2	A	10min	0,00%
Feeks	30	A	6h 7min	1,3%
Gopher-Check	39	A	1d 6h	0,60%
Python-Chess	14	A	3h 26min	0,00%
RapChess	89	A	3d	0,00%
Roller	183	A	5d 1h	0,70%

ScalaChess	11	A	2h 2min	0,30%
SonarQube 7.9	25740	A	430d	1,30%
SonarQube 8.1	23116	A	381d	1,30%

Tabella 4.7: Duplication

Progetto	Density	Block	Line	File
Belofte	3,80%	13	413	4
Feeks	0,00%	0	0	0
Gopher-Check	4,00%	12	259	1
Python-Chess	0,00%	0	0	0
RapChess	56,80%	42	1383	3
Roller	0,50%	4	76	1
ScalaChess	0,00%	0	0	0
SonarQube 7.9	1,90%	1114	19520	541
SonarQube 8.1	2,00%	1055	19070	493

Tabella 4.8: Dimensione, Complessità, Issue

Progetto	LOC	Commenti	Compl. Ciclomatica	Issue
Belofte	6896	15,1%	1352	31
Feeks	950	3,8%	269	34
Gopher-Check	5007	17,7%	1147	39
Python-Chess	866	9,6%	241	15
RapChess	2340	2,1%	656	95
Roller	11431	9,7%	1604	201
ScalaChess	1354	0,1%	377	11
SonarQube 7.9	537556	4.7%	67064	26037
SonarQube 8.1	502048	4.9%	64465	23,402

Escludendo i due software con zero bug (*Gopher-Check e Scala-Chess*), evidentemente di alta qualità, il numero di bug è proporzionale alla dimen-

sione del codice. Il rating, come visto nel par. 3.2, dipende dalla gravità dei bug e per questa ragione i progetti *Python-Chess* e *Feeks* nonostante i pochi bug hanno una valutazione peggiore rispetto agli altri progetti con più bug e che prevedono uno sforzo maggiore per essere corretti.
Il trend è lo stesso anche per quanto riguarda l'aspetto della sicurezza e il numero delle vulnerabilità.
Il progetto che ha ottenuto la valutazione più bassa è *Belofte*, per cui sono state rilevate sei vulnerabilità "bloccanti".
Il numero di code smell non è proporzionale alla dimensione, ma comunque nei progetti *SonarQube 7.9* e *SonarQube 8.1*, molto più grandi degli altri sono più numerosi.
Il progetto con la percentuale maggiore di codice ripetuto è *RapChess*. Il problema è da imputare a due file, "rapshort.js" e "rapspeed.js" in cui vengono ripetute le dichiarazione delle variabili e delle funzioni relative alla scacchiera. Future modifiche saranno più costose.

Nella Figura 4.4 ci sono alcuni esempi di bug trovati nei progetti analizzati ordinati per gravità (blocker, critical, major, minor):

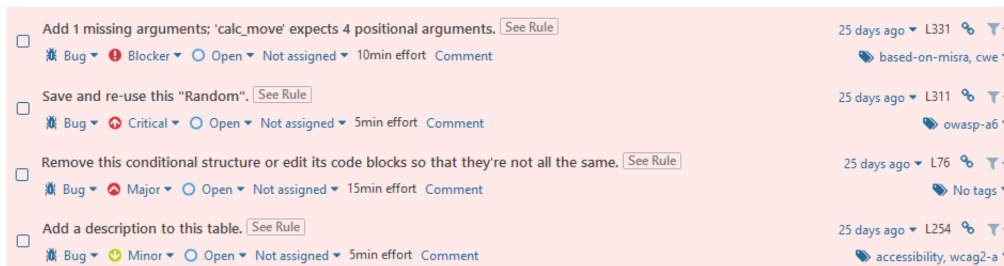

Figura 4.4: Esempi di bug

Il bug contrassegnato come *blocker* è relativo a una chiamata di funzione che ha un numero diverso di argomenti rispetto a quelli aspettati. Il bug contrassegnato come *critical* suggerisce di riusare un oggetto "Random", salvarlo e riusarlo per una migliore efficienza. Il bug contrassegnato come *major* evidenzia che in una struttura condizionale (if o switch) tutti i rami

hanno la stessa implementazione, quindi o è stato commesso un errore o non ci dovrebbe essere un comando condizionale. Il bug contrassegnato come *minor* segnala la mancanza di descrizione in una tabella, questa assenza diminuisce l'accessibilità.
Nella Figura 4.5 ci sono alcuni esempi di vulnerabilità trovate nei progetti analizzati ordinati per gravità (blocker, critical, major, minor):

