# Analisi NewLang Compiler

Angelo Afeltra Mtr: 0522501354

25 Maggio 2023

## 1 Introduzione

NewLang Compiler è un progetto open-source realizzato durante il corso di compilatori. Esso presenta una struttura ben definita composta da diverse componenti: sono presenti due classi (Lexer e Parser) ed un interfaccia (sym) che vengono generate automaticamente tramite l'utilizzo di strumenti come JFlex e Java Cup in fase di build del progetto.

Le suddette classi sono responsabili della generazione dell'Abstract Syntax Tree (AST): una struttura ad albero che rappresenta in modo astratto la struttura del codice. Le classi che rappresentano i nodi di tale albero sono dichiarate all'interno del package Nodi.

Per gestire in modo efficiente il processo di analisi dell'AST, il progetto fa ampio uso del design pattern Visitor, in particolare vengono effettuate tre visite sull'AST, ciascuna implementata all'interno di una specifica classe: SemanticVisitor1, SemanticVisitor2 e TranslatorVisitor. Le prime due visite si occupano della generazione dei diversi scope, che vengono rappresentati dalla classe SymbolTable e analizzano attentamente la correttezza semantica dello script da compilare. La classe TranslatorVisitor, invece, ha il compito di tradurre l'AST in uno script C.

Per integrare e coordinare l'intero processo è presente la classe NewLang, che contiene il main del progetto. All'interno di questa classe lo script viene preso in input e successivamente l'AST viene estratto utilizzando la classe Parser. Le visite semantiche, effettuate tramite SemanticVisitor1 e SemanticVisitor2, permettono di analizzare accuratamente il codice dal punto di vista semantico e successivamente grazie alla classe TranslatorVisitor, esso viene tradotto in uno script C.

Infine, il progetto esegue la compilazione dello script C utilizzando il compilatore gcc completando così l'intero ciclo di analisi e traduzione.

L'analisi che verrà condotta all'interno di questo progetto si articola in diversi passaggi, ognuno dei quali riveste un ruolo fondamentale nel valutare e migliorare la qualità del software. Ecco in dettaglio i vari passi:

1. Sarà creata una pipeline di Continuous Integration (CI) che avrà il compito di automatizzare la compilazione del codice garantendo che il progetto venga buildato correttamente. Inoltre, essa si occuperà anche del caricamento dell'immagine Docker corrispondente su Docker Hub agevolando così la distribuzione e la gestione del software.

2. Per assicurare la qualità del codice, sarà eseguita un'analisi approfondita del progetto tramite SonarCloud. Questo strumento di analisi statica del codice fornirà una serie di metriche e segnalazioni utili per identificare potenziali problemi e migliorare la manutenibilità del software. SonarCloud verrà integrato nella pipeline di CI, in modo che l'analisi sia eseguita automaticamente ad ogni build del progetto.

3. Un aspetto rilevante che verrà preso in considerazione è il consumo energetico del progetto. A tale scopo, si utilizzerà lo strumento EcoCode, che consentirà di valutare l'impatto del software prodotto.

4. Una fase cruciale del processo di sviluppo del software è il testing. Verrà eseguita una completa fase di testing del progetto, che includerà diverse strategie e strumenti per garantire l'affidabilità e la robustezza del software. In particolare, verranno generati test mutazionali, che consentono di valutare la capacità del codice di gestire input diversi e di rilevare eventuali errori. Inoltre, verrà utilizzato il tool EvoSuite per la generazione automatica di casi di test, semplificando il processo di creazione di test esaustivi e completi.

5. Sarà sfruttato il performance testing, per valutare le prestazioni del sistema in relazione a diversi carichi di lavoro.

6. Per osservare la sicurezza del software, sarà eseguita un'analisi approfondita delle vulnerabilità presenti nel progetto. A tale scopo, verranno utilizzati strumenti specializzati come FindSecBugs, OWASP DC e OWASP ZAP. Questi strumenti si concentreranno sull'individuazione di possibili falle di sicurezza, vulnerabilità note o dipendenze non aggiornate che potrebbero rappresentare un rischio per l'applicazione.

Attraverso l'esecuzione di questi passaggi l'analisi completa del progetto fornirà una visione approfondita della qualità del software permettendo di identificare e risolvere eventuali problemi, migliorando così l'affidabilità, la sicurezza e le prestazioni complessive dell'applicazione.

# 2 CI Maven Build & Docker

Prima di creare la pipeline di Continuous Integration (CI), poiché il sistema originale non disponeva di un'interfaccia grafica (gli script da campilare venivano passati tramite linea di comando chiamando il main), a scopo didattico per simulare un architettura a microservizi è stato sviluppato un servizio di font-end.

In tal modo il sistema viene suddiviso in due servizi indipendenti tra loro front-end in cui è stata implementata un'interfaccia web che consente agli utenti di caricare il file da compilare e back-end in cui è implementato il compilatore.

Questo approccio ha consentito di separare chiaramente le responsabilità tra il front-end, che gestisce l'interazione con l'utente, e il back-end, che si occupa dell'elaborazione e della compilazione del file. La Rest API offre un'interfaccia per comunicare tra i due servizi, consentendo un'interazione fluida e controllata.

Per la creazione di una pipeline di Continuous Integration (CI), sono state sfruttate le GitHub Actions. Poiché entrambi i servizi (front-end e back-end) avevano già un sistema di build basato su Maven è stata configurata una specifica GitHub Action per eseguire la build automaticamente ad ogni commit.

Ogni volta che viene effettuato un commit nel repository, la GitHub Action appositamente configurata viene attivata avviando così il processo di build tramite Maven. Questo permette di verificare immediatamente se il codice è corretto e di individuare eventuali errori o problemi di compilazione.

```yaml
# This workflow will build a Java project with Maven, and cache/restore any dependencies to improve the workflow execution time
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven

# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.

name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file NewLang/pom.xml
```

Figure 1: GitHub Action Build Maven.

Per quanto rigurada invece la contenerizzazione del sistema è stato sfruttato il progetto open-source Docker.

