UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Progettazione e Sviluppo di un Bot per l'Identificazione di Test Smell: Il Progetto TOTEM

RELATORE

Prof. **Fabio Palomba**

Università degli Studi di Salerno

CANDIDATO

Simone Tartaglia

Matricola: 0512106347

Anno Accademico 2021-2022

"È meglio chiedere la carità piuttosto che essere incolti: gli uni, infatti, mancano di ricchezze; gli altri, di umanità."

- Aristippo di Cirene

Sommario

In un periodo in cui l'automazione assume un ruolo sempre più importante all'interno dello sviluppo di progetti software, e la scelta dei tool a supporto degli sviluppatori rappresenta una parte fondamentale delle prime fasi della progettazione, divengono di notevole importanza le applicazioni che possono avere componenti di intelligenza artificiale come parte di tali strumenti. Ma sempre durante lo sviluppo, è innegabile l'importanza rappresentata dal testing e dalle risorse investite in esso, e non è raro imbattersi in framework pensati specificatamente per semplificare questo processo. Nella scrittura del codice di produzione, quindi destinato allo sviluppo di quello che poi diventerà il prodotto finale, sono stati individuati i cosiddetti *Code Smell*, ovvero cattive pratiche di programmazione che possono inficiare sulla qualità del codice, rallentandone quindi la scrittura, la rielaborazione, e in generale ostacolandone lo sviluppo, influendo negativamente sulla qualità del software rilasciato. Recenti indagini hanno tuttavia individuato un fenomeno analogo per quanto riguarda il codice pensato per il testing, ovvero i *Test Smell*. Nonostante ciò, è evidente la mancanza di strumenti finalizzati alla rilevazione e correzione di queste anomalie, nonostante la già citata importanza del testing all'interno dello sviluppo software. In tale contesto nasce il progetto **TOTEM** (Module for the De**T**ecti**O**n of **TE**st S**M**ells): un tool per l'identificazione dei suddetti Test Smell. Abbiamo deciso di dare vita a questo strumento come plugin per l'IDE INTELLIJ IDEA, con l'obbiettivo di implementare tutte le funzioni necessarie per eseguire un'accurata rilevazione dei tre tipi di Test Smell attualmente identificati: **General Fixture**, **Eager Test** e **Lack of Cohesion**. Questo modulo si integra nell'interfaccia grafica dell'IDE e notifica l'utente delle anomalie rilevate tramite essa, senza l'ausilio di strumenti esterni. Dal punto di vista tecnico, siamo riusciti a raggiungere questo risultato e a superare i limiti imposti dalle API di INTELLIJ IDEA dividendo il progetto in due moduli: il primo è sviluppato sfruttando le risorse messe a disposizione dall'IDE, responsabile della lettura del codice da analizzare, della ristrutturazione in oggetti contenenti esclusivamente i dati necessari all'analisi, e della comunicazione col secondo modulo; quest'ultimo è stato sviluppato sfruttando il framework SPRING BOOT ed è responsabile invece dell'analisi dei dati ricevuti, nonché dell'invio dei risultati ottenuti dall'analisi al primo modulo.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Il testing rappresenta una parte estremamente importante dello sviluppo software [1]. Tipicamente, fino al 50% delle risorse vengono investite in questa fase, cruciale nella determinazione dell'effettiva qualità del software. E con l'aumento esponenziale dell'importanza del supporto post-release, è naturale che vengano investite ulteriori risorse nella valutazione delle feature aggiunte in un secondo momento.

Il testing del software ricopre un ampio spettro di tecniche applicabili, accomunate da un unico principio ben definito: trovare quanti più errori possibili all'interno del software. Un buon test, quindi, è quello che ha un'elevata possibilità di rilevare falle precedentemente sconosciute. Esistono appunto diverse tecniche di testing, in particolare si individuano: **Testing di Unità**, per mettere alla prova una singola componente del software in modo indipendente dalle altre; **Testing di Integrazione**, per verificare la corretta integrazione di due o più unità distinte; **Testing di Sistema**, per accertarsi del corretto funzionamento del sistema finale; **Testing di Accettazione**, eseguito al termine dello sviluppo, prima del rilascio pubblico del software, per assicurarsi che sia perfettamente conforme alle aspettative dell'utenza.

Per eseguire tali prove, gli sviluppatori eseguono i suddetti applicativi sfruttando particolari insiemi di condizioni e variabili, notoriamente chiamati **Test Case**, per analizzare il comportamento del software in specifiche situazioni, e assicurarsi che sia conforme rispetto a

quanto previsto. Da un punto di vista puramente pratico, questi Test Case consistono in delle classi che simulano l'esecuzione del software, inserendo un particolare input, e verificando che l'output sia quello desiderato. Tali classi possono essere sviluppate sia manualmente, quindi scrivendo "a mano" il codice che eserciterà il software, oppure facendosi assistere da specifici strumenti pensati per tale scopo, come ad esempio JUNIT. In ogni caso, c'è il rischio che queste classi possano sfuggire a quelle che sono le convenzioni imposte dagli sviluppatori per la buona scrittura del codice. Pur non costituendo un errore, è innegabile che tali imprecisioni possono ostacolare il mantenimento di queste classi [2]. Tale problematica assume una rilevanza maggiore se consideriamo che rischia di colpire quella che è la fase dello sviluppo in cui viene investita la maggioranza del budget, rischiando di far lievitare ulteriormente i costi di sviluppo. Queste anomalie sono state denominate **Test Smell**, e sarà oggetto del nostro studio lo sviluppo di uno strumento in grado di rilevare automaticamente tali irregolarità nella scrittura del codice.

Il codice sorgente di TOTEM, così come altre risorse utili, sono disponibili pubblicamente su GITHUB. [1]

1.2 Motivazioni e Obiettivi

Negli ultimi anni, sta assumendo una rilevanza sempre maggiore l'automazione di processi ripetitivi all'interno dello sviluppo software, e sempre più risorse vengono dedicate a tale scopo. In particolare, sta prendendo sempre più piede l'utilizzo di **Bot** [3], ovvero moduli di intelligenza artificiale che assistono gli sviluppatori nell'esecuzione di questi task, nonché nel fornire automaticamente risorse utili ai membri del team di sviluppo. In realtà, i servizi offerti dai Bot non sono una novità assoluta, ma il modo in cui sono presentati agli sviluppatori, tipicamente per mezzo della UI dell'ambiente di sviluppo, rappresenta un notevole passo in avanti per la semplificazione di determinati passaggi e l'accelerazione del processo produttivo.

Come accennato nella sezione precedente, la parte più importante e dispendiosa della produzione del software è il testing, e come naturale conseguenza di tutto ciò, vi è un'ampia gamma di Bot dediti a tale scopo. Esistono, ad esempio, Bot in grado di eseguire automaticamente operazioni di analisi e refactoring del codice, per fare in modo che rispettino le convenzioni imposte nel mondo dello sviluppo. Tuttavia, questa offerta non sembra riconoscere l'esistenza dei Test Smell, che nonostante affliggano una parte così importante

[1] Link alla repository di TOTEM: `https://github.com/drybonez01/TOTEM`

dello sviluppo, restano un argomento estremamente di nicchia e poco conosciuto dagli sviluppatori.

In questo contesto, abbiamo deciso di proporre una nostra soluzione a questo problema. Nasce quindi **TOTEM** (Module for the De**T**ecti**O**n of **TE**st S**M**ells), un innovativo strumento per la rilevazione di Test Smell, concepito come plugin per l'IDE INTELLIJ IDEA, in grado di rilevare automaticamente e notificare lo sviluppatore di eventuali anomalie rilevate nel codice, il tutto sfruttando un'interfaccia estremamente semplice ed intuitiva.

Con questo strumento, ci focalizzeremo sulla rilevazione di tre particolari tipologie di Test Smell, ovvero:

- **Eager Test** [4], quando un metodo di una classe di test esercita più di un metodo di una classe di produzione;
- **General Fixture** [4], quando una classe di test inizializza un'eccessiva quantità di variabili di istanza, e i restanti metodi sfruttano solo una piccola parte di esse;
- **Lack of Cohesion** [5], quando le variabili utilizzate dai metodi di una classe di test variano eccessivamente di metodo in metodo.

