UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

STRUMENTI PER LA SICUREZZA DEL CODICE: ANALISI STATICA

Relatore: Prof. / Dott. Migliardi Mauro

Laureando/a: Lorenzato Gian Battista

Correlatore: Luca Bianconi

ANNO ACCADEMICO: 2021 – 2022

Data di laurea: 20/07/2022

Indice

1 Introduzione 7

1.1 Struttura del Documento 7

1.2 Scopo della tesi.....8

2 Analisi del Codice.....9

2.1 Analisi Statica.....10

2.1.1 Come viene eseguita l'analisi statica? 11

2.1.2 Tipi di analisi statica 12

2.1.3 Vantaggi e svantaggi dell'analisi statica 12

2.1.4 Verifica statica vs. verifica dinamica 13

2.2 Analisi Dinamica.....15

3 Strumenti e fornitori di analisi statica 15

3.1 Livelli di gravità degli analizzatori 18

3.2 SonarQube.....20

3.2.1 Scrivere codice pulito e sicuro 21

3.2.2 Gli studenti e SonarQube 21

4 Esempi d'uso di SonarQube.....23

4.1 introduzione generale alle prove fatte e spiegazione obiettivi.....23

4.2 metodologia adottata per i test fatti con SonarQube 23

4.3 Report test 25

5 Conclusioni 29

Bibliografia 30

Elenco figure e tabelle

1 Flowchart di un software che incorpora l'analisi statica.....14

2 Fasi dello sviluppo di un applicativo nelle quali opera l'analisi del codice 14

3 Embold.....16

4 Kiuwan 17

5 PyCharm 17

6 Generiche problematiche del codice segnalate da un analizzatore 18

7 Finestra di elenco degli errori 19

8 Esempio di segnalazione grafica di criticità del codice sorgente.....19

9 Sonarqube, Sonarlint e Sonarcloud.....20

10 Schermata iniziale Server SonarQube.....24

11 Esempio di schermata principale del Server SonarQube in seguito alla selezione di un progetto 25

12 Esempio di bug segnalato da SonarQube con annessa spiegazione dell'errore riscontrato 26

13 Suggerimento indicato da SonarQube per risolvere il bug evidenziato 26

14 Esempio di code smell segnalato da SonarQube con annessa spiegazione dei possibili problemi che potrebbe comportare 27

15 Suggerimento indicato da SonarQube per risolvere il code smell evidenziato 28

Glossario dei termini tecnici

Concetto	Definizione
Analizzatore	Un'applicazione client che analizza il codice sorgente per calcolare gli **snapshot**
Backdoor	Metodo, spesso segreto, per aggirare la normale autenticazione in un sistema informatico, un **crittosistema** o un algoritmo. Le backdoor sono spesso scritte in diversi linguaggi di programmazione e hanno la funzione principale di superare le difese imposte da un sistema, al fine di accedere in remoto a un personal computer, ottenendo un'autenticazione che permetta di prendere il completo o parziale possesso del computer vittima.
Banca dati	Memorizza la configurazione e gli **snapshot**
Bug	Un problema che rappresenta qualcosa di sbagliato nel codice. Se questo non si è ancora rotto, lo farà, e probabilmente nel peggior momento possibile. Questo deve essere risolto. Ieri.
Bug report	Rapporto che elenca gli errori rilevati nel software per evidenziare esattamente ciò che è considerato sbagliato e include anche dettagli su come affrontare ciascun **problema**.
Crittosistema	Un insieme di algoritmi crittografici che realizzano un particolare servizio di sicurezza, tipicamente allo scopo di raggiungere confidenzialità tramite cifratura.
Code smell	Un problema relativo alla manutenibilità nel codice. Lasciarlo così com'è significa che nella migliore delle ipotesi i manutentori avranno più difficoltà di quanto dovrebbero nell'apportare modifiche al codice. Nel peggiore dei casi, saranno così confusi dallo stato del codice che introdurranno ulteriori errori man mano che apportano modifiche.
Costo della riparazione	Il tempo stimato necessario per risolvere i problemi di **vulnerabilità** e affidabilità.
Debito tecnico	Il tempo stimato necessario per risolvere tutti i problemi di manutenibilità/**code smells**
Hotspot di sicurezza	Parte di codice sensibile alla sicurezza, tuttavia la sicurezza del software potrebbe anche non esserne influenzata. È responsabilità dello

	sviluppatore rivedere il codice per determinare se è necessaria, o meno, una modifica.
Metrica	Un tipo di misura. Le metriche possono avere valori o **misure** variabili nel tempo. Esempi di metrica: *numero di righe di codice*, *complessità*, ecc. Una metrica può essere *qualitativa* (fornisce un'indicazione di qualità sul componente, come la densità delle righe duplicate, la copertura della riga mediante test, ecc.) o *quantitativa* (non fornisce un'indicazione di qualità sul componente, come il numero di righe di codice, complessità, ecc.)
Misurare	Il valore di una **metrica** per un determinato file o progetto in un determinato momento. Ad esempio, *"125 righe di codice sulla classe MyClass" o "densità di righe duplicate del 30,5% sul progetto MyProject"*
Problema	Quando una parte di codice non è conforme a una regola, viene registrato un problema sullo **snapshot**. Un problema può essere registrato su un file di origine o su un file di unit test. Esistono 3 tipi di problemi: **bug**, **code smell** e **vulnerabilità**
Quality gate	Un insieme di **regole** le quali, proprio in virtù di questa loro inclusione, forniranno un controllo mirato dell'analizzatore su più specifiche caratteristiche del software simultaneamente in base ai vincoli considerati. Ogni **snapshot** si basa su un singolo Quality gate.
Regola	Una pratica di codifica che dovrebbe essere seguita al fine di ottenere del codice che rispetti determinati standard di qualità prestabiliti. Il mancato rispetto delle regole di codifica porta a **bug**, **vulnerabilità**, **hotspot di sicurezza** e **code smells**. Le regole possono controllare la qualità dei file sorgenti o dei file di unit test.
Snapshot	Un insieme di **misure** e **problemi** su un dato progetto in un dato momento. Per ogni analisi viene generato uno snapshot.
Vulnerabilità	Un problema relativo alla sicurezza che rappresenta una **backdoor** per gli aggressori.

Capitolo 1

Introduzione

Nello sviluppo di prodotti software di tipo industriale il rispetto di determinati standard, innanzitutto di sicurezza, è essenziale per garantire non solo la qualità del software ma anche per fare in modo che quest'ultimo non sia veicolo per l'introduzione di falle di sicurezza nei sistemi che lo ospitano.

La moderna industria del software prevede, come pratica di routine, l'utilizzo di librerie e componenti software di terze parti da integrare nei propri programmi.
Questo codice è spesso condiviso pubblicamente tramite repository (e pratiche) open source.
Tuttavia, nell'utilizzo di questo codice di solito ci si propone di adottarlo per assolvere alle necessità funzionali necessarie ai sistemi di cui entra a fare parte, senza tenerne in considerazione dettagli e requisiti di sicurezza.
Un aspetto fondamentale per produrre software di qualità è quindi avere la possibilità di analizzare e valutare la sicurezza di queste librerie prima e dopo che vegano inserite all'interno del proprio codice (e del proprio software).
Gruppo SIGLA S.r.l.[1] (azienda del gruppo RELATECH) in quanto azienda produttrice di software ha un precipuo interesse nel garantire secondo standard industriali la qualità del codice prodotto dai suoi progetti.
In particolare, Gruppo SIGLA vuole implementare un analizzatore di librerie di terze parti per supportare i suoi tecnici nel rispetto di specifici requisiti di qualità del codice, in termini di sicurezza.

1.1 Struttura del documento

Il seguente elaborato è suddiviso sostanzialmente in tre parti principali:

- la prima tratta il concetto di analisi del codice, in particolar modo viene descritta la dimensione del problema trattato e viene sottolineata l'importanza dell'uso dell'analisi

[1] http://www.grupposigla.it/

statica che garantisce una serie di miglioramenti sostanziali riguardo allo sviluppo di un software;

- nella seconda si presentano gli strumenti fornitori di analisi del codice nella loro struttura e in particolar modo verrà presentato SonarQube[2], uno degli analizzatori più conosciuti attualmente sul mercato e utilizzato da Gruppo SIGLA S.r.l.
- nella terza si mostrano una serie di esempi d'uso di SonarQube nella pratica di manutenzione del codice, sia in termini di risoluzione degli errori, che di prevenzione di possibili complicanze future.