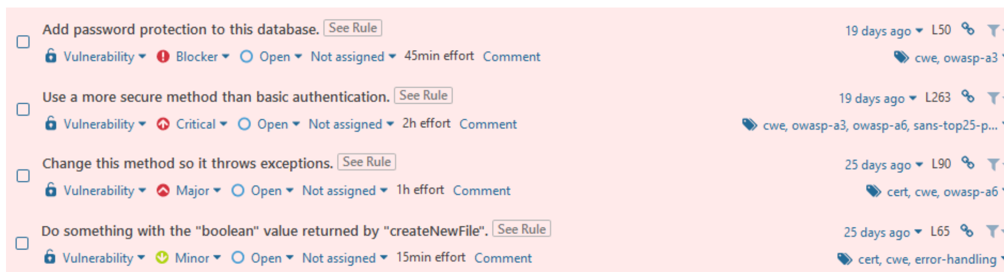

Figura 4.5: Esempi di vulnerabilità

La vulnerabilità contrassegnata come *blocker* segnala che un database non è protetto da password. La vulnerabilità contrassegnata come *critical* evidenzia un tipo di protezione basso. La vulnerabilità contrassegnata come *major* indica la mancata implementazione di eccezioni che può portare ad attacchi di tipo man-in-the-middle perchè consente la connessione di host non certificati. La vulnerabilità contrassegnata come *minor* segnala un non uso del booleano restituito da una funzione.
Nella Figura 4.6 ci sono alcuni esempi di code smell trovati nei progetti analizzati ordinati per gravità (blocker, critical, major, minor, info):

Figura 4.6: Esempi di code smell

Il code smell contrassegnato come *blocker* indica un metodo che ritorna una costante e questo o è un sintomo di un cattivo design o un errore che può portare a bug. Il code smell contrassegnato come *critical* segnala l'assenza di indentazione o di parentesi graffe successivamente a un comando condizionale, questo può indicare un comportamento sbagliato del programma e crea confusione nel lettore del codice. Il code smell contrassegnato come *major* mostra l'utilizzo di un costrutto deprecato. Il code smell contrassegnato come *minor* indica uno switch con meno di tre casi, ma è consigliato usare un if in questa situazione. Il code smell contrassegnato come *info* evidenzia un commento in cui è usata l'espressione *TODO* che di solito è presente per indicare porzioni di codice in cui vanno eseguite delle modifiche o aggiunte. La presenza della regola che segnala l'uso di *TODO* è utile per tracciare queste porzioni di codice ed è eventualmente segnalare i casi in cui il codice è stato prodotto ma non è stato eliminato il commento.
Nella Figura 4.7 ci sono alcuni esempi di security hotspot trovati nei progetti analizzati:

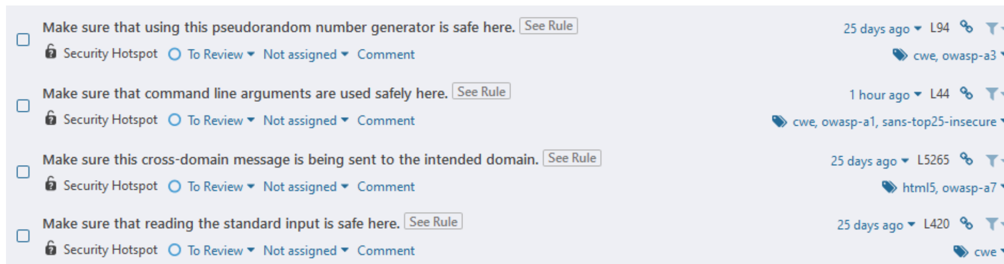

Figura 4.7: Esempi di security hotspots

La Figura 4.8 e la Figura 4.9 mostrano rispettivamente i risultati delle scansioni del codice sorgente di SonarQube 7.9 e di SonanQube 8.1. Si può notare che da una versione a quella successiva sono diminuiti i bug di circa il 6%, i security hotspot di circa il 4%, il debito tecnico del 12% e il numero di code smell del 12%. Il numero di vulnerabilità non è variato. Il numero di linee di codice è diminuito di circa il 7%.