Dal punto di vista strutturale, invece, il nostro obiettivo principale consiste nell'integrare tale modulo all'interno dell'interfaccia grafica di INTELLIJ, sfruttando quindi le API messe a disposizione dai suoi sviluppatori. Tuttavia, tali API sono estremamente soggette a cambiamenti e rischiano di rendere instabile il nostro strumento, specialmente in virtù della compatibilità con le future versioni dell'IDE. Per sopperire a tale problematica, abbiamo deciso di suddividere le funzionalità di TOTEM in due moduli distinti:

- Il primo modulo sarà il plugin vero e proprio, sviluppato sfruttando le API di INTELLIJ IDEA, e sarà integrato nell'interfaccia dell'IDE. Esso si limiterà ad eseguire lo stretto necessario per estrapolare i dati necessari per la rilevazione dei Test Smell dal codice scritto dallo sviluppatore, per poi inviare quanto trovato al secondo modulo, nonché visualizzare i risultati dell'analisi;
- La seconda componente, invece, è rappresentata da un modulo sviluppato in SPRING BOOT, ed è responsabile della rilevazione di Test Smell partendo dai dati ricevuti dall'IDE. Esso quindi analizza queste informazioni, e restituisce quanto individuato al primo modulo. Così facendo, questa componente resta indipendente dall'interfaccia dell'IDE, evitando problemi di compatibilità dovuti ai suoi continui aggiornamenti.

1.3 Risultati ottenuti

Alla fine dello sviluppo, abbiamo ottenuto un tool conforme a quanto previsto. Esso è appunto costituito da due moduli, uno sviluppato come componente dell'IDE INTELLIJ IDEA, e l'altro come componente a se stante sviluppata in SPRING BOOT.

Per attuare la rilevazione, è semplicemente necessario selezionare l'opzione apposita dalla barra degli strumenti di INTELLIJ. Questo darà inizio al processo di analisi del codice e invio dei dati estrapolati al secondo modulo, il quale analizzerà i dati ottenuti e restituirà l'output all'IDE. Infine, verrà visualizzato un semplice messaggio a schermo indicante quali tipi di Test Smell sono stati identificati.

La rilevazione è in grado di identificare i tre Smell previsti inizialmente, quindi Eager Test, General Fixture e Lack of Cohesion. Tuttavia, il nostro strumento si limita semplicemente a rilevare la presenza di tali Smell, e per limitazioni tecniche è stata abbandonata l'idea di offrire un refactoring automatico del codice. Peraltro, nel nostro caso non esiste una correzione unica e precisa per le problematiche individuate, per cui sarebbe ridondante offrire una soluzione che molto probabilmente non sarebbe conforme alle aspettative dello sviluppatore.

1.4 Struttura della tesi

Innanzitutto, partiremo con una breve introduzione, contenuta nel paragrafo appena concluso, esponendo quella che è stata l'idea alla base del nostro lavoro, una panoramica su quelli che erano gli scopi che ci siamo imposti, e un'analisi ad alto livello del risultato del nostro lavoro. Dopodiché, andremo a vedere lo stato dell'arte, quindi la letteratura e la strumentazione già disponibile, relativa all'oggetto del nostro studio, nonché come esso si differenzia da quanto già esistente. A questo punto andremo a vedere le metodologie applicate, come abbiamo sviluppato il nostro strumento e quali fasi sono state attraversate durante lo sviluppo per arrivare allo stato attuale. Infine, andremo a vedere i risultati ottenuti, quanto essi siano fedeli al piano iniziale, e soprattutto quali sono le prospettive future per quanto esposto nella nostra indagine.

CAPITOLO 2

Stato dell'arte

In questo capitolo, andremo ad illustrare lo stato dell'arte e ad esaminare la letteratura già esistente di quanto trattato nel nostro studio. Andremo quindi ad effettuare una panoramica sui cosiddetti Test Smell, andando a comprendere innanzitutto in cosa consistono, quali sono le conoscenze attuali in merito, nonché quali strumenti attualmente disponibili ne permettono l'individuazione, partendo prima da una breve disamina sul concetto alla base dei Test Smell, ovvero i Code Smell.

2.1 Code Smell

Nell'ingegneria del software, è stato introdotto il concetto di Code Smell [6] per identificare anomalie nel codice, che non rappresentano necessariamente un errore, ma che possono essere sintomatiche di problemi più seri e meno evidenti nella stesura del codice stesso, rischiando di influire negativamente sullo sviluppo e soprattutto sulla qualità del prodotto finale. Un Code Smell quindi non è un bug, non compromette il funzionamento dell'applicativo, ma semplicemente esime da quelle che possono essere riconosciute come le giuste convenzioni all'interno della programmazione. Pertanto, non vi è una definizione universale di Smell, ma si identificano come tali quelle anomalie che, empiricamente, hanno dimostrato di essere sintomi delle problematiche sopra descritte, e la cui specifica è totalmente a discrezione della convenzione utilizzata.

2.2 Test Smell

Un fenomeno analogo è stato rilevato in relazione ai metodi di test, prendendo il nome di Test Smell [2]. All'interno dello sviluppo software, il testing rappresenta infatti una componente fondamentale dell'intero processo, nonché quella a cui viene destinata la maggior parte del budget. Durante la produzione, gli sviluppatori sfruttano classi sviluppate appositamente per il testing delle classi di produzione, ovvero quelle contenenti il codice sorgente del prodotto finale. Chiaramente, con l'evoluzione del software tali classi di produzione subiscono continui stravolgimenti, riscritture, nonché aggiunte di funzioni varie, e purché il loro funzionamento possa essere sottoposto ai giusti controlli da parte delle classi di test, è necessario che queste ultime vengano aggiornate costantemente ad ogni nuova versione del sistema.

Non è raro, tuttavia, che tali classi sfuggano a quelle che sono le convenzioni imposte all'interno della programmazione, causando complicanze nella lettura e nel mantenimento del codice. Tali problematiche prendono appunto il nome di Test Smell, la cui identificazione, nonché lo sviluppo di strumenti in grado di rilevarli automaticamente, costituiranno l'oggetto principale del nostro studio.

Recenti indagini, hanno portato all'individuazione di diversi tipi di Test Smell [7]. I principali Smell individuati sono:

- **Assertion Roulette**: quando una serie di `assert` in un metodo di test non contengono alcuna spiegazione sul loro funzionamento. Nel seguente estratto, un esempio di codice affetto da Assertion Roulette:

```
1   @MediumTest
2   public void testCloneNonBareRepoFromLocalTestServer()
3       throws Exception {
4       Clone cloneOp = new Clone(false,
5           integrationGitServerURIFor("small-repo.early.git"),
6           helper().newFolder());
7
8       Repository repo = executeAndWaitFor(cloneOp);
9
10      assertThat(repo,
11          hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
12
13      File readmeFile = new File(repo.getWorkTree(), "README");
14      assertThat(readmeFile, exists());
15      assertThat(readmeFile, ofLength(12));
16  }
```

- **Conditional Test Logic**: un metodo di test dovrebbe limitarsi ad esercitare i metodi di produzione, simulando l'inserimento di un input. Quindi, non dovrebbero contenere istruzioni condizionali, e questo smell identifica appunto i metodi che violano questa condizione. Di seguito, un esempio di questo tipo di Smell:

```
1   @Test
2   public void testSpinner() {
3       for (Map.Entry entry : sourcesMap.entrySet()) {
4           String id = entry.getKey();
5           Object resultObject = resultsMap.get(id);
6           if (resultObject instanceof EventsModel) {
7               ...
8           }
9       }
10  }
```

- **Constructor Initialization**: idealmente, i metodi di test NON dovrebbero avere un costruttore, con l'inizializzazione delle variabili di istanza che dovrebbe avvenire nel metodo `setUp()`. Questo Smell esiste appunto per avvertire gli sviluppatori ignari dell'esistenza di questa condizione. Un esempio di Constructor Initialization:

```
1   public class TagEncodingTest extends BrambleTestCase {
2           private final CryptoComponent crypto;
3           private final SecretKey tagKey;
4           private final long streamNumber = 1234567890;
5
6           public TagEncodingTest() {
7                   crypto = new CryptoComponentImpl(
8                       new TestSecureRandomProvider());
9                   tagKey = TestUtils.getSecretKey();
10          }
11
12          ...
13  }
```