1.2 Scopo della tesi

"Il software svolge oggi nel sistema economico lo stesso ruolo che il settore delle macchine utensili ha svolto nella seconda rivoluzione industriale alla fine del secolo scorso, con una differenza sostanziale: i suoi effetti riguardano sia la produzione che il mondo dei servizi e della comunicazione." [1]. Così Mario Raffa (Direttore del Dipartimento di Ingegneria Economico-Gestionale, nonché insegnante di Economia ed Organizzazione Aziendale presso la Facoltà Di Ingegneria dell'Università di Napoli Federico II) e Giuseppe Zollo (professore di Gestione Aziendale presso la Facoltà di Ingegneria dell'Università di Napoli Federico II, nonché ricercatore in Italia e all'Estero sui temi delle nuove tecnologie, delle piccole imprese e della valutazione delle competenze individuali ed organizzative) descrivono il ruolo fondamentale del software in quella che è l'economia moderna nella prefazione del loro libro "Economia del software - Elementi introduttivi", prevedendo così quello che sarebbe stato il contesto economico del nuovo millennio in cui si sono poi diffusi i moderni software e le applicazioni.

Con essi è insorta dunque anche la necessità di migliorarne la qualità e la sicurezza poiché divenuti versatili e diffusisi al punto tale che per distinguersi tra tutti i potenziali concorrenti e rispondere alle esigenze del mercato, l'industria del software ha dovuto concentrare l'attenzione su questi parametri non più trascurabili.

Si è deciso perciò di introdurre l'utilizzo dell'analisi del codice, suddivisa in analisi statica (eseguita post-compilazione e pre-esecuzione del software per scoprire criticità non individuabili in compilazione) e analisi dinamica (in fase d'esecuzione del software per osservare e gestire tutti i possibili comportamenti del software una volta eseguito), per assolvere a queste importanti esigenze.

[2] https://www.sonarqube.org/

Il seguente elaborato si pone l'obiettivo di fornire al lettore una panoramica dell'analisi del codice, concentrandosi nello specifico su cosa sia l'analisi statica del codice e di far comprendere con l'ausilio di esempi concreti l'importanza dell'utilizzo di strumenti di analisi statica al fine di migliorare l'esperienza di scrittura di un software oltre che la sicurezza e la qualità del codice stesso.

Capitolo 2

Analisi del codice

L'analisi del codice rappresenta da sempre uno degli approcci fondamentali a disposizione di tutti i professionisti del settore informatico per controllare la correttezza del software e migliorarne le prestazioni. Attraverso l'analisi del codice è possibile identificare quali parti presentino errori che causano comportamenti indesiderati, oppure quali sezioni del codice riscontrino criticità prestazionali. Nello specifico essa permette di rispondere a vari interrogativi, quali:

- *capire l'esatta funzionalità implementata da un software o da parte di esso;*
- *comprendere le funzionalità aggiunte in diverse versioni dello stesso software;*
- *analizzare i software al fine di trovare porzioni comuni di codice sorgente.*

Tipicamente l'analisi viene svolta sul codice sorgente del software, ovvero sulla sua versione compilata. Il codice sorgente costituisce la rappresentazione iniziale del software, così come descritta dal programmatore, ed in quanto tale è tipicamente espressa attraverso un linguaggio di programmazione noto e di semplice interpretazione da parte dell'analista. Il processo di analisi in questo caso si espleta attraverso l'ispezione del codice sorgente.

Il codice compilato è costituito dal prodotto del processo di compilazione che traduce il codice sorgente in un linguaggio direttamente interpretabile dall'architettura di riferimento per l'esecuzione del software (ad esempio, l'architettura x64).

Tale codice, comunemente anche chiamato codice macchina, è pensato e progettato per essere interpretabile ed eseguibile direttamente, nel modo più efficiente possibile, da un microprocessore. In quanto tale, la sua analisi è notevolmente più complessa!

Il linguaggio macchina non è pensato per essere interpretato da un essere umano, e quindi non presenta costrutti facilmente comprensibili. In particolare, ciascuna istruzione del linguaggio di programmazione originale può essere tradotta, durante il processo di compilazione, in molte istruzioni del linguaggio macchina di destinazione.

La combinazione di queste può cambiare notevolmente a seconda di come il processo di compilazione venga svolto, ed inoltre, ogni riferimento simbolico usato dal programmatore originale (nomi associati ai dati, commenti, documentazione del codice) viene tipicamente cancellato dal processo di compilazione, dato che tali riferimenti non sono utili, né hanno alcun significato, per il microprocessore che dovrà eseguire le istruzioni.

Infine, dato il codice sorgente di un programma, è possibile da questo ottenere codici compilati notevolmente diversi, a seconda di quale architettura venga utilizzata come riferimento per la successiva esecuzione del programma. Infatti, diverse architetture utilizzano linguaggi macchina notevolmente dissimili. L'analista che voglia interpretare il codice compilato dovrà quindi assicurarsi di conoscere lo specifico linguaggio macchina utilizzato dall'architettura per cui il software oggetto dello studio è stato compilato. [2]

Operativamente l'analisi può svolgersi secondo due approcci principali:

- *Analisi statica;*
- *Analisi dinamica.*

2.1 L'analisi statica e l'analizzatore di codice statico

L'analisi statica è un metodo di debug di un programma per computer che viene effettuato da uno strumento di supporto alla programmazione: l'analizzatore di codice statico, esaminando il codice senza eseguire il programma.

L'analizzatore di codice statico è un software che ispeziona un dato codice sorgente, o un dato codice compilato, al fine di scoprire problemi di varia natura che spaziano dai bug al codice duplicato, da questioni di prestazioni ad aspetti di leggibilità. Il processo di analisi statica fornisce una comprensione della struttura del codice e può aiutare a garantire che il codice aderisca agli standard del settore utilizzati nell'ingegneria dai team di sviluppo del software e agli standard di qualità di quest'ultimo richiesti da ciascun specifico contesto di applicazione.

Il software di analisi statica eseguirà la scansione di tutto il codice in un progetto per verificare la presenza di vulnerabilità durante la convalida del codice.

L'analisi statica è generalmente efficace per trovare problemi di codifica come:

- *Errori di programmazione;*
- *Codifica delle violazioni degli standard;*
- *Valori indefiniti;*
- *Errori della sintassi;*
- *Falle di sicurezza.*

2.1.1 Come viene eseguita l'analisi statica?

Il processo è relativamente semplice, purché automatizzato. In genere, l'analisi statica viene eseguita prima del test del software nelle fasi preliminari di ogni sviluppo e può essere parte integrante dei processi afferenti alla gestione del ciclo di vita del software.

Una volta scritto il codice, è necessario eseguire un analizzatore di codice statico[3] per esaminarlo. Verificherà le regole di codifica definite dagli standard o di regole predefinite personalizzate secondo i requisiti collegati al contesto di esecuzione di un software, alle performance necessarie o a termini contrattuali con un cliente. Una volta che il codice è stato verificato attraverso l'analizzatore di codice statico, l'analizzatore avrà identificato se il codice è conforme o meno alle regole impostate. A volte è possibile che il software contrassegni dei falsi positivi, quindi, è importante che qualcuno li esamini e decida autonomamente come trattarli, se ignorarli o avviare ulteriori analisi. Una volta eliminati i falsi positivi, gli sviluppatori possono iniziare a correggere i punti segnalati preliminarmente come errori, generalmente a partire da quelli più critici. Una volta risolti i problemi di codice, quest'ultimo può passare al test tramite l'esecuzione.

Senza strumenti di test del codice, l'analisi statica richiederà molto lavoro, poiché una persona deve rivedere il codice e capire come si possa comportare negli ambienti di esecuzione (runtime). Pertanto, è una buona idea trovare uno strumento che automatizzi il processo. Eliminare tutti i processi tediosi per la loro lunghezza e ripetitività renderà certamente l'ambiente di lavoro più efficiente.

