UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Estensione del plug-in aDoctor per l'identificazione e il refactoring automatico di code smell energetici in applicazioni Android

RELATORE

Prof. **Andrea De Lucia**

Dott. **Fabiano Pecorelli**

Dott. **Emanuele Iannone**

CANDIDATO

Antonio De Matteo

Matricola: 0512105103

Anno Accademico 2019-2020

Alla mia Famiglia, fonte continua d'ispirazione.

ABSTRACT

Le applicazioni mobili ("app in seguito") sono sempre più popolari ed utilizzate negli anni recenti da parte di persone di tutte le età. La progettazione e la diffusione di un app di successo richiede di tenere conto di diversi vincoli contrastanti tra di loro: User Experience, prestazioni, privacy e consumo energetico. Visto il forte aumento delle prestazioni e della qualità in generale dei dispositivi, gli utenti di app si aspettano un'esecuzione sempre più ottimale. Un modo per poter ridurre il consumo energetico, senza costringere l'utente a ricorrere a modalità di risparmio energetico, sarebbe quello di migliorare la qualità del codice stesso delle app, idealmente tramite un meccanismo di ottimizzazione deterministico e sempre applicabile. Nel contesto del sistema operativo Android, sono state individuate un insieme di cattive pratiche di progettazione e implementazione che si manifestano nel codice delle app con dei sintomi chiamati "code smell" individuabili attraverso tecniche di analisi del codice. Tra questi, ce ne sono sei che riguardano il consumo energetico. Per questi energy smell esistono delle operazioni di refactoring, ovvero soluzioni standard applicabili per poter rimuovere gli smell, lasciando inalterato il funzionamento globale dell'applicazione.

Tuttavia, gli attuali strumenti utilizzabili per app forniscono supporto limitato e , cosa più importante, non sono disponibili per gli sviluppatori interessati a monitorare la qualità delle proprie app. Proprio per risolvere questi problemi, in lavori precedenti, è stato definito il tool A-DOCTOR (AnDrOid Code smell detecTiOn and Refactoring), un plugin per Android Studio che consente di individuare code smell in applicazioni Android "(*a.k.a.,* app)" , mediante tecniche di analisi del codice. Una volta indiviuati, si attuano delle operazioni automatiche di refactoring.

Nell'ambito di questa tesi, è stato esteso A-DOCTOR in modo da consentire l'identificazione e la correzione automatica di una quantitá maggiore di code smells energetici.

Indice

CAPITOLO 1

Introduzione

1.1 Motivazioni e Obiettivi

I sistemi software, in tutto il ciclo di vita, sono soggetti a molti cambiamenti. Tali cambiamenti possono essere dovuti a correzioni di possibili bug presenti all'interno del codice, o per richieste effettuate da parte dei clienti o per l'implementazione di nuove funzionalità. Essi diventano in maggior numero quando si attraversa la fase di implementazione, durante la quale gli sviluppatori possono compiere degli errori di scelte che andranno a impattare negativamente sulla qualità del codice scritto. Anche se queste modifiche risultano funzionanti, esse danneggiano la qualità del software sotto diversi punti di vista, come la manutenibilità, l'usabilità, le performance, il consumo energetico ecc. Con il passare del tempo sono state individuate delle cattive pratiche di programmazione, che hanno assunto il nome di " **bad code smell**", (*a.k.a.,* code smell o smell) Uno smell è un pezzo di codice sorgente che può essere la causa di un problema non di funzionamento ma bensì di qualità, molte volte, gli stessi sviluppatori, creatori di un software, non si accorgono di avere davanti uno smell, peggiorando la qualità del proprio codice. Per risolvere questi problemi, si effettuano delle operazioni di refactoring, ovvero "una modifica apportata alla struttura interna del software per renderlo più facile da capire e più economico da modificare senza cambiare il suo comportamento osservabile". Negli ultimi anni i code smell sono stati analizzati sotto diverse prospettive [1][2]. La loro introduzione [3][4] , evoluzione [5][6][7][8][9], il loro impatto sull'affidabilità [10][11] e sulla manutenibilità [12][13], sono stati analizzati a fondo ed è stato

percepito [14][15][16] che possono essere una seria minaccia per la qualità del codice sorgente. Gli smell non hanno sempre una definizione rigorosa,anzi,spesso la loro definizione è ad alto livello e in linguaggio naturale. Non esistono criteri formali per stabilire la presenza o meno di uno smell, tuttavia vengono in aiuto diverse euristiche che stabiliscono la presenza di uno smell osservando alcuni suoi noti modi di manifestarsi. Similmente, non tutte le operazioni di refactoring possono essere definite in modo preciso (*i.e,* attraverso un insieme finito di azioni di modifica del codice), e spesso la loro buona riuscita è dettata dall'esperienza degli sviluppatori, nemmeno le operazioni di refactoring possono essere definite in modo formale, ma anche esse sono guidate dell'esperienza degli sviluppatori. Le operazioni di refactoring possono essere attuate manualmente o automaticamente:

- Con operazioni manuali, è uno sviluppatore ad effettuare l'analisi, osservando il codice e il suo comportamento in fase di esecuzione, per poi attuare i refactoring dove occorrono;
- Con operazioni automatiche, l'analisi e il refactoring sono realizzati per mezzo di appositi tool. Negli ultimi anni, la comunità scientifica ha prestato molta attenzione allo sviluppo di questi tool, realizzando sia software di identificazione degli smell che software di refactoring automatico.

I code smell sono principalmente problemi di manutenibilità, mentre esistono altri tipi di code smell che impattano su altri aspetti qualitativi, come l'usabilità e le performance. In particolare quelli legati al consumo energetico sono meglio conosciuti come energy smell. Inoltre, alcuni sono specifici di una particolare framework o piattaforma, ad esempio quelli di Android. Nel lavoro di questa tesi si è rivolta l'attenzione sugli energy smell nel contesto di applicazioni Android, ovvero degli smell che possono portare ad un dispendio energetico nei dispositivi in cui il software viene installato. Recenti studi hanno mostrato che scelte sbagliate nell'implementazione di codice sorgente porta ad una diminuzione di efficienza energetica; per esempio, Hasan et al. , hanno dimostrato che una struttura dati sbagliata può portare al 300% di spreco energetico [17]; Palomba et al. [18][19] hanno analizzato l'impatto di alcuni code smell per Android ed ha scoperto che la loro presenza ha un notevole impatto sulla batteria del device. Nel catalogo introdotto da Reimann vengono correlati i concetti di *bad smell, quality* e *refactoring* e viene coniato il termine *quality smell*. In questo catalogo, Reimann inserisce 30 quality smell. A seguito di ciò viene introdotto, una seconda versione di **A-DOCTOR** (AnDrOid Code smell detecTiOn and Refactoring), nel seguito denominato aDoctor[22], tool di identificazione di smell per app Android scritte in linguaggio (*i.e.*, scritte

in JAVA). L'obiettivo di questo studio è quello di espandere il tool preesistente, rendendo possibile la detection e il refactoring di un maggior numero di smell fornendo uno strumento che rappresenti un ulteriore passo in avanti nello studio degli smell e del refactoring.

1.2 Struttura della tesi

La tesi si compone di cinque capitoli, incluso questa introduzione, seguendo questo schema:

- ***Capitolo 2: Stato dell'arte su code smell: detection and refactoring.***
  In questo capitolo viene presentato lo stato dell'arte sulla detection e il refactoring automatici di code smell, descrivendo le principali tecniche e tool.

- ***Capitolo 3: Smell presenti in aDoctor***
  In questo capitolo si descrive nel dettaglio la versione corrente di aDoctor, ovvero quanti e quali energy smell è in grado di identificare e rimuovere.

- ***Capitolo 4: Estensione di aDoctor con analisi di nuovi smell***
  In questo capitolo viene presentata la nuova versione diaDoctor, mostrando le nuove funzionalità aggiunte e le modifiche alle componenti pre-esistenti.

- ***Capitolo 5: Conclusioni e sviluppi futuri***
  In questo capitolo vengono presentate considerazioni finali sul lavoro effettuato e dei possibili sviluppi futuri.

CAPITOLO 2

Stato dell'arte

2.1 Code Smell Detection

Uno dei principali problemi dei code smell è la loro individuazione, infatti, a causa dell'assenza di criteri formali per la corretta identificazione di code smell non è semplice realizzare dei tool che ne permettano l'identificazione. Negli ultimi decenni, diversi ricercatori hanno studiato la natura e l'impatto dei code smell. Nell'identificazione di code smell sono stati effettuati due tipi di approcci: approcci di tipo euristico, basati sul calcolo di metriche (strutturali, testuali o storiche) e approcci basati su tecniche di machine learning.