Il processo di utilizzo di Docker inizia dalla definizione di un file chiamato Dockerfile, che dettaglia i passaggi necessari per creare un'immagine Docker. Questa immagine rappresenta uno stato statico dell'applicazione, comprendendo il codice sorgente, le dipendenze e le istruzioni per l'esecuzione.

E stato creato un Dockerfile personalizzato per ciascuno dei servizi coinvolti nel progetto:


Figure 2: DockerFile NewLangUI & DockerFile NewLang.

I due DockerFile risultano essere simili, tranne che nel primo, ovvero quello del sevizio di front-end, è presente un expose che va ad esporre il container sulla porta 8081, in modo che l'utente potrà contattarlo. Cosa che non è possibile nel servizio di back-end, esso non potrà essere chiamato direttamente dagli utenti ma solamente dal servizio di front-end. Inoltre nel DockerFile del servizio di back-end sono presenti due comandi RUN per l'installazione del compilatore gcc.

Successivamente alla creazione dei DockerFile è stata integrata un'altra GitHub Action nella pipeline di CI che ad ogni commit utilizzando i rispettivi Dockerfile genera due immagini distinte e le carica su Docker Hub. Infine è stato creato un file docker-compose per la gestione dei differenti container, il quale richiamando le immagini da Docker Hub consente di eseguire il sistema con l'ultima versione disponibile.

```yaml
name: Publish Docker image

on:
  schedule:
    - cron: '19 3 * * *'
  push:
    branches: [ "main" ]
    # Publish semver tags as releases.
    tags: [ 'v*.*.*' ]
  pull_request:
    branches: [ "main" ]


jobs:
  push_to_registry:
    name: Push Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@98669ae865ea3cffbcbaa878cf57c20bbf1c6c38
        with:
          images: angeloafeltra/newlang

      - name: Build and push Docker image
        uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
        with:
          context: ./NewLangUI
          file: ./NewLangUI/Dockerfile
          push: true
          tags: angeloafeltra/newlang:front-end
          labels: ${{ steps.meta.outputs.labels }}

      - name: Build and push Docker image back-end
        uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
        with:
          context: ./NewLang
          file: ./NewLang/Dockerfile
          push: true
          tags: angeloafeltra/newlang:back-end
          labels: ${{ steps.meta.outputs.labels }}
```

Figure 3: GitHub Action Docker.

```yaml
version: '3'
services:
  front-end:
    container_name: front-end
    image: angeloafeltra/newlang:front-end
    ports:
      - 8081:8081
    depends_on:
      - back-end

  back-end:
    container_name: back-end
    image: angeloafeltra/newlang:back-end
    ports:
      - 8080
```

Figure 4: File DockerCompose.

In sintesi, grazie all'utilizzo di Docker e la relativa GitHub Action, è stata ottenuta una contenerizzazione efficace del sistema, semplificando la creazione delle immagini, il caricamento su Docker Hub e garantendo un'implementazione coesa e sempre aggiornata del sistema tramite l'utilizzo di docker-compose.

# 3 Sonarcloud

SonarCloud rappresenta un servizio di analisi del codice basato su cloud appositamente progettato per identificare potenziali problemi all'interno del codice. Utilizzando un'analisi statica che non richiede l'esecuzione effettiva del codice, SonarCloud mira a individuare diverse tipologie di problematiche, tra cui: Bugs, Code Smells, Vulnerabilità e Security Hotspots.

Esso è stato integrato nella pipeline di Continuous Integration (CI) mediante l'aggiunta di un'apposita GitHub Action in modo tale da monitorare la qualità di ogni commit effettuato nel sistema.

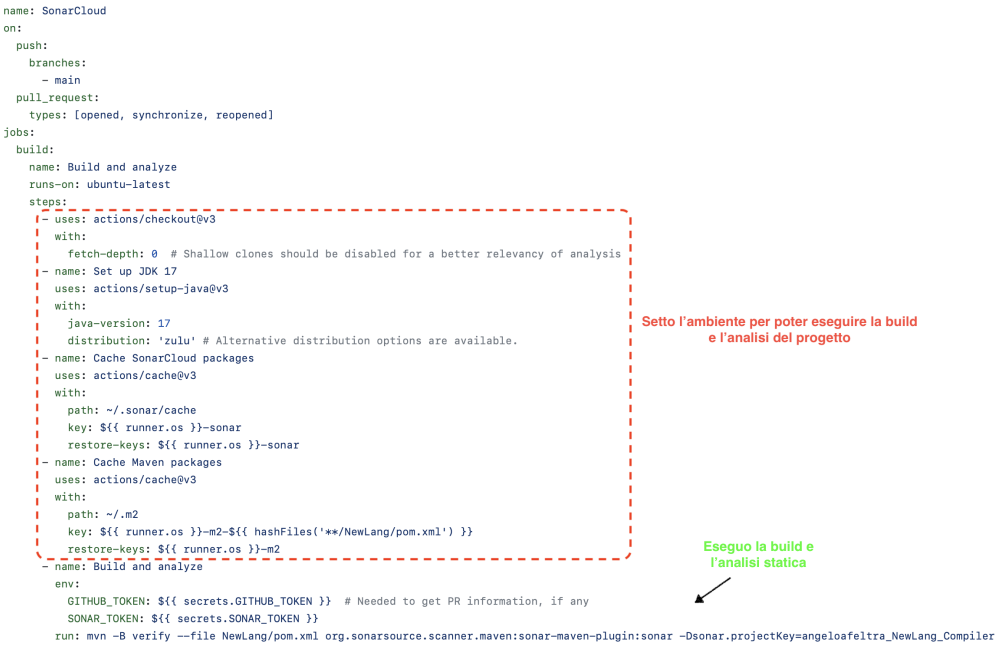


Figure 5: GitHub Action SonarCloud.

L'analisi condotta è stata incentrata principalmente sul servizio di back-end, poiché esso rappresenta il nucleo principale del sistema. SonarCloud analizzando un totale di 5000 linee di codice, ha individuato 541 problematiche, le quali vengono categorizzate in base alla loro gravità secondo le seguenti categorie: Blocker, Critical, Major e Minor.