[3] Vedi: 2.1 L'analisi statica e l'analizzatore di codice statico

2.1.2 Tipi di analisi statica

Esistono diversi metodi di analisi statica che un'organizzazione può utilizzare. Tra questi citiamo:

- *Analisi di controllo*: si concentra sul flusso di controllo in una struttura di chiamata. Ad esempio, un flusso di controllo potrebbe essere un processo, una funzione, un metodo o una subroutine.
- *Analisi dei dati*: assicura che i dati definiti vengano utilizzati correttamente, assicurandosi anche che gli oggetti dei dati funzionino correttamente.
- *Analisi guasti / errori*: analizza guasti ed errori nei componenti del modello.
- *Analisi dell'interfaccia*: verifica le simulazioni per controllare il codice e assicurare che l'interfaccia si adatti al modello e alla simulazione.

In un senso più ampio, con una categorizzazione meno ufficiale, l'analisi statica può essere suddivisa in categorie formali, estetiche, di progettazione, controllo degli errori e categorie predittive:

- Significato formale, se il codice è corretto;
- significato estetico, se il codice si adegua a determinati standard di stile;
- proprietà di progetto, sulla base di determinati livelli di complessità;
- controllo degli errori che cerca violazioni del codice;
- predicibilità, che stabilisce come si comporterà il codice una volta eseguito.

2.1.3 Vantaggi e svantaggi dell'analisi statica

L'analisi statica consente di individuare problemi comuni e violazioni delle procedure di programmazione consigliate. Rileva anche i difetti difficili da individuare tramite test. Gli avvisi sono diversi dagli errori e dagli avvisi del compilatore: cerca modelli di codice specifici che possono causare problemi. Ciò significa che il codice è valido, ma potrebbe comunque creare problemi, per l'utente o per altri utenti che usano il codice.

I vantaggi dell'utilizzo dell'analisi statica includono:

- *Si può valutare tutto il codice in un'applicazione, aumentando la qualità del codice;*
- *Fornisce velocità nell'utilizzo di strumenti automatizzati rispetto alla revisione manuale del codice;*

- *Associato ai normali test, il test statico consente una maggiore profondità nel codice di debug;*
- *Gli strumenti automatizzati sono meno soggetti a errori umani;*
- *Aumenterà la probabilità di trovare vulnerabilità nel codice, aumentando la sicurezza del web o delle applicazioni;*
- *Può essere eseguito in un ambiente di sviluppo offline.*

Tuttavia, l'analisi statica presenta alcuni inconvenienti. Ad esempio, i programmatori dovrebbero essere consapevoli di quanto segue:

- *È possibile rilevare falsi positivi;*
- *Se ci fosse un errore nel codice, uno strumento potrebbe non rilevarlo;*
- *Non tutte le regole di codifica possono essere sempre seguite, come le regole che richiedono documentazione esterna;*
- *L'analisi statica non è in grado di rilevare come verrà eseguita una funzione;*
- *Potrebbe non essere possibile analizzare le librerie di sistema di terze parti.*

2.1.4 Verifica statica vs. verifica dinamica

Il vantaggio principale dell'analisi statica consiste nella possibilità di rivelare errori che non si manifestano fino a quando non si verifica un bug report dopo settimane, mesi o anni dopo il rilascio. Dopo che è stata eseguita l'analisi statica, l'analisi dinamica viene spesso eseguita nel tentativo di scoprire difetti o vulnerabilità sottili. L'analisi dinamica prevede il test e la valutazione di un programma basato sull'esecuzione.[3]

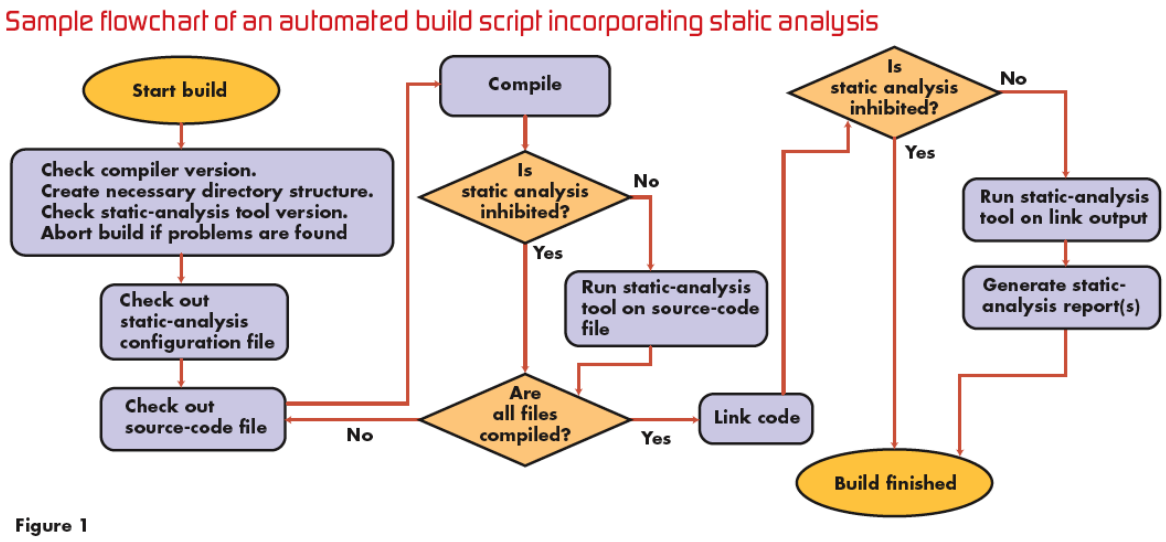

Figura 1: flowchart di un software che incorpora l'analisi statica

Figura 1 rappresenta un diagramma di flusso in cui viene mostrato dove si inserisca l'analisi statica del codice all'interno delle fasi dello sviluppo di un software. In particolar modo si descrive che:

- in seguito alla compilazione l'analizzatore andrà a verificare le varie vulnerabilità del codice sorgente dando dei possibili suggerimenti per la conseguente correzione
- prima della fase di testing per poter effettuare una serie di controlli preliminari e generare dunque una serie di report sui vari test effettuati, in modo da mostrare quali parti del software hanno superato questi test e quali parti invece non avendo superato il controllo potrebbero essere responsabili dei possibili errori in fase di esecuzione e dunque sarebbe il caso di effettuare delle correzioni al fine di superare il controllo.

Figura 2: fasi dello sviluppo di un applicativo nelle quali opera l'analisi del codice

L’analizzatore statico interviene per l’appunto durante la fase di implementazione fornendo una serie di strumenti per visionare il codice in maniera più efficace ed efficiente, ad esempio con una serie di marcatori presentati nel capitolo successivo, e durante la fase di test e integrazione grazie ai report contenenti informazioni sulle varie parti di codice che hanno o superato o meno una serie di controlli prestabiliti.

2.2 Analisi Dinamica

Questo secondo approccio prevede che il software venga eseguito, completamente o in alcune sue parti, per poterne osservare i diversi comportamenti ed effetti a fronte di diversi stimoli. In questo modo è possibile studiare direttamente gli effetti indesiderati ricercati. Laddove questi effetti avessero conseguenze negative sul sistema di analisi o su altri sistemi, l’analisi deve essere svolta con particolare cautela. Nello specifico qualora il codice analizzato sia un software dannoso (ad esempio un virus, o più genericamente un malware), l’esecuzione deve avvenire in particolari ambienti controllati. [2]

Capitolo 3

Strumenti e fornitori di analisi statica

Ci sono molti strumenti di verifica statica messi a disposizione di aziende sul mercato e dalla comunità open source. Di conseguenza, potrebbe non essere facile scegliere quelli più idonei. Gli strumenti software funzioneranno a una varietà di livelli:

- *Gli strumenti a livello di unità esaminano programmi o subroutine.*
- *Gli strumenti a livello di tecnologia testeranno tra i programmi unitari e una visione del programma generale.*
- *Gli strumenti a livello di sistema analizzeranno le interazioni tra i programmi delle unità.*
- *E gli strumenti a livello di obiettivi si concentreranno su termini, regole e processi a livello di obiettivi.*

Prima di utilizzare uno strumento, i programmatori dovrebbero anche assicurarsi che lo strumento supporti il linguaggio di programmazione che stanno utilizzando e gli standard da rispettare.