- **Default Test**: quando una classe di test mantiene il nome di default generato dall'IDE, violando quindi le convenzioni sulla nomenclatura delle classi di test. Qui di seguito, un esempio di Default Test:

```
1   public class ExampleUnitTest {
2       @Test
3           public void addition_isCorrect() throws Exception {
```

```
4                       assertEquals(4, 2 + 2);
5               }
6       }
```

- **Duplicate Assert**: anomalia che si verifica quando un metodo di test contiene più di un assert con gli stessi parametri. Qui di seguito, un esempio di codice affetto da Duplicate Assert:

```
1       @Test
2       public void testXmlSanitizer() {
3           boolean valid = XmlSanitizer.isValid("Fritz-box");
4           assertEquals("Minus is valid", true, valid);
5           System.out.println("Minus test - passed");
6
7           valid = XmlSanitizer.isValid("Fritz-box");
8           assertEquals("Minus is valid", true, valid);
9           System.out.println("Minus test - passed");
10      }
```

- **Eager Test**: tale problematica insorge quando un metodo di una classe di test esercita più di un metodo di una classe di produzione, complicando la comprensibilità del metodo stesso. Vediamo un esempio di questo Smell:

```
1       @Test
2       public void NmeaSentence_GPGSA_ReadValidValues(){
3           NmeaSentence nmeaSentence = new NmeaSentence("$GPGSA,A,3,04,05,,
4               09,12,,,24,,,,,2.5,1.3,2.1*39");
5           assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(),
6               is("2.5"));
7           assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(),
8               is("1.3"));
9           assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(),
10              is("2.1"));
11      }
```

- **Empty Test**: quando un metodo di test non contiene alcuno statement, "sporcando" inutilmente il codice. Possono presentarsi quando tali metodi vengono creati per fini di debug, ma poi il codice contenuto in essi viene commentato o cancellato poiché ha esaurito il suo scopo. Qui di seguito, un esempio di Empty Test:

```
public void testCredGetFullSampleV1() throws Throwable {
//        ScrapedCredentials credentials =
//            innerCredTest(FULL_SAMPLE_v1);
//        assertEquals("p4ssw0rd", credentials.pass);
//        assertEquals("user@example.com",credentials.user);
}
```

- **Exception Handling**: si verifica quando il superamento o il fallimento di un test dipendono dalla gestione di un'eccezione da parte del codice di produzione. Un esempio di Exception Handling:

```
@Test
public void realCase() {
    Point p34 = new Point("34", 556506.667, 172513.91, 620.34, true);
    Point p45 = new Point("45", 556495.16, 172493.912, 623.37, true);
    Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
    Abriss a = new Abriss(p34, false);
    a.removeDAO(CalculationsDataSource.getInstance());
    a.getMeasures().add(new Measure(
        p45, 0.0, 91.6892, 23.277, 1.63));
    a.getMeasures().add(new Measure(
        p47, 281.3521, 100.0471, 108.384, 1.63));

    try {
        a.compute();
    } catch (CalculationException e) {
        Assert.fail(e.getMessage());
    }
}
```

- **General Fixture**: una classe di test è affetta da questa anomalia quando presenta una dichiarazione eccessivamente generica delle variabili di istanza, ovvero quando i metodi di test sfruttano solo una parte delle variabili inizializzate nel metodo di `setUp`. Un esempio di General Fixture può essere il seguente:

```
protected void setUp() throws Exception {
    assetManager = getInstrumentation().getContext().getAssets();
    certificateFactory = CertificateFactory.getInstance("X.509");

    infoDebianTestCA = loadCertificateInfo("DebianTestCA.pem");
    infoDebianTestNoCA = loadCertificateInfo("DebianTestNoCA.pem");
    infoGTECyberTrust =
        loadCertificateInfo("GTECyberTrustGlobalRoot.pem");
```

```
9
10          // user-submitted test cases
11              infoMehlMX = loadCertificateInfo("mehl.mx.pem");
12      }
13
14      public void testIsCA() {
15          assertTrue(infoDebianTestCA.isCA());
16          assertFalse(infoDebianTestNoCA.isCA());
17          assertNull(infoGTECyberTrust.isCA());
18
19          assertFalse(infoMehlMX.isCA());
20      }
```

- **Ignored Test**: simile all'Empty Test, si verifica quando un metodo viene esplicitamente ignorato (per mezzo dell'annotazione `@Ignore`) ma è comunque presente nella classe di test. Di seguito, un frammento di codice affetto da Ignored Test:

```
1       @Ignore("disabled for now as this test is too flaky")
2       public void peerPriority() throws Exception {
3               final List addresses = Lists.newArrayList(
4                   new InetSocketAddress("localhost", 2000),
5               new InetSocketAddress("localhost", 2001),
6               new InetSocketAddress("localhost", 2002)
7               );
8           peerGroup.addConnectedEventListener(connectedListener);
9       }
```

- **Lazy Test**: Smell che si manifesta quando più metodi di test invocano lo stesso metodo della classe di produzione. Un esempio di Lazy Test:

```
1       @Test
2       public void testDecrypt() throws Exception {
3           ...
4           assertEquals("Testing simple decrypt",expectedResult,
5               Cryptographer.decrypt(enfileData, "test"));
6       }
7
8       @Test
9       public void testEncrypt() throws Exception {
10          ...
11          String decrypt = Cryptographer.decrypt(encrypted, "test");
12          assertEquals(xml, decrypt);
13      }
```

- **Magic Number Test**: si verifica quando un assert contiene al suo interno un valore numerico come parametro, senza indicarne il significato, piuttosto che una variabile o una costante, come mostrato di seguito:

```
1     @Test
2     public void testGetLocalTimeAsCalendar() {
3         Calendar localTime = calc.getLocalTimeAsCalendar(
4             BigDecimal.valueOf(15.5D), Calendar.getInstance());
5         assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
6         assertEquals(30, localTime.get(Calendar.MINUTE));
7     }
```

- **Mystery Guest**: ambiguità che si verifica quando una classe di test sfrutta risorse esterne, complicandone dunque il mantenimento. Nel seguente estratto, un esempio di Mystery Guest:

```
1     public void testPersistence() throws Exception {
2         File tempFile = File.createTempFile("systemstate-", ".txt");
3         ...
4     }
```

- **Redundant Print**: il metodo di test contiene una o più chiamate a una funzione per stampare una stringa sullo standard output, probabilmente per fini di debug, ma che non sono state mai rimosse. Un esempio di codice che causa Redundant Print:

```
1     @Test
2     public void testTransform10mNEUAndBack() {
3         Leg northEastAndUp10M = new Leg(10, 45, 45);
4         Coord3D result =
5             transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
6         System.out.println("result = " + result);
7         ...
8     }
```

- **Redundant Assertion**: Smell che si verifica quando un metodo contiene un assert che si rivela sempre vero. Di seguito, un esempio di Redundant Print:

```
1     @Test
2     public void testTrue() {
```

```
3         assertEquals(true, true);
4     }
```

- **Resource Optimism**: se un metodo estrae dati da una fonte esterna (come un file), dando per scontato che tale fonte esista, si dice che tale metodo è affetto da Resource Optimism. Nel seguente codice, illustriamo un esempio di questo Test Smell:

```
1     @Test
2     public void saveImage_noImageFile_ko() throws IOException {
3     File outputFile =
4         File.createTempFile("prefix", "png", new File("/tmp"));
5     ProductImage image =
6         new ProductImage("01010101010101",
7             ProductImageField.FRONT, outputFile);
8     ...
9     }
```