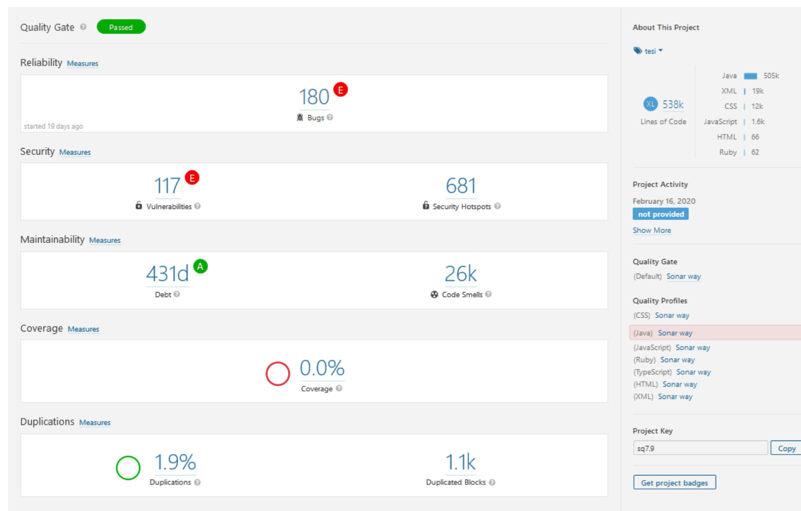

Figura 4.8: Risultati analisi SonarQube 7.9

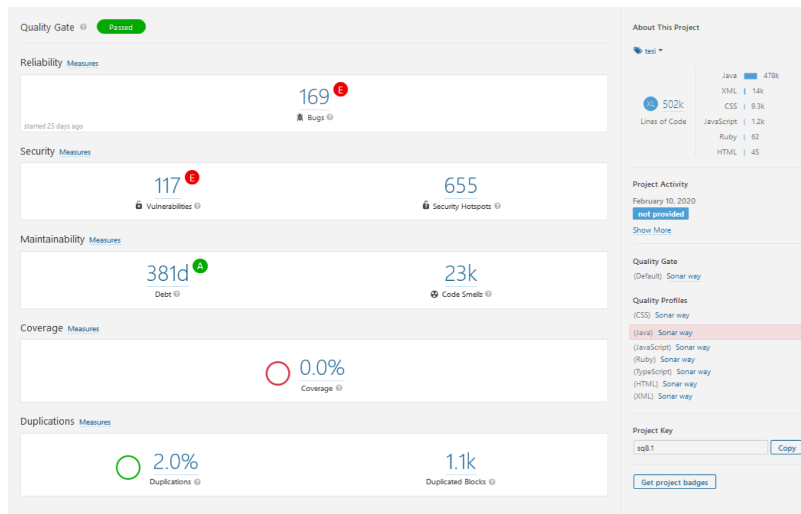

Figura 4.9: Risultati analisi SonarQube 8.1

4.5 Gli studenti e SonarQube

Gli studenti che imparano a programmare, quando scrivono codice funzionante ne sono solitamente soddisfatti, però non considerano la qualità del codice e spesso non sono nemmeno consapevoli degli errori commessi, soprattutto in riferimento agli attributi interni di qualità come la leggibilità, la comprensibilità e la manutenibilità del codice. Sarebbe una richiesta utopica pretendere dagli studenti alti livelli di qualità dall'inizio del percorso di studi, il problema è che con il passare dei semestri la qualità rimane la stessa o peggiora perché i progetti assegnati agli studenti diventano più grandi e complessi [10].
La mancanza di interesse per la tematica della qualità è causa di un divario tra le abilità dei neolaureati e le richieste del mondo del lavoro [11]. Gli studenti o i neolaureati non sono impreparati solo riguardo a tecniche di programmazione, ma anche relativamente l'uso di strumenti software utili nei processi di sviluppo usati in contesti aziendali.
La carenza di attenzione in questo tema può svantaggiare gli studenti che cercano lavoro e può influenzare la produttività e il benessere dei nuovi

assunti. Inoltre è importante che gli studenti si abituino a scrivere non solo codice che funzioni correttamente, ma che si adegui agli standard internazionali di qualità.
I sistemi odierni non vengono sviluppati da programmatori che lavorano in maniera isolata, quindi gli studenti dovrebbero familiarizzare con strumenti che possano permettere la collaborazione e la coordinazione.
La continua integrazione unita alla continua ispezione del codice è un metodo di sviluppo che permette di individuare gli errori più comuni e le cattive abitudini di programmazione [12].
In [13] vengono presentati i risultati di uno studio condotto su studenti che hanno utilizzato SonarLint (un plugin per Eclipse basato sullo stesso analizzatore di software di SonarQube) ed è stato osservato che:

- Gli errori più comuni sono stati quelli di sintassi e una errata gestione dei log;
- Gli studenti velocemente si sono adattati ai suggerimenti;
- Quando gli studenti si sono trovati in disaccordo con il tool, dopo aver letto la spiegazione fornita si sono trovati in accordo con SonarLint.