Embold[4] è un esempio di strumento di analisi statica che afferma di essere una piattaforma di analisi software intelligente. Lo strumento può assegnare automaticamente la priorità ai problemi con il codice e fornirne una chiara visualizzazione. Lo strumento verificherà anche la correttezza e l'accuratezza dei modelli di progettazione utilizzati nel codice.

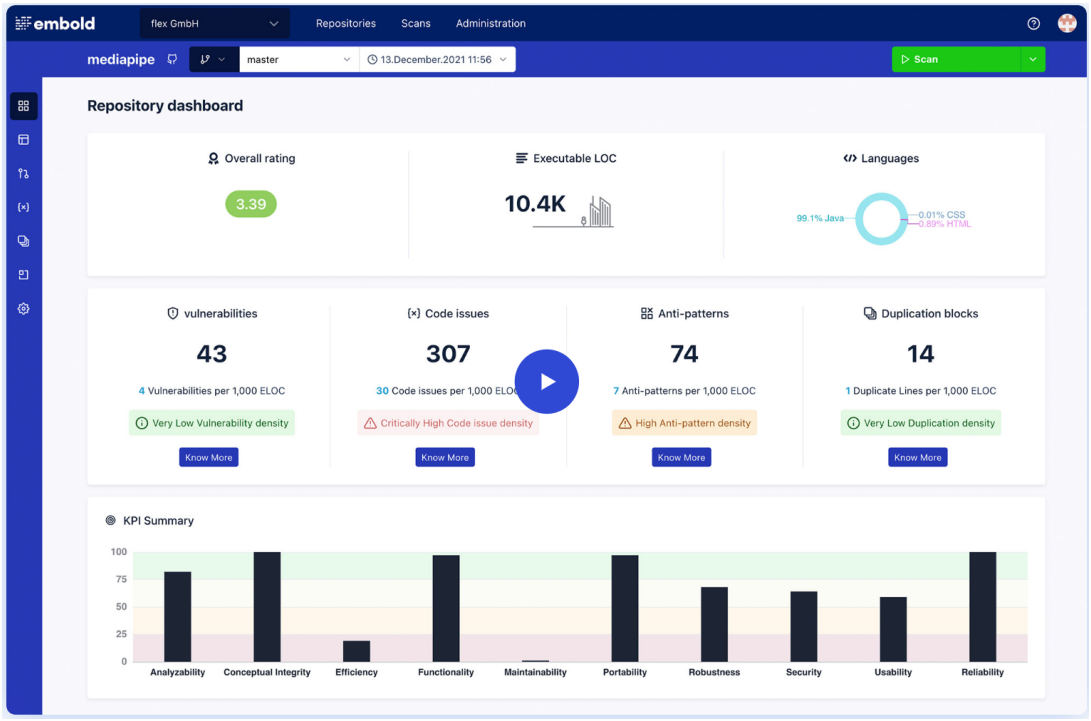

Figura 3: Embold

Kiuwan[5] è un altro esempio di analizzatore statico. È una grande piattaforma che si concentra sull'implementazione dell'analisi statica in un ambiente DevOps. Presenta fino a 4,000 regole aggiornate basate su circa 25 standard di sicurezza. Si integra bene anche con Jenkins.

[4] https://embold.io/

[5] https://www.kiuwan.com/

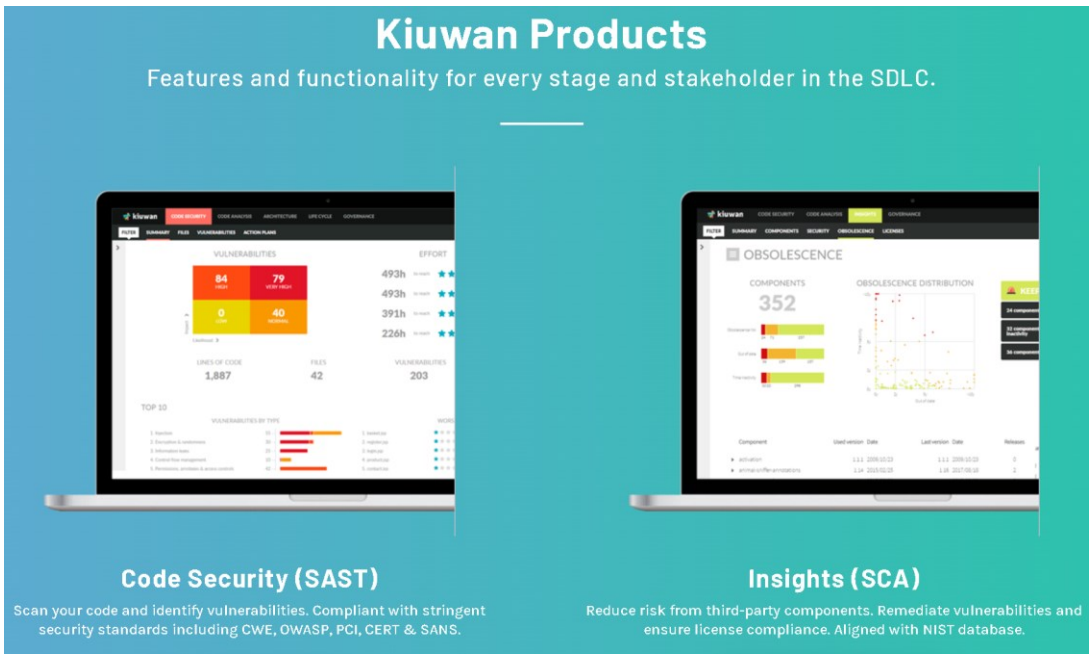

Figura 4: Kiuwan

PyCharm[6] è un altro strumento di esempio creato per gli sviluppatori che lavorano in Python con basi di codice di grandi dimensioni. Lo strumento offre navigazione nel codice, refactoring automatico e una serie di altri strumenti di debugging. [3]

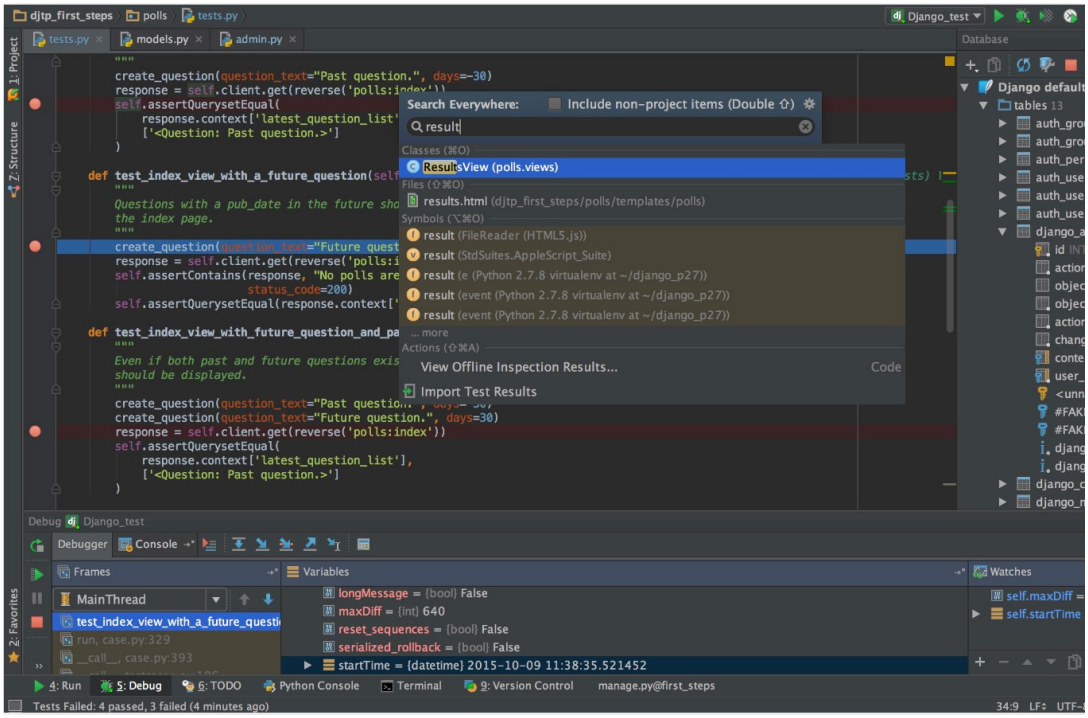

Figura 5: PyCharm

[6] https://www.jetbrains.com/pycharm/

3.1 Livelli di gravità degli analizzatori

Una delle feature più utili in fase di sviluppo del software è rappresentata certamente dal *linter*, uno strumento che, integrandosi nell'editor, analizza il codice sorgente per contrassegnare errori di programmazione, bug, errori stilistici e costrutti sospetti. Il termine deriva dall'omonimo strumento Unix che esamina il codice sorgente del linguaggio C.