- **Sensitive Equality**: nell'evenienza in cui un'uguaglianza venga verificata per mezzo di un metodo `toString`, quindi riducendo i suoi parametri ad una rappresentazione testuale, generalmente poco rappresentativa del contenuto del parametro. Un esempio di Sensitive Equality è il seguente:

```
1     @Test
2     public void test1() throws UnknownHostException {
3         String peersPacket = "...";
4         byte[] payload = Hex.decode(peersPacket);
5         byte[] ip = decodeIP4Bytes(payload, 5);
6         assertEquals(InetAddress.getByAddress(ip).toString(),
7             ("/54.204.10.41"));
8     }
```

- **Sleepy Test**: Smell che identifica tutti quei casi in cui, in un metodo di test, viene effettuata una chiamata a un metodo bloccante, ovvero che pone in uno stato di stallo il thread su cui viene eseguito il codice, come mostrato qui di seguito:

```
1     public void testEdictExternSearch() throws Exception {
2             Thread.sleep(500);
3     }
```

- **Unknown Test**: si dice che un metodo di test è affetto da Unknown Test se non contiene alcun assert. Nel seguente codice, un esempio di questo Test Smell:

```
1     @Test
2     public void hitGetPOICategoriesApi() throws Exception {
3         POICategories poiCategories = apiClient.getPOICategories(16);
4         for (POICategory category : poiCategories) {
5           System.out.println(category.name() + ": " + category);
6         }
7     }
```

Empiricamente, è stato dimostrato come i Test Smell possano incidere sul mantenimento e la leggibilità delle classi di test [2]. Nonostante ciò, tale problema è attualmente estremamente sottovalutato all'interno dello sviluppo software. Anche per questo, è difficile trovare una convenzione generalmente accettata che possa identificare queste problematiche. Una conoscenza adeguata sul tema permetterebbe di comprenderne al meglio la natura e, soprattutto, aiuterebbe a sviluppare strumenti che ne consentano l'individuazione.

2.2.1 Strumenti per la rilevazione di Test Smell

Per ciò che concerne l'identificazione di Test Smell, tuttavia, oltre ad esserci pochissimi tool disponibili, essi sono estremamente di nicchia, e difficilmente raggiungono una platea di sviluppatori sufficientemente ampia.

Uno degli strumenti attualmente in circolazione è **tsDetect** [8], un applicativo open-source basato su un'interfaccia a riga di comando, disponibile come file `jar` autonomo, quindi non è legato ad uno specifico IDE ed è utilizzabile per qualsiasi progetto JAVA. Tale strumento è caratterizzato da una serie di moduli aggiuntivi che permettono di automatizzare il processo di ricerca, effettuando un'analisi dei file per rilevare le classi di test tra tutte le classi del progetto, per poi associare una classe di produzione ad ognuna di esse. Per quanto riguarda la rilevazione di test smell, essa avviene in forma indipendente per ogni anomalia rilevata, riportando poi in un file CSV l'esito della ricerca, ovvero un booleano che indica se è stato rilevato lo smell in questione, per ognuno degli smell rilevabili. Tra i più importanti possiamo notare *Assertion Roulette*, *Duplicate Assert*, *Eager Test* e *Mystery Guest*.

Alternativamente, è disponibile **VITRuM** [9], un plugin per l'IDE INTELLIJ IDEA, in grado di visualizzare metriche riguardanti il testing, semplificando la diagnosi di problemi nel codice di produzione. In particolare, esso è responsabile del monitoraggio di **metriche strutturali**, ovvero informazioni sulla qualità dei casi di test, **metriche dinamiche** con dati

sull'efficacia dei metodi di test, nonché appunto della rilevazione di sette tipi di **Test Smell** (*Assertion Roulette*, *Eager Test*, *General Fixture*, *Mystery Guest*, *Resource Optimism*, *Sensitive Equality* e *Indirect Testing*).

Un altro tool utile per la rilevazione di Test Smell è **TASTE** (**Textual AnalySis for Test smEll detection**) [10]. Esso si concentra sulla rilevazione di tre tipi di Smell differenti: **General Fixture**, ovvero quando l'inizializzazione delle variabili d'istanza nel metodo di `setUp` è eccessivamente generica, e la classe di test utilizza solo parte di esse; **Eager Test**, che avviene quando un metodo di test esercita più di un metodo di produzione, complicandone la comprensibilità e aumentando inutilmente la dipendenza tra metodi; **Lack of Cohesion of Methods**, un particolare tipo di Smell rilevato nelle classi che contengono una scarsa coesione tra i metodi di test, rendendone problematico il mantenimento.

Uno strumento molto simile al precedente è **DARTS** (**Detection And Refactoring of Test Smells**) [5]. Esso riconosce gli stessi tipi di Test Smell riconosciuti da TASTE, ma a differenza del precedente, è disponibile come plugin per l'IDE INTELLIJ IDEA. La rilevazione avviene sulla base dei commit effettuati, e sfrutta le API fornite da INTELLIJ per consentire un refactoring adeguato. Tale modulo è divisibile in due layer:

- **Layer di interfaccia**: responsabile dell'interazione con l'utente e della logica pertinente alla creazione dei contenuti visibili nell'interfaccia grafica dell'IDE, sfruttando quindi gli strumenti messi a disposizione da INTELLIJ per inserirlo all'interno della finestra principale del software;

- **Layer di applicazione**: contenente tutta la parte logica del plugin, responsabile quindi di tutti i meccanismi di detection e refactoring. Dal punto di vista pratico, ciò avviene sfruttando estensivamente la **PSI** (**Program Structure Interface**), ovvero una libreria contenente funzioni per la manipolazione del codice offerta da INTELLIJ.

Per lo sviluppo di TOTEM, è stato deciso di sfruttare quanto già fatto con DARTS e tool affini, per offrire un'esperienza più conforme a quella che è la user experience degli IDE più recenti. Anch'esso si tratta di un plugin per INTELLIJ, ma a differenza dei precedenti, è strutturato in modo da discernere la componente relativa all'interfaccia da quella responsabile della rilevazione di Test Smell. La prima, è stata sviluppata sfruttando le API di INTELLIJ IDEA, mentre la seconda è costituita da un modulo sviluppato in SPRING. TOTEM si occupa della rilevazione di tre tipi di Test Smell: *Eager Test*, *General Fixture* e *Lack of Cohesion*. Inoltre, TOTEM non sfrutta alcun modulo esterno: ogni notifica verrà visualizzata all'interno della finestra principale di INTELLIJ IDEA.

2.3 Architetture di riferimento per l'implementazione di Bot

Come menzionato nell'introduzione, tra i tool di assistenza agli sviluppatori che stanno riscuotendo particolare rilevanza negli ultimi anni rientrano i cosiddetti Bot [3]. Essi sono appunto strumenti caratterizzati da una componente di intelligenza artificiale, utile per consentire l'automazione di procedimenti che, alternativamente, risulterebbero eccessivamente ripetitivi e stancanti per gli sviluppatori.

Data la larga diffusione di queste tecnologie, sono state individuate delle architetture di riferimento per la loro implementazione [11]. In particolare, si possono notare due particolari architetture, ovvero **Dialogflow** e **Watson Assistant**, prese come standard per lo sviluppo di ChatBot.

Dialogflow è caratterizzata dall'unione di più componenti, normalmente autonome, all'interno di una *blackbox*, il cui contenuto è perfettamente ignorabile da parte degli sviluppatori. Oltre a questo, l'architettura in questione provvede un'interfaccia unificata per la comunicazione con lo sviluppatore, che sia in forma scritta o vocale, oltre che la possibilità di configurare un singolo endpoint in grado di gestire molteplici tipi di eventi. Come possiamo vedere, sfruttando questa architettura lo sviluppatore ignora completamente il funzionamento del Bot, limitandosi quindi a visualizzare le informazioni riportate dal ChatBot in questione.

L'architettura Watson Assistant, invece, nasconde anch'essa il suo funzionamento allo sviluppatore, offrendo però la possibilità di utilizzare un *Dialog Tree* per interagire con uno specifico nodo del sistema, piuttosto che comunicare con un unico endpoint. Tuttavia, essa prevede una comunicazione esclusivamente testuale (se non sfruttando componenti esterne), oltre che limitare notevolmente l'interazione tra moduli esterni e il *Dialog Tree* per l'attivazione di uno specifico nodo.