Per quanto riguarda i problemi legati a Bugs, Vulnerabilità e Security Hotspots, tutte le problematiche individuate sono state risolte, indipendentemente dalla loro categoria di gravità. L'obiettivo è garantire la correttezza e la sicurezza del codice. Nel caso dei code smells, l'attenzione è stata posta principalmente sulla rimozione di quelli classificati come Blocker, Critical e Major. Inoltre, le problematiche che provenivano da classi autogenerate sono state ignorate, in quanto esse si ripresenteranno durante una nuova build.

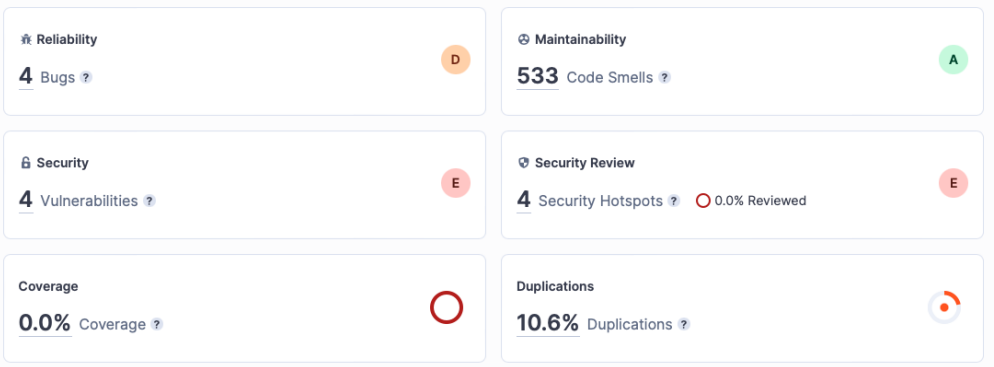


Figure 6: SonarCloud Report.

## 3.1 Bugs

Il primo bug riguarda l'object Random. Al interno del metodo compileFileNewLang(...), era stato istanziato un object Random utilizzato per generare un intero da concatenare al nome del file da compilare. Tuttavia la creazione di un nuovo object Random ogni volta che è necessario un valore casuale è inefficiente e può produrre numeri che non sono casuali a seconda del JDK. Per una migliore efficienza bisogna creare un singolo object Random e riutilizzarlo ogni volta che serve.Pertanto tale bug è stato fixato rendendo l'object Random attributo della classe NewLangController.

Il secondo bug riguarda invece il confronto tra due stringhe. Esso era presente all'interno della classe SemanticVisitor2, in cui si confrontavano due stringhe con l'operatore !=. L'utilizzo di tale operatore non esegue un confronto sul valore ma sulla locazione di memoria, pertanto è stato sistemato eseguendo il confronto tramite il metodo equals().

Gli ultimi due bugs erano presenti all'interno dei metodi compileFileNewLang(...) e visit(ProgramOp programOp) della classe TranslatorVisitor sui metodi createNewFile() e delete(). Tali bugs erano stati segnalati in quanto non era stata prevista nessun'azioni in base ai valori di ritorno di tali metodi. Nel primo caso se il valore ritornato dalla create è falso allora si interrompe la traduzione del codice in quanto lo script C non viene creato correttamente, mentre per quanto rigurada la delete(), se essa dovesse fallire allora si stampa un messaggio d'errore nel Logger.

## 3.2 Vulnerabilità

Sono state identificate quattro vulnerabilità che riguardano una potenziale path-injection, tutte con una root-cause comune dipendente dal metodo compileFileNewLang(...). Questo metodo riceve una richiesta contenente non solo il contenuto del file da compilare, ma anche il suo nome. Il nome del file viene riutilizzato più volte per istanziare un oggetto FILE(), un esempio di utilizzo si trova nella classe TranslatorVisitor, che si occupa della traduzione dello script NewLang in script C. In questo processo, viene generato un file.c il cui nome dipende dal nome del file da compilare. Tuttavia, questo flusso di esecuzione può essere sfruttato come un punto di attacco per sovrascrivere o cancellare file arbitrari, mettendo a rischio il corretto funzionamento del sistema. Per risolvere questo problema, il refactoring applicato consiste nell'ignorare completamente il nome del file inviato tramite la richiesta. Al suo posto, viene generata una stringa randomica a tempo di esecuzione e utilizzata come nome del file. Questo approccio rende impraticabile la path-injection e garantisce la sicurezza del sistema.

## 3.3 Code Smells

Per quanto riguarda i code smell ne sono stati fixati 336, tuttavia per evitare di creare un elenco troppo lungo, di seguito viene riportato il link al summary di SonarCloud. Nel seguente documento saranno discussi solamente quelli piu complessi e data una motivazione per quelli non fixati.
SonarCloud NewLang_Compiler Summary.

Tra i vari code smell risolti, uno in particolare richiedeva tecniche di refactoring più complesse e riguardava la complessità cognitiva dei metodi.

La complessità cognitiva di un metodo indica che il suo flusso d'esecuzione è difficile da comprendere. Questo si verifica quando ci sono numerosi if o cicli annidati, o una combinazione di if e cicli annidati. La tecnica più utilizzata per affrontare la complessità cognitiva di un metodo è l'estrazione di metodi ("Extract Method"). Tuttavia, prima di applicare questa strategia a un metodo, è stato valutata la possibilità di semplificare la complessità cognitiva del flusso di esecuzione, ad esempio rimuovendo if superflui oppure sostituendo if a cascata con uno switch.