Il *linter* è in grado di segnalare una serie di criticità, le quali possono presentare uno dei livelli di gravità seguenti (in questo caso è basato su Sonarlint, tuttavia le modalità di segnalazione possono essere molto simili in altri analizzatori):

Gravità	Comportamento in fase di compilazione	Comportamento dell'editor
error	Le violazioni vengono visualizzate *come errori* nell'elenco errori e nell'output di compilazione della riga di comando e causano errori nelle compilazioni.	Il codice offensivo viene sottolineato con una sottolineatura a sottolineatura rossa e contrassegnato da una piccola casella rossa nella barra di scorrimento.
Warning	Le violazioni vengono visualizzate *come avvisi* nell'elenco errori e nell'output di compilazione della riga di comando, ma non causano errori nelle compilazioni.	Il codice che ha commesso l'offesa è sottolineato con una sottolineatura a scorrimento verde e contrassegnato da una piccola casella verde nella barra di scorrimento.
Suggestion	Le violazioni vengono visualizzate *come Messaggi* nell'elenco errori e non nell'output di compilazione della riga di comando.	Il codice che ha commesso l'offesa è sottolineato con una sottolineatura a sottolineatura grigia e contrassegnato da una piccola casella grigia nella barra di scorrimento.
Silent	Non visibile all'utente.	La diagnostica viene tuttavia segnalata al motore di diagnostica IDE.

None	Eliminato completamente.	Eliminato completamente.
Default	Corrisponde alla gravità predefinita della regola.	Corrisponde alla gravità predefinita della regola.

Figura 6: Generiche problematiche del codice segnalate da un analizzatore

Se le violazioni delle regole vengono trovate da un analizzatore, vengono segnalate nell'editor di codice e nella finestra Elenco errori:

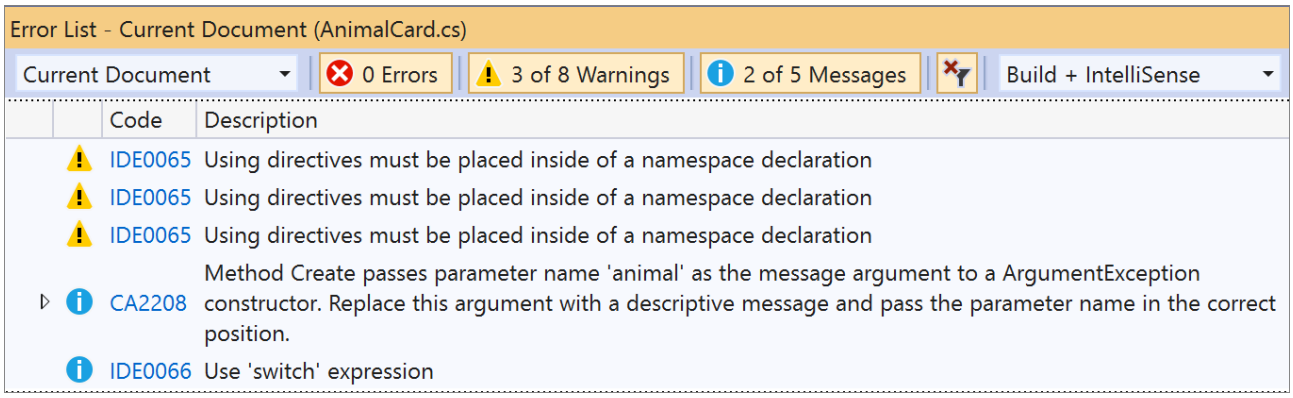

Figura 7: Finestra di elenco degli errori

Le violazioni dell'analizzatore segnalate nell'elenco errori corrispondono all'impostazione del livello di gravità della regola. Le violazioni di cui sopra vengono inoltre mostrate nell'editor di codice sotto il codice in errore. L'immagine seguente mostra tre esempi di violazioni: un errore (segno in rosso), un avviso (segno in verde) e un suggerimento (tre punti grigi):

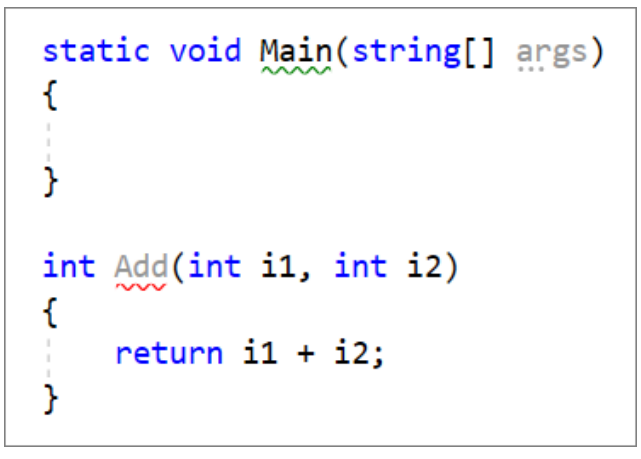

Figura 8: Esempio di segnalazione grafica di criticità del codice sorgente

Molte regole dell'analizzatore hanno una o più correzioni di codice associate che è possibile applicare per correggere la violazione della regola. Le correzioni del codice vengono

visualizzate nel menu con l'icona a forma di lampadina insieme ad altri tipi di azioni rapide. [4]

3.2 SonarQube

SonarQube[7] è uno strumento di revisione automatica del codice per rilevare bug, vulnerabilità e code smells nel codice. Può integrarsi con il flusso di lavoro esistente per consentire l'ispezione continua del codice tra i project branches e le pull requests.

In un tipico processo di sviluppo:

- *Gli sviluppatori implementano e uniscono il codice in un IDE (preferibilmente utilizzando SonarLint per ricevere un riscontro immediato nell'editor) e archiviano il codice nella loro piattaforma DevOps.*
- *Lo strumento di integrazione continua (CI) adottato in un processo di sviluppo verifica, crea ed esegue unit test e uno scanner SonarQube integrato analizza i risultati.*
- *Lo scanner invia i risultati al server SonarQube che fornisce riscontri agli sviluppatori tramite l'interfaccia SonarQube, e-mail, notifiche in-IDE (tramite SonarLint) e decorazioni su richieste pull o merge (quando si utilizza Developer Edition e versioni successive).*

Figura 9: Sonarqube, Sonarlint e Sonarcloud

La figura 9 mostra una serie di strumenti di supporto specializzati di Sonarqube per determinati scopi, come il controllo del codice sorgente (Sonarlint) e dei repository in cloud (Sonarcloud).