CAPITOLO 3

TOTEM - Module for the Detection of Test Smells

Nel prossimo capitolo, andremo innanzitutto a vedere quali sono i requisiti che ci siamo imposti riguardo lo sviluppo del nostro strumento. Andremo quindi a vedere come è proseguito lo sviluppo, partendo dalle tecnologie da noi utilizzate, per poi esaminare la struttura del sistema.

3.1 Obiettivo del tool

Al fine di sviluppare un tool che fosse capace di soddisfare gli obiettivi posti per il presente lavoro, si sono andati a individuare i seguenti requisiti funzionali:

RF 1.1 Rilevazione delle classi di produzione e di test

Il sistema deve presentare le funzionalità necessarie per la rilevazione automatica delle classi. TOTEM è uno strumento di analisi statica del codice, in grado di effettuarne automaticamente una revisione. Per fare ciò, è necessario che sia in grado di ricevere i contenuti oggetto dell'analisi. Per cui, è necessario sviluppare un sistema che, quando richiesto dall'utente, esegua in maniera autonoma una scansione delle classi presenti nei file del progetto, ed identificare quelle che sono le classi di test e le relative classi di produzione. Una volta individuate, il sistema dovrà effettuare autonomamente un'indagine sui contenuti delle classi in questione, per poi quantificare i dati importanti ai fini del processo di analisi;

RF 1.2 Analisi del codice

Il sistema deve offrire un meccanismo di analisi del codice. Una volta estrapolati i

dati in questione, è necessario che lo strumento interpreti quanto individuato. Sarà quindi necessario fare in modo che le informazioni ottenute vengano utilizzate per determinare se nel codice analizzato vi siano delle anomalie da riportare all'utente oppure no;

RF 1.3 Visualizzazione dei risultati
Il sistema dovrà prevedere un modo per visualizzare all'utente quali classi presentano le problematiche individuate.

Oltre a quanto già detto, sarebbe preferibile se il sistema presentasse anche le seguenti caratteristiche:

RNF 1 Un'interfaccia grafica curata e intuitiva, piuttosto che una semplice interfaccia a riga di comando, per rendere il tool più semplice da utilizzare;

RNF 2 Un sistema in grado di suggerire automaticamente possibili soluzioni alle problematiche individuate, oltre che un eventuale sistema di refactoring automatico;

RNF 3 Una sezione che comunica all'utente quali metodi causano un eventuale Smell e, se possibile, quali segmenti di codice sono interessati da esso;

RNF 4 Un'adeguata modularità del sistema, così da non renderlo eccessivamente dipendente dall'IDE.

Il codice sorgente di TOTEM, così come altre risorse utili, sono disponibili pubblicamente su GITHUB. [1]

3.2 Tecnologie utilizzate

Per lo sviluppo del plugin e dell'interfaccia grafica (vedi RNF 1), è stato sfruttato il workflow di INTELLIJ basato sul plugin GRADLE [12]. Il modulo in questione automatizza la gestione delle *dependencies* del progetto e, soprattutto, offre diverse funzionalità utili allo sviluppo di un plugin, in primis la possibilità di simulare un'istanza dell'IDE in cui viene eseguito il plugin sviluppato. Questa componente è stata sviluppata in JAVA sfruttando la **Program Structure Interface (PSI)** [13] di INTELLIJ, ossia il layer delle API di INTELLIJ IDEA responsabile della lettura dei file e dell'analisi della loro struttura.

Per il modulo responsabile della detection di Test Smell (vedi RNF 4), è stato impiegato SPRING BOOT [14], un'estensione del framework SPRING [15] pensata per la creazione di

[1] Link alla repository di TOTEM: `https://github.com/drybonez01/TOTEM`

Web Server sviluppati in JAVA. In questo modulo avverrà quindi la detection di Test Smell all'interno del codice, sfruttando parte del lavoro già fatto in DARTS [5] per la rilevazione dei tre tipi di Smell interessati.

Per la comunicazione tra i due moduli, i dati vengono scambiati sotto forma di file JSON, sfruttando la libreria di Google GSON [16] per la serializzazione e deserializzazione tra oggetto e JSON. In fase di sviluppo, è stato inoltre utilizzato il software POSTMAN [17] per testare le API del modulo di detection.

3.3 Architettura del sistema

Il sistema si basa su un'architettura **Client-Server**, dove il client è rappresentato dal plug-in di INTELLIJ IDEA sviluppato in JAVA, mentre invece il modulo per la detection è rappresentato da un server locale basato su SPRING.

Tra i due moduli, la comunicazione avviene per mezzo di file JSON, contenenti le informazioni necessarie per individuare eventuali Test Smell (nel caso dei dati destinati al modulo di detection), oppure le informazioni relative agli Smell individuati (nel caso dei dati destinati al plug-in). Questi dati sono rappresentati da dei JAVABEANS contenenti le informazioni necessarie ai rispettivi moduli di destinazione, mentre per il mapping tra oggetto e file JSON è stata adoperata la libreria GSON.

Per comunicare tra i moduli, quindi, sfruttiamo le classi `PsiClassBean` e, per le informazioni relative ai singoli metodi, `PsiMethodBean`. Quest'ultima contiene al suo interno i dati relativi ad un singolo metodo utili alla detection di Test Smell, ovvero quali variabili sono state inizializzate, quali sono state usate e quali altri metodi vengono chiamati, nonché il nome del metodo in questione. La prima, invece, sfrutta la stessa logica per comunicare informazioni relative alle singole classi: abbiamo quindi il nome della classe, una lista dei metodi al suo interno (sfruttando la classe `PsiMethodBean` descritta in precedenza), una lista delle variabili di istanza e, nel caso delle classi di test, un riferimento alla classe di produzione. È presente anche un riferimento al package di cui fa parte la classe, ma essendo inutile ai fini della detection, esso resta inutilizzato. La classe `PsiClassBean` è quindi utilizzata sia per comunicare al server quali sono le classi di test da sottoporre ad analisi, sia per comunicare al plugin quali sono le classi di test che contengono Test Smell.

Per comunicare al plugin quali metodi causano Eager Test e General Fixture, invece, utilizziamo le classi `MethodsWithEagerTest` e `MethodsWithGeneralFixture`.

Nella figura 3.1, viene illustrata l'architettura dell'intero sistema.