Di seguito sono elencati i metodi con una complessità cognitiva elevata e i relativi refactoring.

| Metodo | Strategia di Refctoring |
|---|---|
| compiler.visitors.TranslatorVisitor.visit(ProgramOp programOp) | Extract Method - Sono stati creati 4 metodi, generateListaPrototipi, generaVariabiliGlobali, generaMain e generaFunzioni |
| compiler.visitors.TranslatorVisitor.visit(BodyOp bodyOp) | Semplificazione del flusso - E stato semplificato il flusso riutilizzando il metodo generaVariabiliGlobali |
| compiler.visitors.TranslatorVisitor.visit(ReadOp readOp) | Semplificazione del flusso - E stato semplificato il flusso sostituendo gli if a cascata con uno switch |
| compiler.visitors.TranslatorVisitor.visit(AritAndRelOp aritAndRelOp) | Semplificazione del flusso ed extract method |
| compiler.visitors.SemanticVisitor1.visit(ProgramOp programOp) | Extract Method - Sono stati creati 2 metodi addVariableToScope e addFunToScope |
| compiler.visitors.SemanticVisitor1.visit(BodyOp bodyOp) | Semplificazione del flusso - E stato semplificato il flusso riutilizzando il metodo addVariableToScope |
| compiler.visitors.SemanticVisitor2.visit(CallFunOpStat callFunOpStat) | Extract Method e Semplificazione del flusso |
| compiler.visitors.SemanticVisitor2.visit(CallFunOpExpr callFunOpExpr) | Extract Method e Semplificazione del flusso |
| compiler.visitors.SemanticVisitor2.(AritAndRelOp aritAndRelOp) | Extract Method - Sono stati creati 4 metodi, resultAritmeticOp, resultRelOp, resultStringOp e resultBooleanOp |

Table 1: Refactoring Cognitive Complexity.

Per quanto riguarda i code smell non corretti, escludendo quelli delle classi autogenerate, ne rimangono solo 6 di tipo "Critical" e 23 di tipo "Major", che possono essere raggruppati in due regole:

1. (Java) Fields in a "Serializable" class should either be transient or serializable
2. (Java) Generic exceptions should never be thrown

La prima regola essenzialmente afferma che bisogna rendere una classe (SymbolTable) Serializabile, ma essendo che tale classe non viene mai salvata tale refactoring risulta essere inulte. La seconda regola indica che dovremmo utilizzare eccezioni specifiche anziché eccezioni generiche. Tuttavia, poiché tutte le eccezioni presenti nel progetto in esame estendono la classe "Extends" è stato scelto di ignorare questi code smell.

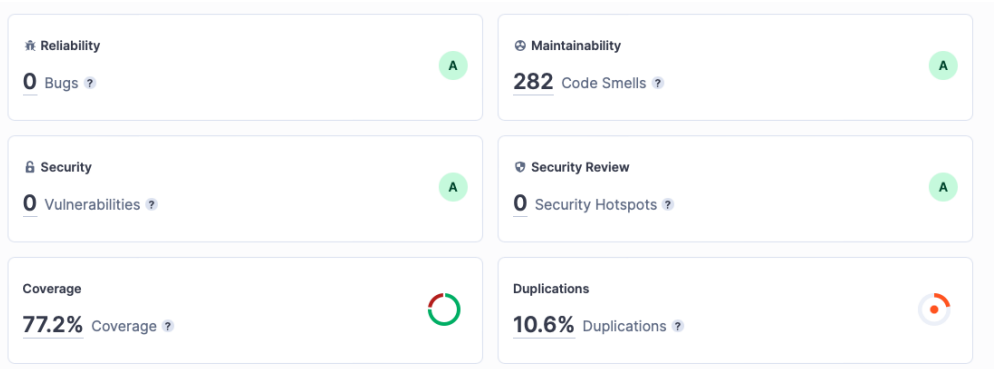


Figure 7: SonarCloud Report Post Analysis.

# 4 EcoCode

Attraverso l'Energy Testing, siamo in grado di valutare l'efficienza energetica di un sistema, identificare le componenti che richiedono maggiori quantità di energia e individuare possibili ottimizzazioni per ridurne il consumo. Il progetto EcoCode ha l'obiettivo di ridurre il consumo energetico di un sistema agendo direttamente sul codice, fornendo una serie di regole da seguire. Esso è stato integrato come plug-in di SonarQube ed è stata effettuata una nuova analisi del progetto. I risultati hanno mostrato un aumento dei "code smells" rispetto alla precedente analisi, indicando la presenza di un debito tecnico in termini di efficienza energetica.

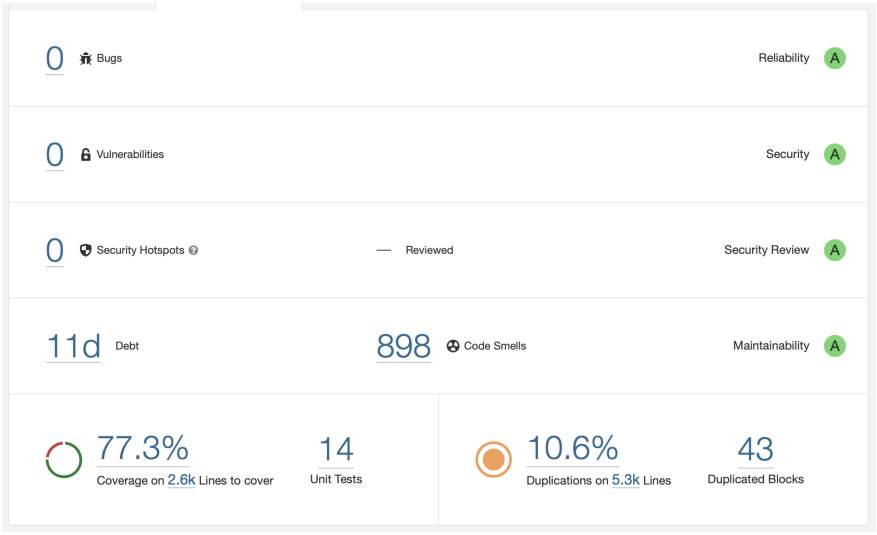


Figure 8: SonarQube Report.