[7] https://www.sonarqube.org/

3.2.1 Scrivere codice pulito e sicuro

SonarQube offre diversi strumenti per scrivere codice pulito e sicuro:

- *SonarLint:* è un prodotto complementare che funziona nell'editor fornendo un riscontro immediato in modo da poter rilevare e risolvere i problemi prima che arrivino al repository;
- *SonarCloud*: analizza automaticamente i progetti nei repository online e riporta tramite decorazioni delle richieste git un resoconto delle problematiche riscontrate;
- *Quality Gate:* consente di sapere se il progetto è pronto per la produzione;
- *Problemi:* SonarQube solleva problemi ogni volta che una parte del codice infrange una regola di codifica, sia che si tratti di un errore che interromperà il codice (bug), di un punto del tuo codice aperto agli attacchi (vulnerabilità) o di un problema di manutenibilità (code smell);
- *Hotspot di sicurezza:* Sonarqube inoltre segnala delle porzioni di codice che secondo le stime dell'analizzatore potrebbero rappresentare delle criticità; tuttavia, esse possono in base alla scelta dello sviluppatore essere trascurate o meno. [5]

3.2.2 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 [6]. 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 [7]. Gli studenti o i neolaureati generalmente 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 negativamente la loro produttività e il loro benessere. 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 [8]. In [9] 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 [10] 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 [8].

Capitolo 4

Esempi d'uso di SonarQube

4.1 introduzione generale alle prove fatte e spiegazione obiettivi

In questo capitolo vedremo una panoramica dell'utilizzo pratico di SonarQube unito a una serie di esempi che mostrano come questo analizzatore possa fornire agli sviluppatori una serie di strumenti che se sfruttati al pieno delle loro potenzialità possono migliorare in maniera consistente oltre che la sicurezza e qualità del software anche l'esperienza stessa di revisione del codice, rendendola più semplice, efficace, efficiente e veloce.

4.2 metodologia adottata per i test fatti con SonarQube

Per effettuare la scannerizzazione dei propri progetti e visualizzare i conseguenti report generati sarà necessario rispettare una serie di prerequisiti hardware e compiere alcuni passi preliminari per configurare il pc come descritto nel dettaglio nella documentazione ufficiale di SonarQube[8] e riportato in sintesi di seguito:

1. controllare di avere installata una versione di Java compatibile (attualmente ver. 11 per il Server SonarQube e ver. 11 o 17 per lo Scanner);
2. aver configurato una versione compatibile di uno dei linguaggi SQL su cui si appoggia SonarQube (PostgreSQL, Microsoft SQL server, Oracle);
3. avere a disposizione un Web Browser compatibile aggiornato all'ultima versione (Chrome, Firefox, Edge, Safari);
4. disporre di un pc con almeno 2GB di memoria RAM e spazio libero sul disco necessario in quantità dipendente dalla dimensione del progetto che si vuole analizzare.

[8] https://docs.sonarqube.org/latest/requirements/requirements/

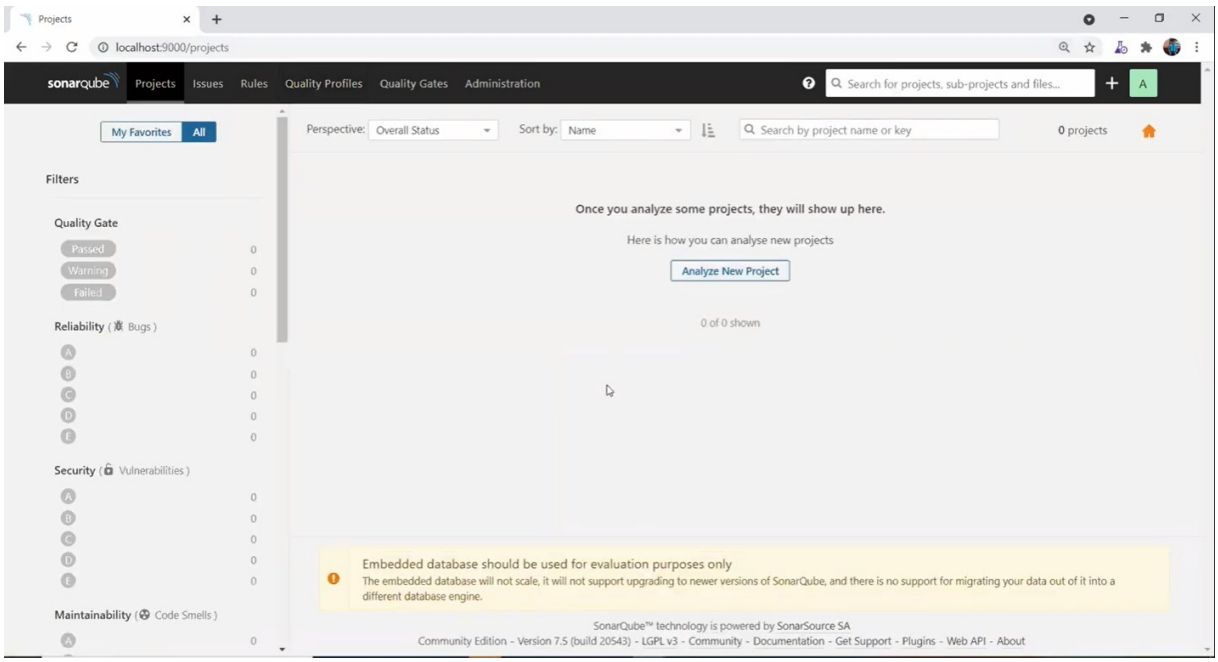

Figura 10: Schermata iniziale Server SonarQube

In seguito alla verifica dei passi preliminari di cui sopra sarà necessario aver scaricato l'ultima versione di SonarQube dal sito ufficiale[9], estrarre la cartella presente nell'archivio scaricato, navigare nella cartella all'interno della sottocartella bin e successivamente in quella dal nome corrispondente al sistema operativo del computer utilizzato, aprire nella sottocartella un terminale e far eseguire il programma StartSonar.bat.

A questo punto sarà possibile come riportato in Figura 10 aprire una pagina web collegandosi all'indirizzo localhost:9000, cioè indirizzo IP locale del pc e porta 9000.

In questo modo si verrà indirizzati alla schermata home del server SonarQube per poter iniziare ad analizzare i propri progetti.

Per selezionare un progetto da analizzare basterà seguire la procedura mostrata nel sito cliccando il bottone "Analyze New Project", secondo le direttive della schermata che comparirà fornire delle stringhe per identificare il progetto, selezionare poi il linguaggio di programmazione principale con cui il progetto è stato scritto (sincerandosi che sia tra i linguaggi supportati da SonarQube) e infine far eseguire nel terminale aperto in precedenza uno alla volta i tre comandi che verranno generati.

Questi tre comandi in sequenza:

1. Installeranno i vari tool di SonarQube all'interno della cartella del progetto;

[9] https://www.sonarqube.org/

2. Ricompileranno il progetto controllando che non siano presenti errori di compilazione (nel qual caso sarà necessario risolverli prima di lanciare il comando successivo);
3. Effettueranno la scansione del progetto nella sua interezza per aggiornare la pagina del Server SonarQube contenente le informazioni raccolte sul progetto.

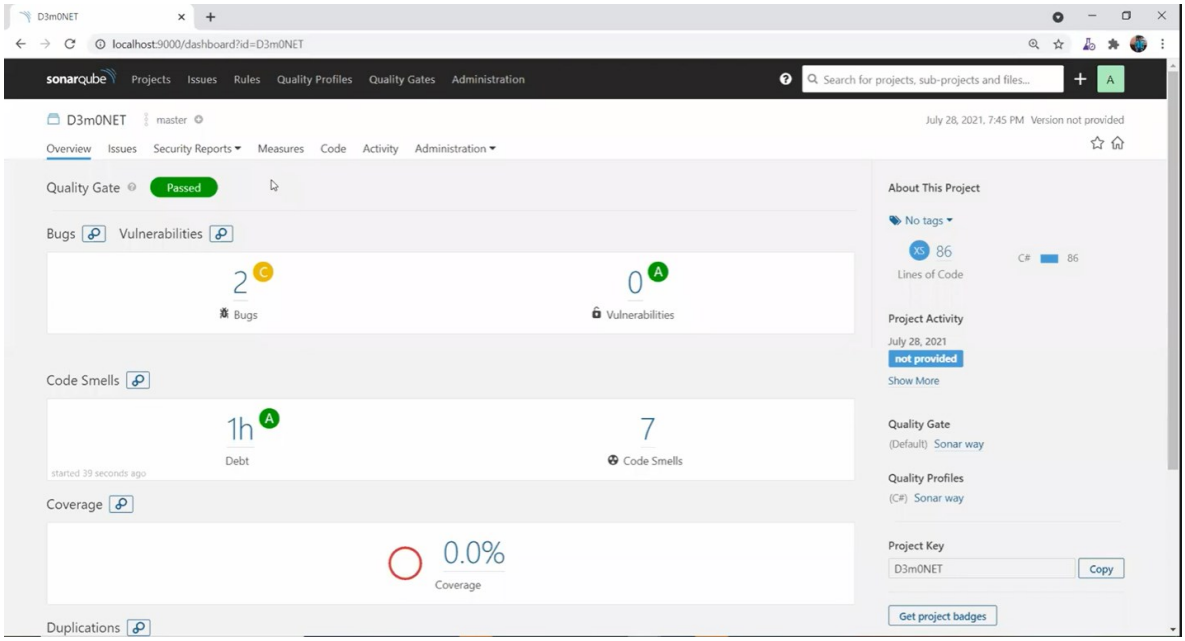

Figura 11: Esempio di schermata principale del Server SonarQube in seguito alla selezione di un progetto