2.1.1 Identificazione di code smell tramite tecniche di machine learning

Le tecniche di questa categoria sfruttano un metodo supervisionato: più nello specifico, un insieme di variabili indipendenti (*a.k.a.*, predictors) usate per predire il valore della variabile dipendente usando un classificatore di machine learning (come ad esempio Logistic Regression [28]). Il modello deve essere addestrato usando una quantità di dati sufficientemente elevata provenienti da un singolo progetto o un insieme di altri simili. Kreimer [29] ha proposto un modello predittivo che, sulla base delle metriche del codice, può portare a livelli elevati di accuratezza. Il modello adotta il decision tree per rilevare due tipi di code smell (Blob e Long Method). Più tardi, Amorim et al. [30] hanno confermato i valori precedenti valutando le performance del decision tree su quattro progetti opensource di media scala. Vaucher et al. [31] hanno studiato l'evoluzione di Blob basandosi su un classificatore Naive

Bayes, mentre Maiga et al. [32] [33] hanno proposto l'uso di Support Vector Machine (SVM) e hanno dimostrato che un tale modello può raggiungere una F-measure di circa 80%. L'uso di Bayesian Belief networks per rilevare istanze Blob, Functional Decomposition e Spaghetti Code su programmi open-source proposti da Khom et al. [34][35] hanno portato ad una F-measure complessiva vicina al 60%. Allo stesso modo Hassaine et al. [36] hanno definito un approccio immune-inspired per la rilevazione di Blob smell, mentre Oliveto et al. [37] hanno usato un B-Splines per rilevarli. Più recentemente alcuni autori hanno investigato sulla flessibilità del machine-learning per rilevare code clones [38] [39]. Arcelli Fontana et al. hanno portato i progressi più rilevanti in questo campo [40], [41], [42]. Nel loro lavoro:

- Hanno sostenuto che la ML potrebbe portare ad una valutazione più oggettiva della pericolosità dei code smell [41];
- Fornito un metodo ML per valutare l'intensità di un code smell [42];
- comparando 16 tecniche ML per la rilevazione di quattro tipi di code smell [40] hanno mostrato che ML può portare il valore della F-measure vicino al 100%.

Tuttavia questo risultato e stato smentino da Di Nucci et al.[43] prima e da Pecorelli et al. [27] poi, mostrando che le performance degli approcci basati su Machine Learning peggiorano vertiginosamente se testate su dataset reali.

2.1.2 Identificazione euristica e strutturale

Gli approcci euristici identificano code smell per mezzo di regole di rilevamento basate su metriche software. Il processo generale seguito da questi approcci è composto da due fasi:

- l'identificazione di sintomi chiave che caratterizzano un code smell possono essere mappate su un insieme di soglie basate su strutture metriche (esempio, se le linee di codice sono oltre un certo numero k).
- la combinazioni di questi sintomi, che porta alla regola finale per la rilevazione dello smell [22], [23, 24, 25] [26].

Le tecniche di rilevamento adottate in questa categoria si differenziano maggiormente per l'insieme di metriche strutturali sfruttate, che dipendono dal tipo di code smell da identificare e da come i sintomi chiave sono combinati. Le tecniche strutturali, analizzano il codice sorgente e ne derivano strutture sintattiche (AST tipicamente) su cui fare analisi. La maggior parte degli approcci proposti hanno ottenuto una combinazione impiegando

gli operatori AND/OR [23, 25], [26] ed altri ancora hanno adottato metodi di clustering [24]. In questo contesto Moha et al. [22] hanno introdotto DECOR, un metodo per definire e rilevare code e design smell usando uno specifico linguaggio del dominio (DSL). Seguendo il processo generale descritto sopra, DECOR, usa un insieme di regole chiamate "rule card", che descrivono le caratteristiche intrinseche di una classe affetta da smell. Tsantalis et al. [24] hanno presentato JDEODORANT, un tool la cui prima versione è stata in grado di individuare Feature Envy smell, code smell che descrive quando un oggetto accede ai campi di un altro oggetto per eseguire qualche operazione, invece di dire semplicemente all'oggetto cosa fare, suggerendo il refactoring da effettuare. Successivamente altri code smell sono stati supportati (State checking, Long Method, Blob). La strategia di rilevazione per questi smell è basata su metriche di codice che sono connesse l'une con le altre utilizzando algoritmi di clustering e soglie per tagliare i dendogrammi (diagramma che mostra la relazione gerarchica tra gli oggetti) risultanti. Pecorelli et al. [27] hanno approfondito il confronto empirico tra le prestazioni delle tecniche basate sull'euristica e quelle basate sull'apprendimento automatico. I risultati affermano che c'è bisogno di ulteriori ricerche per migliorare l'efficacia dell'apprendimento automatico e degli approcci euristici per il rilevamento di code smell.