Di tali code smells 601 non rispettano regole di EcoCodo di cui 374 appartengono a classi autogenerate. Tuttavia, è possibile categorizzare questi "code smells" in 10 regole distintive. Nella tabella sottostante, sono riportati il numero di "code smells" associati a ciascuna regola ed il relativo refactoring:

| Regola | Code Smells | Metodo di Refactoring |
|---|---|---|
| Do not unnecessarily assign values to variables | 274 | Rimuovere la variabile non usata |
| Avoid using global variables | 129 | Evitare l'uso di variabili globali sostituendole con variabili locali |
| Avoid multiple if-else statement | 109 | Usare lo statement switch invece degli if a cascata |
| Don't concatenate Strings in loop, use StringBuilder instead | 35 | Eseguire la concanetazione di stringe con StringBuilder |
| Use ++i instead of i++ | 28 | |
| Avoid the use of Foreach with Arrays | 12 | Utilizzare for each su l'object List invece di Array |
| Initialize builder/buffer with the appropriate size | 9 | Inizializzare il buffer con una dimensione |
| Avoid getting the size of the collection in the loop | 2 | Evitare di utilizzare il metodo size() per iterare in un for, ma ottenere la size precedentemente |
| Do not call a function when declaring a for-type loop | 2 | Non utilizzare chiamate di funzioni nella dichiarazione di un ciclo for. |

Table 2: EcoCode CodeSmells.

Sono stati fixati solamente 200 code smells trascurandone molti che derivano dalle prime 3 regole. Questo perchè alcune variabili che venivano segnalate come non usate, in realtà sono passate come parametri ad alcuni metodi, oppure molte varaibili globali sono state consigliate da SonarCloud. Mentre per quanto riguarda la sostituzione di if-else con uno switch, molti if segnalati non risultano essere if a cascata.

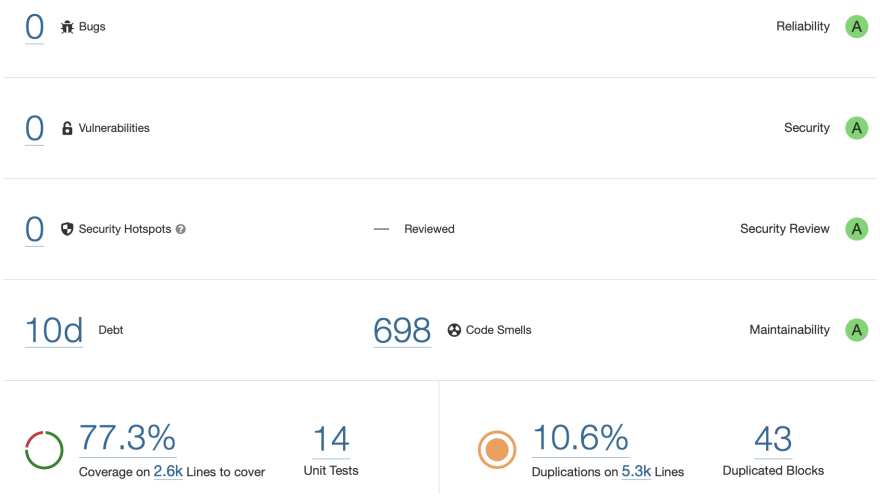


Figure 9: SonarQube Report Post Refactoring.

# 5 Testing

Durante l'analisi del sistema, è stato riscontrato che la suite di test interna non era stata integrata nella build di Maven, di conseguenza, quando veniva eseguito il comando "mvn test", nessun test era eseguito. Per risolvere questa problematica è stato effettuato l'integrazione del plugin Surefire nel file POM. Questo plugin consente l'esecuzione dei test durante la fase di build del progetto.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <configuration>
        <includes>
            <include>testCompiler.SemanticTest.java</include>
            <include>testCompiler.SemanticTestAritmeticOp.java</include>
            <include>testCompiler.SemanticTestAssegnazione.java</include>
            <include>testCompiler.SemanticTestBooleanOp.java</include>
            <include>testCompiler.SemanticTestCallFun.java</include>
            <include>testCompiler.SemanticTestFunzioni.java</include>
            <include>testCompiler.SemanticTestIf.java</include>
            <include>testCompiler.SemanticTestRead.java</include>
            <include>testCompiler.SemanticTestRelationalOp.java</include>
            <include>testCompiler.SemanticTestReturn.java</include>
            <include>testCompiler.SemanticTestStringOp.java</include>
            <include>testCompiler.SemanticTestUnaryOp.java</include>
            <include>testCompiler.SemanticTestVariabili.java</include>
            <include>testCompiler.SemanticTestWhile.java</include>
            <include>testCompiler.TraductorTest.java</include>
        </includes>
    </configuration>
</plugin>
```

Figure 10: Surefire Plugin Configuration.

Inoltre è stato integrato anche il plugin JaCoCo che permette di calcolare la copertura del codice in base ai test eseguiti.

Grazie all'integrazione di questi plugin, ad ogni build del progetto, vengono lanciati automaticamente i test interni e calcolata la rispettiva code coverage.

```xml
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>
    <executions>
        <execution>
            <id>jacoco-initialize</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>jacoco-site</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Figure 11: JaCoCo Plugin Configuration.

**NewLang**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| compiler | | 88% | | 65% | 117 | 328 | 174 | 1.221 | 16 | 52 | 2 | 5 |
| compiler.nodi | | 55% | | 50% | 29 | 67 | 65 | 147 | 20 | 50 | 0 | 5 |
| com.spring.newlang | | 0% | | 0% | 13 | 13 | 53 | 53 | 7 | 7 | 2 | 2 |
| compiler.visitors.semanticVisitor | | 90% | | 78% | 62 | 213 | 65 | 443 | 20 | 77 | 8 | 25 |
| compiler.visitors | | 90% | | 74% | 43 | 155 | 50 | 390 | 3 | 38 | 0 | 1 |
| compiler.nodi.statement | | 67% | | 54% | 25 | 77 | 48 | 162 | 20 | 66 | 0 | 9 |
| compiler.symbolTable | | 52% | | 45% | 24 | 52 | 43 | 102 | 11 | 32 | 0 | 4 |
| compiler.nodi.expr | | 77% | | 50% | 14 | 58 | 21 | 106 | 13 | 56 | 0 | 8 |
| compiler.utilsClass | | 65% | | n/a | 1 | 8 | 6 | 16 | 1 | 8 | 0 | 1 |
| Total | 2.194 of 13.703 | 83% | 289 of 929 | 68% | 328 | 971 | 525 | 2.640 | 111 | 386 | 12 | 60 |

Figure 12: JaCoCo Report.