Come mostrato nell'esempio in Figura 11 il server SonarQube, in seguito a un refresh della pagina web, fornirà un riepilogo della scansione effettuata e una visione nel dettaglio delle varie problematiche eventualmente riscontrate all'interno del progetto.

Nella prossima sezione vedremo alcuni esempi di varie criticità riscontrabili e di come sia possibile gestirne la risoluzione.

4.3 Report test

Dopo aver eseguito correttamente la procedura vista nella sezione precedente per scannerizzare un progetto verranno dunque raccolte tutte le informazioni relative alle criticità del software all'interno di un report.

Questo report è costituito da una serie di schermate visualizzabili all'interno del server SonarQube che mostrano un riepilogo di tutte le problematiche riscontrate sia come visione d'insieme che nel dettaglio una ad una. Nel primo caso il report apparirà come mostrato in figura 11 raggruppando tutti i dati relativi alla scannerizzazione, altresì nel secondo, una

volta selezionata una categoria tra quelle visualizzabili, verrà mostrato l'elenco delle criticità in ordine cronologico all'interno del codice sorgente con annesso riferimento.

La caratteristica più propriamente interessante della visualizzazione nel dettaglio però è la possibilità da parte dello sviluppatore di revisionare il codice in modo veloce ed efficace.

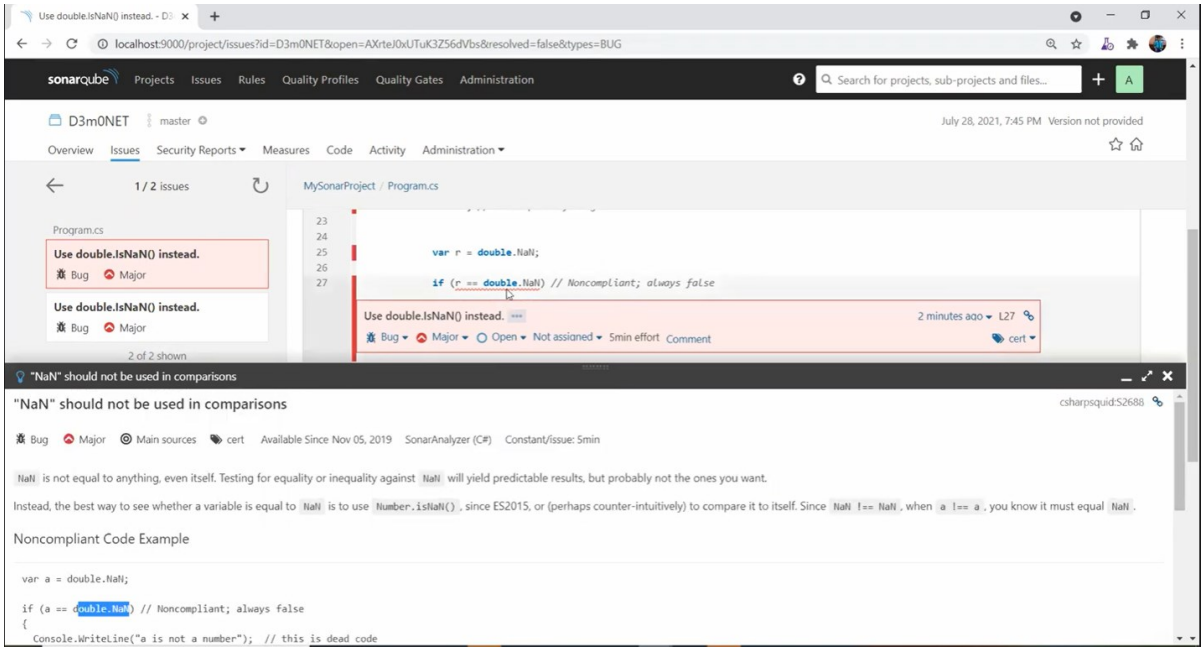

Figura 12: Esempio di bug segnalato da SonarQube con annessa spiegazione dell'errore riscontrato

Interagendo con gli elementi dell'elenco a sinistra dello schermo si potrà visionare il punto esatto dove si trova la singola problematica nel sorgente e, grazie anche ad una spiegazione annessa dell'errore in questione (figura 12) affiancata ad un suggerimento di SonarQube sulla correzione (figura 13), si otterrà una notevole semplificazione del processo di manutenzione del codice.

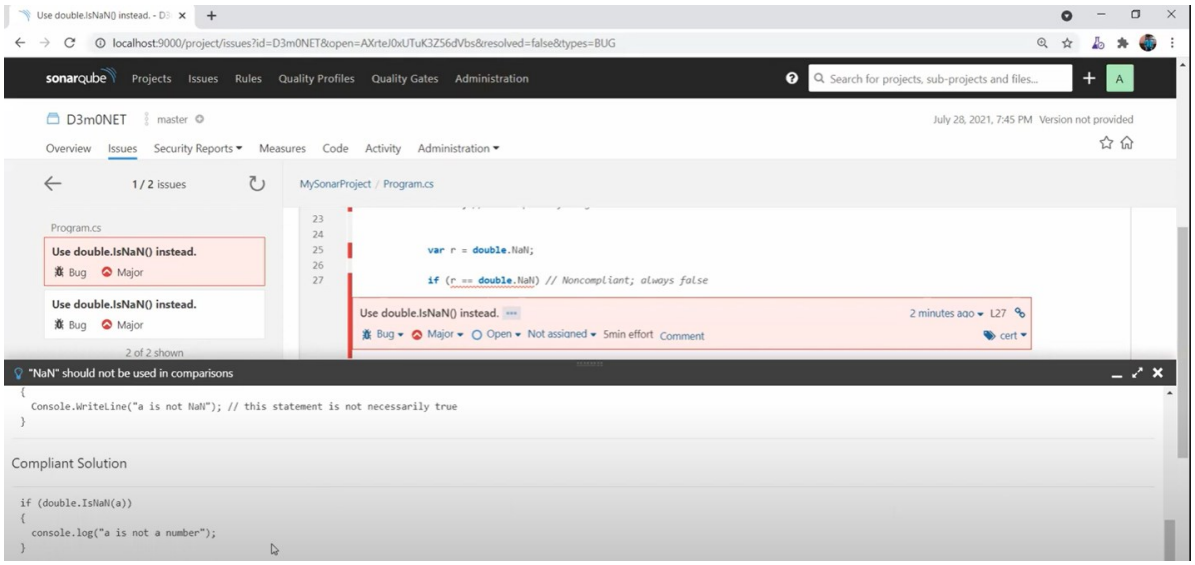

Figura 13: Suggerimento indicato da SonarQube per risolvere il bug evidenziato

Quest'ultimo punto è cruciale per comprendere il vantaggio che caratterizza l'uso di SonarQube, in quanto il mancato utilizzo di questo strumento di supporto alla programmazione ha costretto la maggior parte degli sviluppatori per molto tempo a focalizzare molte delle loro energie nella ricerca manuale degli errori in esecuzione, la loro collocazione nel codice, nel comprenderne l'esatta natura e molto spesso rimanendo con la frustrazione di non aver trovato la soluzione in tempi brevi o peggio di non averla proprio trovata.