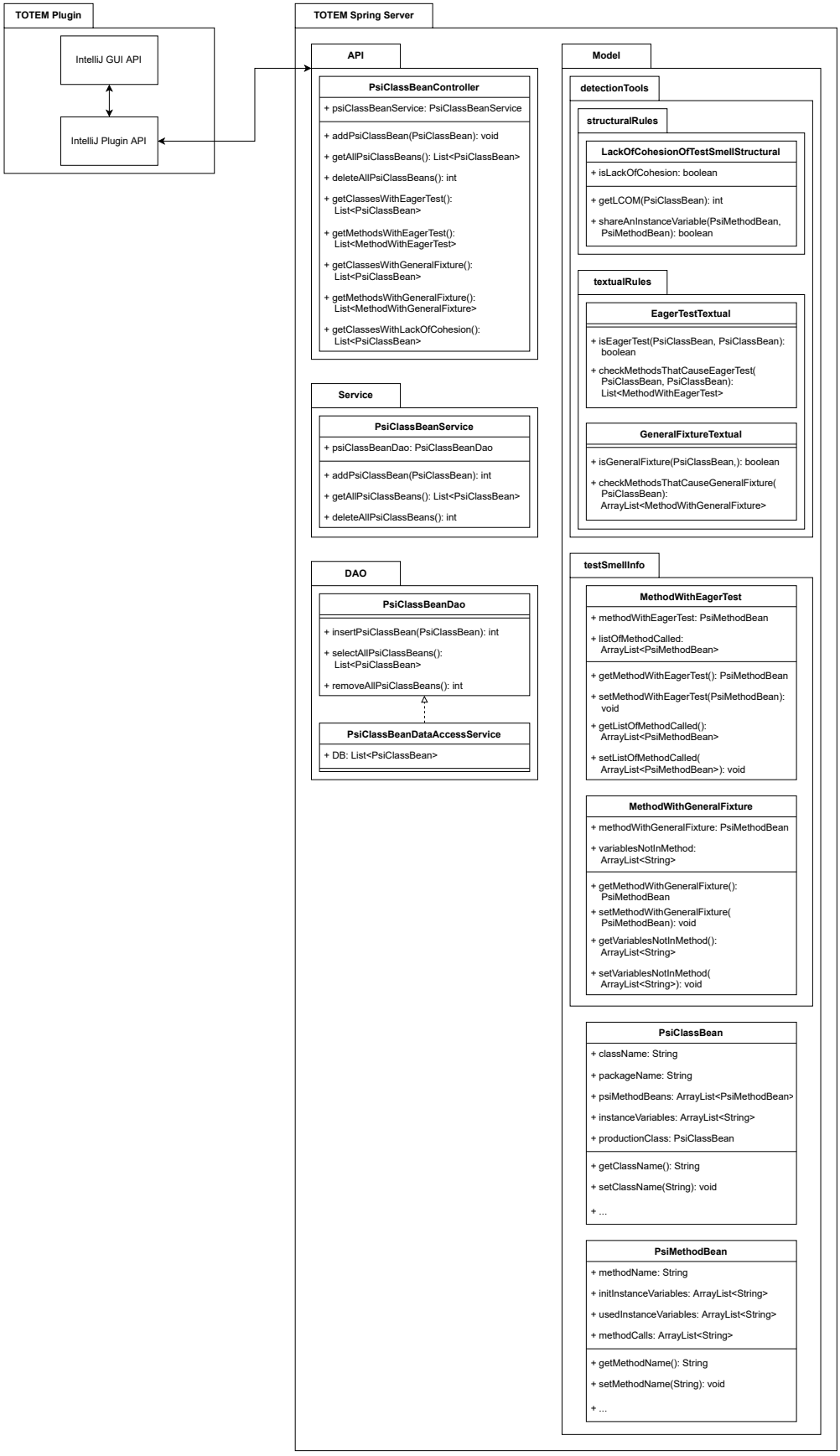

Figura 3.1: Un'illustrazione dell'architettura di TOTEM

3.3.1 Plug-in di IntelliJ

Il plug-in di INTELLIJ si presenta come un elemento selezionabile dalla barra degli strumenti dell'IDE. Praticamente, è necessario che l'utente prema un pulsante per far partire il processo di analisi del codice. Per la sua implementazione, è stato sfruttato il concetto di *Action*, una funzionalità delle API di INTELLIJ che permette di eseguire del codice quando l'utente interagisce con un qualche elemento dell'interfaccia.

Una volta inizializzato il processo, è necessario in primis che il plugin rilevi le classi di test. Per fare ciò, vengono innanzitutto cercate tutte le classi che compongono il progetto. Dopodiché, viene determinato quali di esse sono classi di test e, ad ognuna di esse, viene associata la relativa classe di produzione.

Le classi di test sono individuate verificando se contengono la parola *Test* nel nome, mentre le classi di produzione ad esse associate sono individuate verificando se il loro nome è lo stesso della classe di test meno la sottostringa *Test*.

A questo punto, estrae i dati importanti ai fini dell'analisi, ovvero le variabili di istanza della classe, i metodi contenuti in essa e, per ogni metodo, le variabili inizializzate, utilizzate e i metodi chiamati, incapsulando tali informazioni relative ai metodi della classe in una lista di oggetti `PsiMethodBean`, per poi inserire tale lista, assieme a tutte le altre informazioni relative alla classe esaminata, in un oggetto `PsiClassBean`. Viene quindi inviato quanto individuato al modulo responsabile della detection, sotto forma di JSON nel quale vengono mappati tutti i parametri che compongono l'oggetto contente i dati necessari per l'analisi.

Pur essendo possibile eseguire l'analisi di più classi per volta, il plug-in è stato sviluppato in modo tale da inviare una sola classe, ottenere le informazioni sui Test Smell presenti in essa, memorizzare queste informazioni in una struttura dati apposita, rimuovere la classe dal database, e ripetere lo stesso procedimento per tutte le classi di test individuate.

Al termine dell'analisi, viene generata una stringa contenente tutte le informazioni relative all'analisi effettuata, e viene visualizzata per mezzo di una notifica a schermo. Qualora dovessero essere presenti dei Test Smell, verrà indicato quali classi ne sono affette e, nel caso di Eager Test e General Fixture, da quali metodi sono stati causati. Nella figura 3.2, un esempio della finestra con i risultati dell'analisi effettuata.

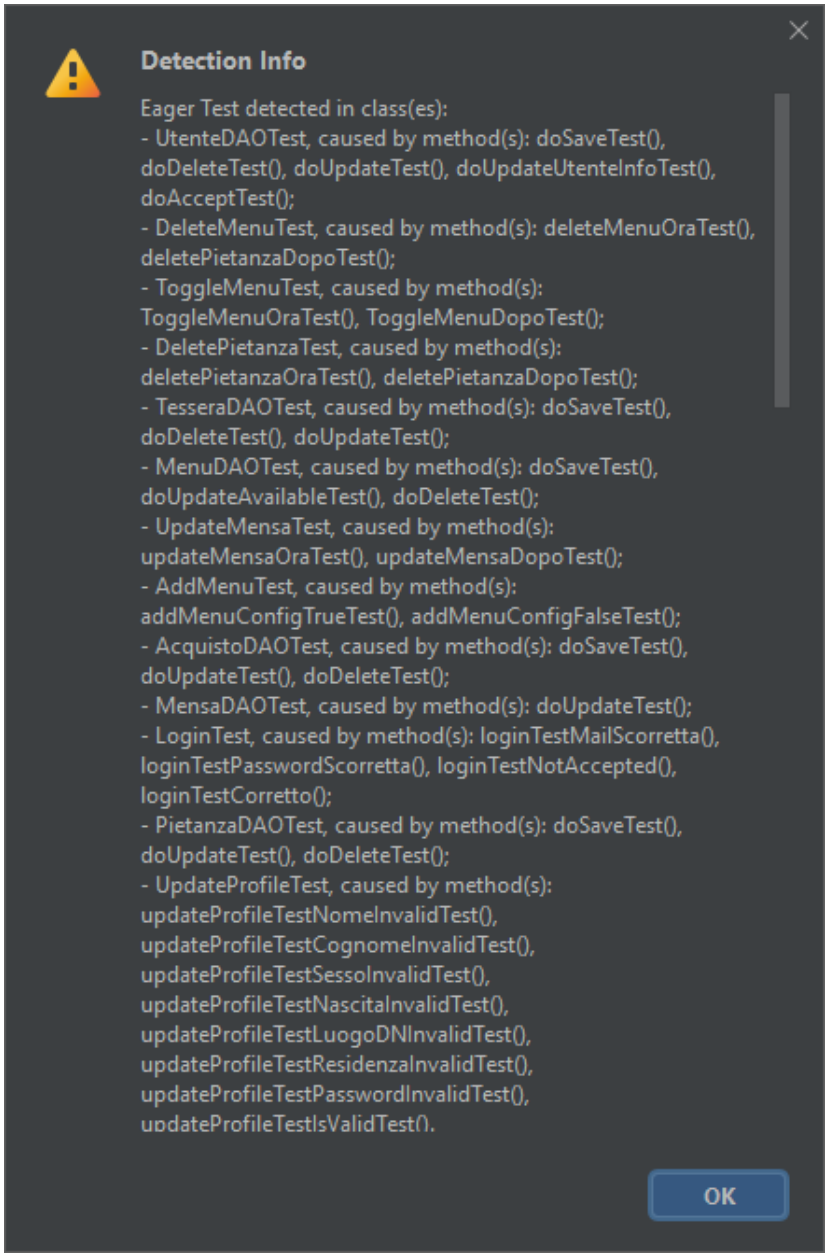

Figura 3.2: La finestra di dialogo di IntelliJ che mostra i risultati dell'analisi

3.3.2 Modulo per la detection di Test Smell

Analizziamo dunque il funzionamento del cuore di TOTEM. All'interno di questo modulo, avvengono tutte le operazioni di analisi della struttura del codice. Questo modulo non è altro che un server in esecuzione sullo stesso sistema da cui opera l'utente, al quale il plug-in si interfaccia per mezzo di richieste HTTP, ricevendo e inviando informazioni sotto forma di file JSON. Il modulo offre quindi delle API per l'invio delle informazioni necessarie all'analisi, nonché per ricevere dati indicanti se e quali classi contengono i tre Test Smell che è in grado di riconoscere: Eager Test, General Fixture e Lack of Cohesion.