La suite di test presente all'interno del sistema, garantiva già una branch coverage del 68% ed una statement coverage del 83%, tuttavia grazie all'utilizzo del tool EvoSuite sono stati generati test automatici, con lo scopo di aumentare tali coverage. Il tool è stato utilizzato per la generazione dei test relativi a tutte le classi del package compiler. I test generati sono stati tutti aggiunti alla suite di test ad esclusione delle classi del package compiler.visitors.sematicVisito, in quanto riportavano una coverage inferiore a quella già riaggiunta. Di seguito è riportata la coverage dopo l'introduzione dei test genrati da EvoSuite all'interno della suite di test.

**NewLang**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| compiler | | 87% | | 65% | 117 | 328 | 175 | 1.222 | 16 | 52 | 2 | 5 |
| com.spring.newlang | | 0% | | 0% | 13 | 13 | 53 | 53 | 7 | 7 | 2 | 2 |
| compiler.visitors.semanticVisitor | | 90% | | 78% | 62 | 213 | 65 | 443 | 20 | 77 | 8 | 25 |
| compiler.visitors | | 91% | | 75% | 40 | 155 | 46 | 390 | 2 | 38 | 0 | 1 |
| compiler.symbolTable | | 89% | | 77% | 5 | 52 | 8 | 102 | 0 | 32 | 0 | 4 |
| compiler.nodi.statement | | 94% | | 90% | 4 | 77 | 8 | 162 | 3 | 66 | 0 | 9 |
| compiler.nodi | | 96% | | 100% | 1 | 66 | 6 | 145 | 1 | 50 | 0 | 5 |
| compiler.nodi.expr | | 98% | | 100% | 1 | 58 | 1 | 106 | 1 | 56 | 0 | 8 |
| compiler.utilsClass | | 100% | | n/a | 0 | 8 | 0 | 16 | 0 | 8 | 0 | 1 |
| Total | 1.572 of 13.711 | 88% | 247 of 927 | 73% | 243 | 970 | 362 | 2.639 | 50 | 386 | 12 | 60 |

Figure 13: JaCoCo Report Post EvoSuite.

# 6 Performance Tests

Per l'analisi delle prestazioni del sistema, è stato utilizzato il framework di benchmarking Java Microbenchmark Harness (JMH). Sono stati introdotti appositi metodi di benchmarking relativi alla classe NewLang al fine di valutare le prestazioni del sistema in relazione al numero di linee di codice presenti nello script da compilare.

```java
public class BenchmarkCompiler {

    11 usages  4 inheritors  Angelo Afeltra *
    @State(Scope.Benchmark)
    public static class FileToCompile{

        1 usage
        private SecureRandom random = new SecureRandom();
        Angelo Afeltra
        @Setup
        public void setup(){}

        9 usages  Angelo Afeltra
        public byte[] getFileContent(String path_test_file) {...}

        9 usages  Angelo Afeltra
        public String getFileName() {...}

    }



    10 usages  Angelo Afeltra
    @Benchmark
    @Fork(3)
    @Warmup(iterations = 2)
    @Measurement(iterations = 6)
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MILLISECONDS)
    public void measureAvgTimeCompilerFile1Line(FileToCompile file) throws IOException {
        NewLang newLang=new NewLang();
        newLang.compile(file.getFileContent( path_test_file: "/test_script_performance_testing/Test_1Line.txt"),file.getFileName());
    }
```

Figure 14: JMH Class.

La scelta di tale classe non è stata casuale, poiché rappresenta il nucleo del sistema, includendo il metodo compile(). Questo metodo riceve in input un flusso di byte estratto dallo script proveniente dal servizio di front-end e restituisce lo script compilato. Nel grafico sequente sono riportate le score ottenute, relative ai vari script passati con 1, 10, 50, 100, 250, 500, 750, 1000, 2000 di righe di codice. Da come si puo osservare anche passondo degli script con 2000 righe di codice, che nel contesto d'applicazione di tale sistema risultano essere abbastanza eccessive, il sistema riesce a mantenere delle buone performance, anche se esse risultano essere in leggero aumento.

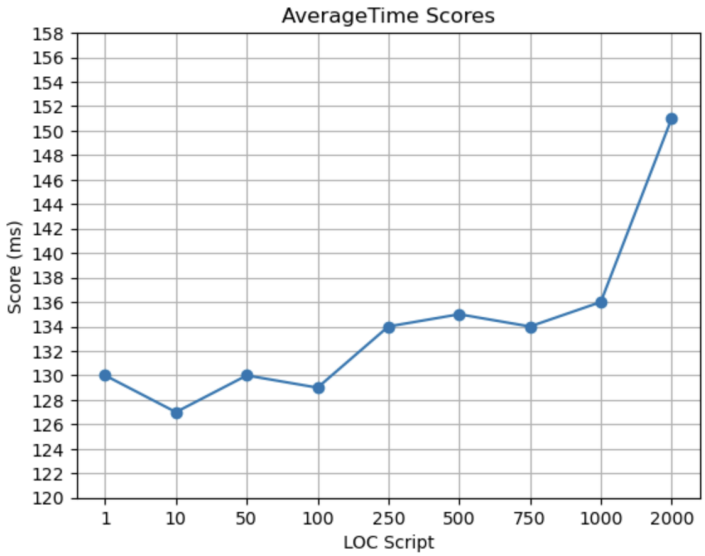


Figure 15: Benchmark AvarageTime.