Con SonarQube invece queste tempistiche si possono ridurre notevolmente e si è in grado di risolvere i problemi legati al codice più efficacemente rispetto a prima, poiché gran parte della ricerca degli errori viene automatizzata.

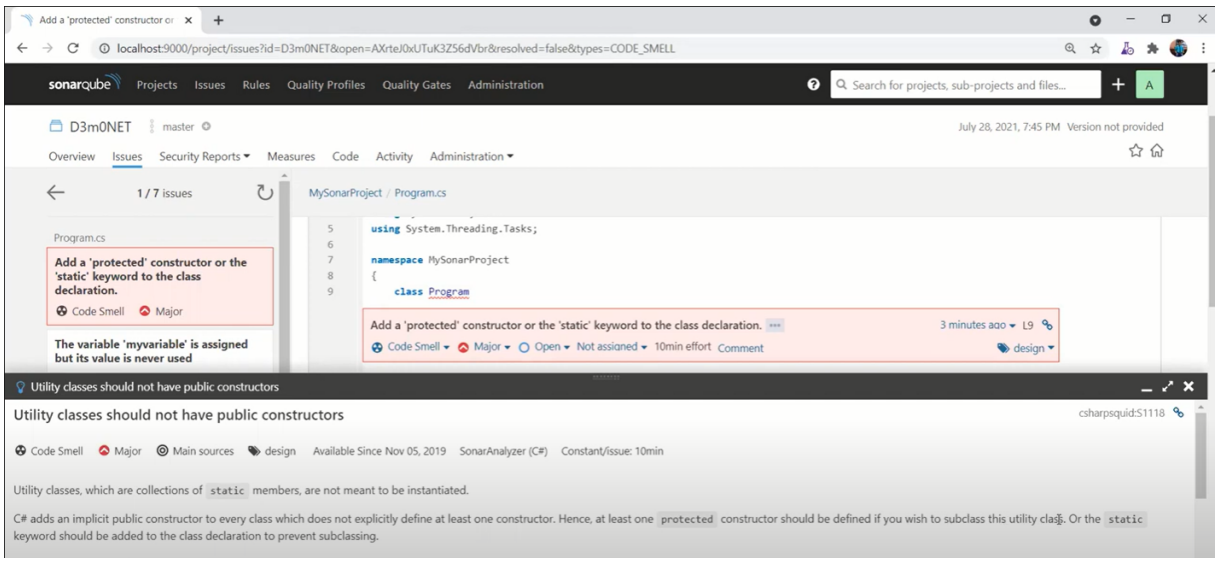

Figura 14: Esempio di code smell segnalato da SonarQube con annessa spiegazione dei possibili problemi che potrebbe comportare

SonarQube tuttavia non solo è in grado di gestire i *bug* del codice, ma riesce anche a migliorarne la manutenibilità futura segnalando anche i *code smells* che dal punto di vista della funzionalità non presentano falle, ma che per future modifiche al sorgente potrebbero generare inutili (se non dannosi) rallentamenti, ambiguità o peggio nuovi *bug* nel codice.

In figura 14 ed esempio viene segnalata una scelta di progettazione che non rispetta quelli che sono degli standard consolidati nell'ingegneria del software, in questo caso l'utilizzo di un costruttore pubblico per una "Utility Class", cioè una classe contenente solo metodi statici che di norma non ammette costruttori per poterla istanziare.

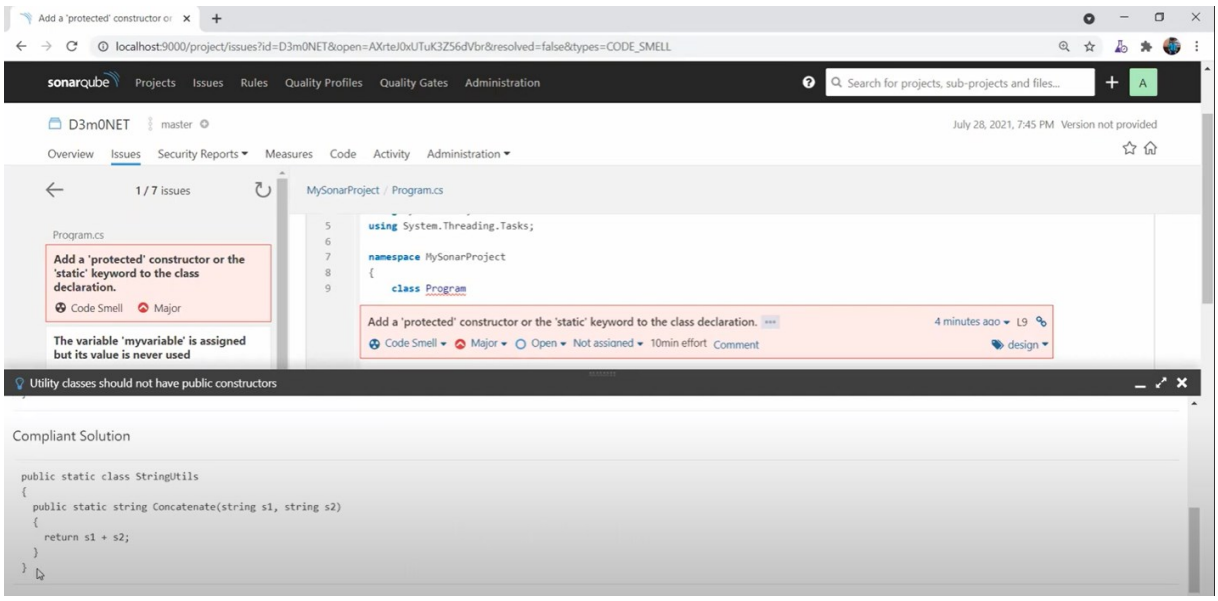

Figura 15: Suggerimento indicato da SonarQube per risolvere il code smell evidenziato

Una possibile soluzione, come suggerito da SonarQube in figura 15, consiste nel rendere il costruttore della classe in questione “protected” impedendo così qualunque tentativo da parte di elementi esterni alla classe di creare nuove istanze indesiderate della suddetta classe.

Un accorgimento di questo tipo, anche se può sembrare esiguo, potrebbe davvero fare la differenza ed evitare inutili operazioni nel codice scaturite da banali ambiguità di interpretazione, e queste accortezze stavano interamente alla bravura di chi progettava il software prima dell’introduzione questa tecnologia.

Con l’auspicabile maggior diffusione dell’uso di analizzatori statici si potranno dunque concentrare le energie dedicate alla manutenzione su altre attività che richiederebbero maggiore attenzione e automatizzare invece la risoluzione di altre operazioni più semplici.

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 è stato introdotto il contesto in cui si inserisce la tesi unito alla struttura di questa, nel secondo sono stati presentati i concetti fondamentali che riguardano l'analisi del codice, nel terzo capitolo vengono spiegati gli analizzatori, in particolar modo SonarQube, riprendendo quelle che sono le loro metriche spiegate a livello teorico nel capitolo precedente e nel quarto è stata presentata una serie di esempi di utilizzo dell'analisi statica con SonarQube per mostrare i sostanziali miglioramenti che questa può apportare in qualunque progetto essa si inserisca. Una prospettiva interessante sarebbe 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 alle importanti tematiche della sicurezza e della qualità del software, divenute in questi ultimi anni di massivo progresso tecnologico non più trascurabili.

Bibliografia

[1] M. Raffa, G. Zollo, “Economia del software - Elementi introduttivi”, 2000, prefazione.

[2] https://www.agendadigitale.eu/cultura-digitale/analisi-del-codice-nuova-frontiera-dellintelligenza-artificiale/

[3] https://tecnologico.wiki/analisi-statica-analisi-statica-del-codice/

[4] https://docs.microsoft.com/it-it/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022

[5] https://docs.sonarqube.org/latest/

[6] 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.

[7] 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.

[8] 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.

[9] J. Berg and E. Gralén, “The effects of continuous code inspection on code quality,” 2019.

[10] 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.