Eager Test

Un Eager Test [4] è uno Smell rilevato in tutte quelle classi che contengono metodi che esercitano più di un metodo della classe di produzione. Per cui, l'algoritmo utilizzato esegue un'analisi del codice per verificare se uno dei metodi di test contiene più di una chiamata a un metodo di produzione. In tal caso, tale classe viene etichettata come "sporca" e aggiunta alla lista delle classi affette da Eager Test, oltre che identificare tutti i metodi alla base di questo problema.

General Fixture

Si dice che una classe è affetta da General Fixture [4] quando l'inizializzazione delle variabili d'istanza è eccessivamente generica, e quindi alcune di esse restano inutilizzate. Per ogni metodo di una classe di test, quindi, viene verificato se utilizza tutte le variabili di istanza, iterando tra tutte le variabili e verificando se ognuna di esse è utilizzata in tutti i metodi. In caso contrario, quindi se tale variabile non compare in un metodo, tale metodo causa General Fixture all'interno della classe.

Lack of Cohesion

Lack of Cohesion [5] è un particolare tipo di Test Smell, caratterizzato da un'eccessiva discrepanza tra le variabili dei vari metodi di test. La tolleranza di questa differenza è totalmente a discrezione dello strumento utilizzato, ma nel nostro caso, vengono analizzate le variabili utilizzate dai metodi di test e, qualora due o più metodi dovessero utilizzare un sottoinsieme di variabili differente, allora la classe analizzata è affetta da Lack of Cohesion.

3.4 Caso di studio

Al termine dello sviluppo di TOTEM, abbiamo eseguito il tool su una serie di progetti sviluppati in JAVA. Per osservare al meglio il comportamento del tool, è stato eseguito su progetti di dimensione sempre crescente.

Pur cercando di mantenere una certa eterogeneità nei progetti esaminati, ci siamo concentrati perlopiù su tool di sviluppo e su macro-raccolte di algoritmi. Abbiamo quindi selezionato alcune delle repository più popolari su GITHUB, e abbiamo osservato il funzionamento di TOTEM con ognuno dei progetti presi in esame, ovvero:

- **authlib-injector** [18]: un plug-in per il videogioco *Minecraft*, pensato per introdurre un sistema di autenticazione per i vari giocatori;
- **java-concurrency-samples** [19]: un piccolo progetto pensato per mostrare il funzionamento della concorrenza in JAVA;
- **logstash-gelf** [20]: estensione che provvede la possibilità di utilizzare LOGSTASH con diversi framework JAVA normalmente incompatibili;
- **aviatorscript** [21]: un linguaggio di scripting basato su JAVA, eseguibile sulla JAVA VIRTUAL MACHINE;
- **Recaf** [22]: tool che permette di modificare il bytecode del codice JAVA;
- **Java-WebSocket** [23]: implementazione basilare di un WebSocket, contenente sia un client che un server di esempio;
- **Algorithms** [24]: una collezione di una vasta serie di algoritmi e strutture dati comunemente utilizzati, scritti interamente in JAVA;
- **checkstyle** [25]: un tool per verificare l'aderenza del codice ad una o più convenzioni specifiche.

Nella tabella 3.1, illustriamo quanto osservato dalle varie esecuzioni di TOTEM su questi progetti.

Nome del progetto	Classi di test	Eager Test	General Fixture	Lack of Cohesion	Totale	Smell per classe
authlib-injector	4	0	0	3	3	0,75
java-concurrency-samples	6	6	0	1	7	1,17
logstash-gelf	6	0	0	3	3	0,5
aviatorscript	7	2	3	3	8	1,14
Recaf	10	0	4	1	5	0,5
Java-WebSocket	29	17	0	19	36	1,24
Algorithms	69	56	13	57	126	1,83
checkstyle	317	3	2	271	276	0,87

Tabella 3.1: Report dell'esecuzione di TOTEM su alcuni progetti open-source

Innanzitutto, possiamo notare come dalla nostra analisi possa risultare evidente che il numero di smell per ogni progetto NON sia direttamente proporzionale al numero di classi

che costituiscono tale progetto. Questo significa che, ad esempio, un progetto con un certo numero di classi di test può tranquillamente presentare meno Test Smell rispetto ad uno con un numero decisamente inferiore di classi di test. Ciò evidenzia come la buona scrittura di una classe di test sia innanzitutto una prerogativa del programmatore, e non si tratti necessariamente di un requisito irrealizzabile con l'aumentare delle dimensioni del progetto.

Oltre a questo, è da notare come, nella maggior parte dei progetti, il Test Smell più frequente è senza dubbio Lack of Cohesion. A onor del vero, è necessario precisare come lo strumento da noi sviluppato sfrutti una metrica particolarmente "severa" per la sua rilevazione, per cui sarebbe da considerare di favorire maggiormente l'utente in una eventuale futura versione del software. Tuttavia, è palese come ci sia una notevole discrepanza nella rilevazione tra questo tipo di Test Smell e tutti gli altri, sinonimo di come vi sia una scarsa cultura a riguardo, quando invece si tende a dare la priorità agli altri due Smell.

Da un punto di vista più tecnico, invece, lo strumento si è rivelato particolarmente affidabile. L'unico limite riguarda la strategia utilizzata per rilevare le classi di test, nonché le classi di produzione ad esse associate. TOTEM sfrutta un principio abbastanza semplice: se una classe contiene la parola *Test* nel nome, allora è una classe di test, e la classe di produzione ad essa associata condividerà lo stesso nome meno la sottostringa *Test*. È tuttavia palese come questo strumento non funzioni adeguatamente con progetti che non rispettano tale convenzione. Per limitazioni di tempo, non ci è stato possibile sviluppare un meccanismo alternativo, ma questa possibilità è da considerare per una futura versione del software.

3.5 Guida all'installazione

Per installare TOTEM, è prima di tutto necessario scaricare il codice sorgente dalla repository indicata precedentemente. Il progetto è composto da due moduli: il modulo principale, ovvero il plug-in di INTELLIJ, ed un sotto-modulo, contenuto nella directory `TOTEM_Spring`, ovvero il server responsabile della rilevazione di Test Smell.

Sarà innanzitutto necessario avviare l'esecuzione del server in background. È possibile eseguire il server direttamente dall'IDE, oppure esportare il progetto in un file `jar` ed avviarne l'esecuzione esternamente.

Una volta terminato questo passaggio, sarà possibile utilizzare il plug-in vero e proprio. È possibile sia simulare un'istanza dell'IDE su cui è stato installato il plug-in, sia installare direttamente il plug-in sul proprio IDE, esportandolo sempre in un file `jar` ed eseguendo un'installazione da file locale. L'importante è che il server resti in esecuzione.

CAPITOLO 4

Conclusioni

In quest'ultimo capitolo, andremo a riflettere su quanto fatto finora, sulle difficoltà affrontate durante lo sviluppo e, soprattutto, sui possibili sviluppi futuri del nostro software. Dovremo quindi inquadrare quelle che sono le criticità del modulo sviluppato, per poi riflettere su come sarebbe possibile risolverle in una eventuale futura versione.

Ora che lo sviluppo del software è giunto a conclusione, è lecito chiedersi se e quanto il risultato finale sia conforme a quanto previsto inizialmente.

Innanzitutto, TOTEM può definirsi pienamente coerente con quanto dichiarato nei requisiti funzionali. Il sistema prevede quindi le funzionalità necessarie per la rilevazione automatica delle classi di test, per l'analisi del codice e per la comunicazione dei risultati.