2.2 Identificazione e Refactoring di Energy Smell

Come introdotto precedentemente i code smell incidono sulla qualità del codice scritto anzichè sul funzionamento vero e proprio. Infatti, tramite l'utilizzo del tool PETrA [18][44] è stato dimostrato che alcuni code smell hanno un impatto sul consumo energetico del dispositivo, tali da essere considerati energy smell. L'individuazione degli smell, per via dell'assenza di criteri formali, non è facilmente realizzabile. Tuttavia, nel tempo, è nata l'esigenza di fornire una classificazione precisa e dettagliata degli smell più frequenti. Sono allora nati diversi cataloghi di smell, ovvero elenchi in cui vengono spiegati nel dettaglio un insieme di smell, mostrando i casi in cui essi si manifestano, quali aspetti di qualità vanno ad intaccare e come vanno risolti. Esistono diversi cataloghi noti pubblicamente, ma il più rilevante è sicuramente quello di Martin Fowler [20], dove vengono descritti ventidue smell relativi all'object-oriented programming ("OOP" nel seguito), e che rappresentano violazioni ai principi della stessa. Questi cataloghi spiegano come gli smell possano esistere a livello dell'intera applicazione (cioè dell'intero codice sorgente), a livello di package, a livello di classe oppure a livello di un singolo metodo. Grazie al loro aiuto, i lavori di identificazione e risoluzione vengono velocizzati, e sono quindi realizzabili sin dalle prime fasi dell'implementazione del

codice. E' bene infatti che gli smell vengano subito rimossi non appena individuati, poichè un grosso accumulo di essi rende difficile attuare le operazioni di refactoring. I cataloghi pubblici, tuttavia, possono essere lunghi e complessi da consultare, e quindi identificare un eventuale problema nel codice rischia di diventare tedioso e di scoraggiare gli sviluppatori nel farlo. Sono stati, allora, formulati dei cataloghi ridotti, più specifici per certe piattaforme, come ad esempio quelle desktop, quelle embedded e quelle mobile. Di nostro interesse è il contesto mobile con sistema operativo Android, basandoci sugli smell presenti nel catalogo di Reimann [21] (guardare tabella, nel quale ne vengono descritti trenta. Come già anticipato, l'identificazione degli smell eseguita in modo manuale non sempre risulta efficace, poichè gli sviluppatori potrebbero non essere capaci di individuare gli smell presenti nel loro codice, nonostante il supporto dei cataloghi pubblici. Nasce quindi l'esigenza di avere strumenti automatici per rilevare gli smell, ed eventualmente anche proporre una soluzione, da applicare con operazioni di refactoring automatico. Per andare in contro a questa necessità, Hecht et al. hanno ideato Paprika [45], un software di riconoscimento di smell in applicazioni Java. Tale tool riconosce tredici smell legati alla piattaforma Android ed altri quattro relativi alla OOP in generale, per un totale di diciassette. Sulla base di esso è stato poi costruito il tool Hot Pepper [46] che, assieme ad un sistema hardware di rilevazione del consumo energetico di un device Android, denominato Naga Viper, poteva tracciare una stima sull'impatto energetico degli smell rilevati da Paprika. Più di recente, Palomba et al. hanno realizzato aDoctor [43], in grado di riconoscere quindici smell relativi alla piattaforma Android. Paprika e aDoctor insieme sono capaci di rilevare un buon numero di smell relativi ad Android, tuttavia essi non sono in grado di realizzare operazioni di refactoring automatico per risolverli, e tanto meno proporre allo sviluppatore una possibile soluzione.

Tra i tool in grado di realizzare anche refactoring si cita, oltre al tool JDeodorant sopracitato, EARMO , in grado di realizzare refactoring di otto tipi di smell presenti nelle appAndroid. Gli autori di EARMO, tuttavia, nella realizzazione del tool hanno dato maggioreattenzione agli smell definiti da Fowler, e quindi maggiormente legati alla OOP, non tenendoconto degli altri smell più specifici della piattaforma Android

Catalogo di Reimann

1. **Nome Smell:** Bulk data transfer on slow network

   Colpisce: Efficienza energetica

   Problema: Il trasferimento dei dati su una connessione di rete più lenta consumerà

molta più energia rispetto a una connessione veloce.
Refactoring: Dovrebbe essere prima controllato su quale connessione di rete siamo.