# 7 Security

Tramite l'utilizzo dei tool FindSecBugs, OWASP DC e OWASP ZAP, sono state individuate le varie vunerabilià presenti all'interno del progetto. Il primo tool FindSecBugs ha rilevato 111 Warning tutti a priorità media. Tali warning sono stati classificati dal tool in:

| Type Warnings | Warnings |
|---|---|
| Malicious code vulnerability Warnings | 92 |
| Security Warnings | 4 |
| Dodgy code Warnings | 15 |

Table 3: FindSecBugs Report.

Per quanto riguarda la prima categoria le cause di tali warnings sono solamente due e risiedono nei metodi getter e setter presenti all'interno del codice. Essi o non restituiscono una copia dell'object restituito oppure non memorizzano una copia dell'object ricevuto.

Per i security warnings abbiamo 3 cause differenti, command injection sul process builer presente nella classe NewLang, CRLF Injection sul logger e Path Traversal sui metodi del object File.

I warnings di tipo Dodgy code invece sono dipendenti dal cast di una collezione astratta come List ad un object ArrayList. Tuttavia bisogna dire che allo stato originale del sistema essi non erano presenti, in quanto sono stati introdotti dopo l'analisi di SonarCloud. Esso segnalava come code smell metodi che avevano come parametri un ArrayList, consigliando di utilizzare List.

Per maggiore chiarezza di seguito è riportato il report restituito da FindSecBugs. FindSecBugs Report

Il tool OWASP DC non ha rilevato nessuna vulnerability tranne alcune, derivanti EcoCode (OWASPDC Report ), mentre OWASP ZAP, ha rilevato 12 Allert classificati in:

| Risk Level | Number of Alerts |
|---|---|
| High | 0 |
| Medium | 3 |
| Low | 5 |
| Informational | 4 |
| False Positives: | 0 |

**Alerts**

| Name | Risk Level | Number of Instances |
|---|---|---|
| Absence of Anti-CSRF Tokens | Medium | 2 |
| Content Security Policy (CSP) Header Not Set | Medium | 2 |
| Missing Anti-clickjacking Header | Medium | 2 |
| Application Error Disclosure | Low | 1 |
| Cookie without SameSite Attribute | Low | 1 |
| Information Disclosure - Debug Error Messages | Low | 1 |
| Permissions Policy Header Not Set | Low | 3 |
| X-Content-Type-Options Header Missing | Low | 4 |
| Loosely Scoped Cookie | Informational | 2 |
| Modern Web Application | Informational | 2 |
| Non-Storable Content | Informational | 1 |
| Storable and Cacheable Content | Informational | 6 |

Figure 16: OWASPZAP Allert.

Essi indicano tutte vulnerabilità differenti come la mancanza di un token Anti-CSRF sul form per il cariacmento del file da compilari oppure la mancanza di Content Security Policy per evitare Cross Site Scripting (XSS). Oppure ad esempio l'alerts Non-Storable Content, indica che eventuali errori o risposte del sistema non sono salvate all'interno della cache per poterne ottimizzare le prestazioni.

Per maggiore chiarezza di seguito è riportato il report restituito da OWASPZAP. OWASPZAP Report

# 8 Extra - Kubernetes

Kubernetes è un alternativa a docker compose per la gestione di servizi multicontainer. Esso è un sistema open-source di orchestrazione dei contenitori che facilita la gestione, la scalabilità e la distribuzione delle applicazioni containerizzate.

Grazie all'utilizzo di Minikube è stato possibile creare un ambiente Kubernetes funzionante in locale, in modo da poter testare il sistema in un ambiente simile a quello di un cluster Kubernetes reale.

Pertendo dalle immagini di docker sono stati creati i rispettivi deployment tramite il comando *kubectl create deployment*:

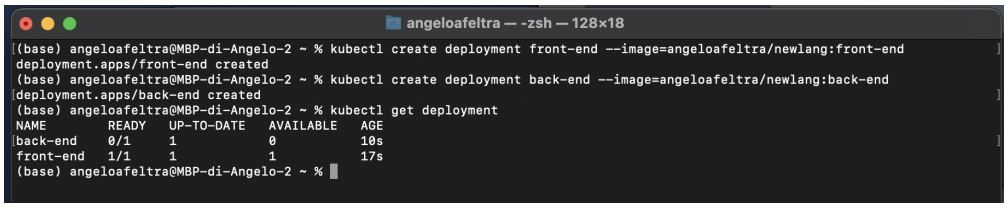


Figure 17: Kubectl Create Deployment.

Per ogni deployment kubernetes crea un pods, in quanto il replication set di default risulta essere 1. Tuttavia tramite il comando *kubectl scale* è possibile aumentare il replication set e sfruttare cosi la scalabilità orizzontale di Kubernetes:

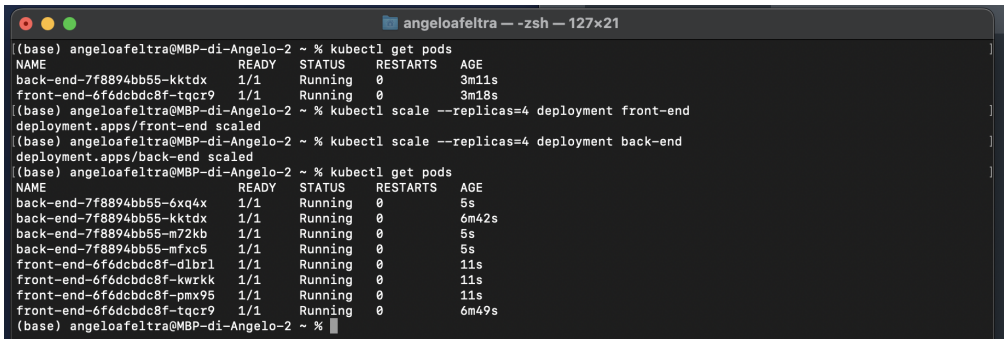


Figure 18: Kubectl scale.