In quanto ai requisiti non funzionali, tuttavia, non siamo stati in grado di garantire un sistema perfettamente conforme a quanto dichiarato inizialmente. In particolare, TOTEM soddisfa il criterio di modularità previsto, dividendo correttamente il progetto in più moduli in base al compito richiesto. Inoltre, siamo riusciti anche ad implementare correttamente i sistemi di comunicazione con l'utente, presentando l'output dell'analisi svolta per mezzo della UI e comunicando precisamente non solo quali classi, ma anche quali metodi presentano Test Smell. Purtroppo però, l'interfaccia non rispetta pienamente gli standard di qualità che ci eravamo imposti inizialmente. Attualmente TOTEM si limita a mostrare una notifica a schermo contenente tutte le informazioni relative ai Test Smell individuati, quando sarebbe stato preferibile avere un'interfaccia più curata e con qualche ulteriore funzionalità rispetto a quelle proposte. Anche le informazioni relative ai Test Smell sarebbero potute essere

più corpose: pur essendo riusciti ad implementare una detection a livello di metodo, non siamo riusciti a fornire informazioni più precise riguardo lo Smell individuato, ad esempio indicando le variabili che causano General Fixture. Infine, è stata accantonata l'idea di offrire un meccanismo di refactoring automatico per gli Smell poiché, oltre ad essere un processo eccessivamente dispendioso, non siamo riusciti a individuare un'euristica adeguata a tale fine.

4.1 Sviluppi futuri

Di seguito si andranno ad elencare i possibili sviluppi futuri individuati nel contesto del presente lavoro di tesi.

Prima di tutto, sarà necessario partire dai requisiti non funzionali che non sono stati implementati per offrire un software più intuitivo, nonché conforme a quelle che possono essere le aspettative dell'utenza.

Oltre a questo, sono state individuate altre lacune all'interno di TOTEM che sarebbe opportuno risolvere in future versioni del software. Innanzitutto, abbiamo già menzionato il metodo utilizzato per individuare le classi di test, basato sulla nomenclatura delle classi. Questa euristica, tuttavia, si rivela inutile con i progetti che non rispettano tale convenzione, per cui sarebbe opportuno proporre metodi alternativi per l'individuazione delle classi di test.

Inoltre, durante l'esecuzione del tool sono stati individuati degli evidenti limiti prestazionali. Tale strumento è infatti estremamente pesante, richiedendo anche diversi minuti per eseguire l'analisi di un progetto, specialmente su pc meno prestanti. Questo è attribuile anche a una scarsa ottimizzazione delle risorse sfruttate, la quale, per limitazioni di tempo, non è stata presa particolarmente in considerazione. Sarebbe quindi opportuno prestare maggiore attenzione alle risorse impiegate e sfruttare algoritmi più efficienti, qualora il software dovesse venire aggiornato in futuro.

Infine, per quanto si tratti di un lavoro tutt'altro che semplice, sarebbe opportuno implementare in futuro le funzionalità necessarie per individuare altri Test Smell. Difatti, il tool si è dimostrato perfettamente funzionante, e sarebbe opportuno sfruttare quanto già fatto come base per sviluppare un nuovo strumento, ma che possa offrire nuove funzionalità oltre a quelle già presenti, a partire appunto dalla possibilità di individuare altri Test Smell oltre a quelli già rilevabili.

Bibliografia

[1] L. Luo, "Software testing techniques," *Institute for software research international Carnegie mellon university Pittsburgh, PA*, vol. 15232, no. 1-19, p. 19, 2001. (Citato a pagina 1)

[2] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pp. 4–15, 2016. (Citato alle pagine 2, 6 e 13)

[3] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 928–931, 2016. (Citato alle pagine 2 e 15)

[4] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on software engineering*, vol. 33, no. 12, pp. 800–817, 2007. (Citato alle pagine 3 e 22)

[5] S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia, and F. Palomba, "Just-in-time test smell detection and refactoring: The darts project," in *Proceedings of the 28th International Conference on Program Comprehension*, pp. 441–445, 2020. (Citato alle pagine 3, 14, 18 e 22)

[6] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 403–414, IEEE, 2015. (Citato a pagina 5)

[7] "Software Unit Test Smells." `https://testsmells.org/index.html`. (Citato a pagina 6)

[8] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1650–1654, 2020. (Citato a pagina 13)

[9] F. Pecorelli, G. Di Lillo, F. Palomba, and A. De Lucia, "Vitrum: A plug-in for the visualization of test-related metrics," in *Proceedings of the International Conference on Advanced Visual Interfaces*, pp. 1–3, 2020. (Citato a pagina 13)

[10] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 311–322, IEEE, 2018. (Citato a pagina 14)

[11] S. Srivastava and T. Prabhakar, "A reference architecture for applications with conversational components," in *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 1–5, IEEE, 2019. (Citato a pagina 15)

[12] "Building Plugins with Gradle - IntelliJ Platform Plugin SDK." `https://plugins.jetbrains.com/docs/intellij/gradle-build-system.html`. (Citato a pagina 17)

[13] "Program Structure Interface (PSI) - IntelliJ Platform Plugin SDK." `https://plugins.jetbrains.com/docs/intellij/psi.html`. (Citato a pagina 17)

[14] "Spring Boot." `https://spring.io/projects/spring-boot`. (Citato a pagina 17)

[15] "Spring." `https://spring.io/`. (Citato a pagina 17)

[16] "Gson." `https://github.com/google/gson`. (Citato a pagina 18)

[17] "Postman API Platform." `https://www.postman.com/`. (Citato a pagina 18)

[18] "yushijinhun / authlib-injector." `https://github.com/yushijinhun/authlib-injector`. (Citato a pagina 23)

[19] "aheusingfeld / java-concurrency-samples." `https://github.com/aheusingfeld/java-concurrency-samples`. (Citato a pagina 23)

[20] "mp911de / logstash-gelf." `https://github.com/mp911de/logstash-gelf`. (Citato a pagina 23)

[21] "killme2008 / aviatorscript." `https://github.com/killme2008/aviatorscript`. (Citato a pagina 23)

[22] "Col-e / recaf." `https://github.com/Col-E/Recaf`. (Citato a pagina 23)

[23] "Tootallnate / java-websocket." `https://github.com/TooTallNate/Java-WebSocket`. (Citato a pagina 23)

[24] "williamfiset / algorithms." `https://github.com/williamfiset/Algorithms`. (Citato a pagina 23)

[25] "checkstyle / checkstyle." `https://github.com/checkstyle/checkstyle`. (Citato a pagina 23)

Ringraziamenti

In questo momento si conclude un percorso estremamente importante per me. Un percorso estremamente turbolento, non andrò a mentire, un po' per sua natura, un po' per le difficoltà di chi sta scrivendo queste righe, un po' per... eventi imprevisti e che hanno stravolto la mia routine e il mio metodo di studio più e più volte nel corso di questi anni, se così possiamo dire. Suvvia, penso si sia capito di cosa sto parlando.

Però sì, sono arrivato a questo punto. Ho raggiunto un traguardo che fino a qualche anno fa mi sembrava così lontano, una vetta assolutamente irraggiungibile con le mie sole forze, eppure... ce l'ho fatta. Non sono interessato a fare i soliti discorsi smielati. In fondo, sapevo che se mi fossi impegnato sarei riuscito a fare qualsiasi cosa. E quindi l'ho fatto. E ci sono riuscito!

Ma in ogni caso, se sono arrivato a questo punto è stato anche grazie a tutte le persone che mi sono state vicine in questi anni. Vorrei ringraziare innanzitutto il Prof. Fabio Palomba, per aver accettato di supportare il mio progetto, e i Dott. Stefano Lambiase e Valeria Pontillo, che nel corso di questi mesi mi hanno seguito e aiutato con costanza in questo compito tutt'altro che semplice. Infine, vorrei ringraziare tutti i miei amici e compagni di corso più stretti, dal primo all'ultimo, per avermi sempre motivato a portare avanti il mio lavoro, e per essersi sempre schierati dalla mia parte, soprattutto nei momenti meno felici. E ovviamente, ringrazio anche i miei genitori, per il supporto costante che mi hanno garantito in questi tre anni.

Cosa mi aspetta adesso? Sinceramente, non lo so. Ho bisogno di un po' di tempo per decidere quel che ne sarà del mio futuro, ma intanto, lasciatemi godere il presente.