I risultati hanno inoltre suggerito che il vantaggio maggiore è stato l'uniformazione del codice il quale risulta più facile e comprensibile da leggere.
In [14] viene esposto un metodo basato sul modello dell'ispezione continua usato dalla piattaforma GitHub. Si pone l'enfasi su un controllo continuo della qualità, operato attraverso l'uso di strumenti di analisi statica. Inoltre la continua ispezione (e le conseguenti modifiche) permettono di imparare e memorizzare meglio i pattern suggeriti per aumentare la qualità [12].

Capitolo 5

Conclusioni

Il software permea molti ambiti del nostro quotidiano e in generale delle organizzazioni e della società di cui facciamo parte. Nonostante la sua importanza è facilmente soggetto a malfunzionamenti che sono dovuti a errori o vulnerabilità presenti nel codice. Inoltre il software è costoso non solo da sviluppare, ma anche da manutenere. Per queste ragioni è fondamentale concentrarsi e investire sulla qualità.
Nel primo capitolo sono stati introdotti i concetti che poi sono stati esaminati nei capitoli successivi.
Nel secondo capitolo viene data la definizione di qualità e delle sue metriche, di debito tecnico e fornite delle tecniche per mantenere alta la qualità.
Nel terzo capitolo è stato presentato uno strumento per l'analisi statica del codice, SonarQube.
Il quarto capitolo è riguardo all'utilizzo di SonarQube e un esperimento svolto su nove software open source, di diversa dimensione e scritti in linguaggi diversi.
Per questo argomento, come possibilità di sviluppo futuro c'è l'idea di verificare come la qualità cambia a seconda del metodo di sviluppo utilizzato.
Un'altra prospettiva interessante è quella di verificare come gli studenti di corsi di studio universitari come informatica o ingegneria informatica possano integrare l'utilizzo di SonarQube alla pratica della programmazione sistematicamente e quindi imparare a programmare ponendo da subito attenzione alla tematica della qualità del software.

Questa pagina è lasciata intenzionalmente bianca.

Bibliografia

[1] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, “Continuous code quality: Are we (really) doing that?” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 790–795. [Online]. Available: https://doi.org/10.1145/3238147.3240729

[2] K. M. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. J. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, “Manifesto for agile software development,” 2013.

[3] C. Jones, *The Economics of Software Quality*. Addison-Wesley Professional, 2011.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162),” *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6693

[5] E. Tom, A. Aurum, and R. Vidgen, “An exploration of technical debt,” *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[6] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet, “What is social debt in software engineering?” in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2013, pp. 93–96.

[7] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd workshop on managing technical debt*, 2011, pp. 1–8.

[8] J. P. Miguel, D. Mauricio, and G. Rodríguez, "A review of software quality models for the evaluation of software products," *arXiv preprint arXiv:1412.2977*, 2014.

[9] P. Quezada Sarmiento, D. Guaman, L. R. Barba Guamán, L. Enciso, and P. Cabrera, "Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis," 07 2017.

[10] D. M. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 13–17.

[11] A. Radermacher, G. Walia, and D. Knudson, "Investigating the skill gap between graduating students and industry expectations," in *Companion Proceedings of the 36th international conference on software engineering*, 2014, pp. 291–300.

[12] Y. Lu, X. Mao, T. Wang, G. Yin, and Z. Li, "Improving students' programming quality with the continuous inspection process: a social coding perspective," *Frontiers of Computer Science*, vol. 14, no. 5, p. 145205, 2019.

[13] J. Berg and E. Gralén, "The effects of continuous code inspection on code quality," 2019.

[14] Y. Lu, X. Mao, T. Wang, G. Yin, Z. Li, and H. Wang, "Poster: Continuous inspection in the classroom: Improving students' programming quality with social coding methods," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 141–142.

Questa pagina è lasciata intenzionalmente bianca.

Elenco delle figure

2.1 Diagramma ISO 9126 11

4.1 Cartella bin 22
4.2 Nuovo progetto su SonarQube 23
4.3 Dashboard SonarQube 28
4.4 Esempi di bug 31
4.5 Esempi di vulnerabilità 32
4.6 Esempi di code smell 33
4.7 Esempi di security hotspot 34
4.8 Risultati analisi SonarQube 7.9 34
4.9 Risultati analisi SonarQube 8.1 35

Elenco delle tabelle

4.1 Parametri obbligatori 23
4.2 Parametri opzionali 24
4.3 Progetti analizzati 27
4.4 Reliability 28
4.5 Security 29
4.6 Maintainability 29
4.7 Duplication 30
4.8 Dimensione, Complessità, Issue 30