Creati i vari deployment per poter connettersi ad essi devono esere esposti creando quindi dei services. Kubernetes mette a disposizione diverse tipologie di services: ClusterIp, NodePort, LoadBalancer e ExternalName. Nel contesto di tale sistema per poter permettere la connessione agli utenti solamente al servizio di front-end il quale comunicherà in maniera interna al container con il servizio di back-end, per il primo sarà creato un services di tipo NodePort, mentre per il secondo un services ClusterIP:

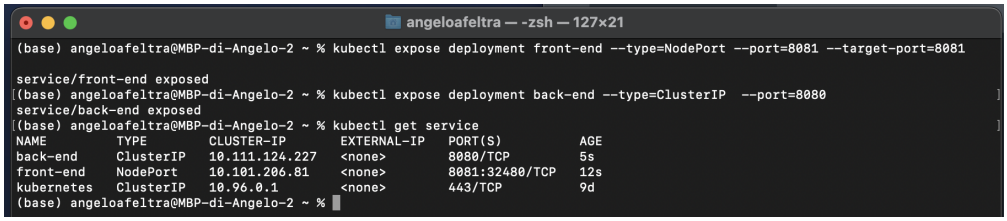


Figure 19: Kubectl expose.

Tuttavia tali passaggi possono essere semplificati sfruttando il comando *kompose convert*, il quale dato un file docker compose genera diversi YAML, dove ognuno di esso rappresenta una risorsa Kubernetes. Utilizzando poi il comando *kubectl apply* si andranno a creare tutte le risorse descritte precedentemente.

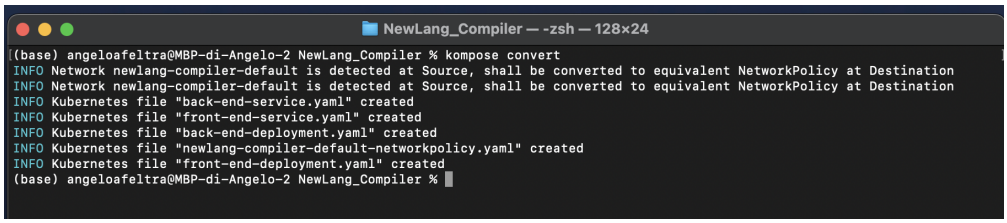


Figure 20: kompose convert.

I vari YAML generati possono essere recuperati utilizzando il seguente link: YAML Kubernetes

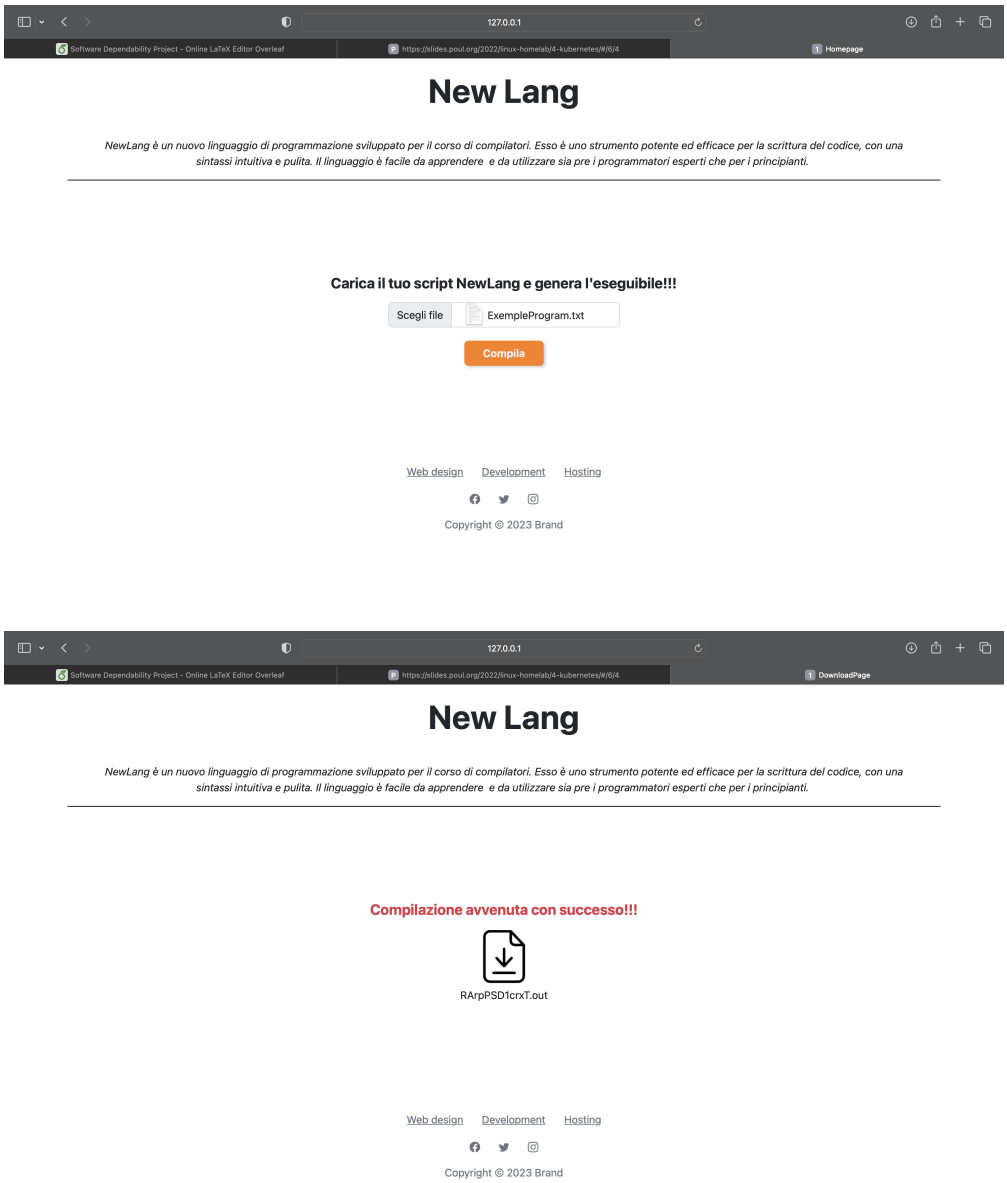


Figure 21: Simulazione Kubernetes.