2. **Nome Smell:** Data Transmission Without Compression
   Colpisce: Efficienza energetica, conformità dell'utente
   Problema: Trasmettere un file su un'infrastruttura di rete senza comprimerlo consuma più energia.
   Refactoring: Aggiungi la compressione dei dati alla trasmissione di file basata sul client HTTP Apache

3. **Nome Smell:** Debuggable Release
   Colpisce: Efficienza energetica, conformità dell'utente
   Problema:L'attributo android:debuggable in AndroidManifest.xml è impostato in fase di sviluppo per eseguire il debug dell'app. Ma è lasciato in tempo di rilascio. Questo è un grave problema di sicurezza.
   Refactoring: Rimuovere l' attributo android:debuggable o impostarlo su false in modo esplicito.

4. **Nome Smell:** Dropped Data
   Colpisce: Esperienza dell'utente, conformità dell'utente
   Problema: L'input dell'utente lattina o modificare i dati in un'attività o di frammento . Quindi si apre un'altra attività (chiamata in arrivo) e l'input viene perso.
   Refactoring: Salva lo stato dell'istanza.

5. **Nome Smell:** Durable WakeLock
   Colpisce: Efficienza energetica
   Problema: È necessario un WakeLock per comunicare al sistema che l'app deve mantenere il dispositivo acceso (per utilizzare CPU, sensori, GPS, rete, ecc.). Dopo aver utilizzato le risorse, l'applicazione dovrebbe rilasciare WakeLock. In caso contrario, la batteria si scaricherà.
   Refactoring: Acquisisci WakeLock con timeout

6. **Nome Smell:** Early Resource Binding
   Colpisce: Efficienza energetica
   Problema: se le risorse fisiche che consumano energia di un dispositivo Android vengono richieste troppo presto, viene consumata più energia. Più precisamente, "troppo presto" significa quando vengono richiesti nei metodi in esecuzione prima che l'utente

interagisca con l'app.

Refactoring: Spostare la richiesta di risorsa nel metodo visibile

7. **Nome Smell:** Inefficient Data Structure

   Colpisce: Efficienza

   Problema: L'uso di HashMap<Integer, Object> è lento (mappatura da un intero a un oggetto).

   Refactoring: Usa una struttura dati efficiente

8. **Nome Smell:** Inefficient SQL query

   Colpisce: Efficienza

   Problema: Per recuperare i dati da un database, viene inviata una query SQL tramite una connessione JDBC a un server remoto.

   Refactoring: Usa query JSON

9. **Nome Smell:** Inefficient data format and parser

   Colpisce: Efficienza

   Problema: L'uso di TreeParsers è lento.

   Refactoring: Utilizza un parser e un formato dati efficienti

10. **Nome Smell:** Internal Getter/Setter

    Colpisce: Efficienza

    Problema: I campi interni sono accessibili tramite getter e setter. Ma in Android il metodo virtuale è costoso.

    Refactoring: Accesso diretto al campo

11. **Nome Smell:** Interrupting from background

    Colpisce: Esperienza dell'utente, conformità dell'utente

    Problema: È un cattivo smell da cui iniziare attività BroadcastRecievers or Services che funzionano in background.

    Refactoring: Rimuovi startActivity () dal background.

12. **Nome Smell:** Leaking Inner Class

    Colpisce: Efficienza della memoria

    Problema: Le classi interne non statiche contengono un riferimento alla classe esterna. Ciò potrebbe causare una perdita di memoria.

    Refactoring: Introdurre la classe statica

13. **Nome Smell:** Leaking Thread

    Colpisce: Efficienza della memoria

    Problema: In Java i thread sono radici GC, ovvero vengono conservati nel runtime e non vengono raccolti. Quindi, se un'attività avvia un thread e non lo interrompe, questo è considerato un bug. Poiché può perdere memoria.

    Refactoring: Introdurre Run Check Variable

14. **Nome Smell:** Member-Ignoring Method

    Colpisce: Efficienza

    Problema: Metodi non statici che non accedono a nessuna proprietà.

    Refactoring: Introdurre il metodo statico

15. **Nome Smell:** Nested Layout

    Colpisce: Efficienza, esperienza utente, tempo di avvio

    Problema: I layout con elementi che hanno l'attributo weight impostato devono essere calcolati due volte. Sebbene ogni nuovo elemento richieda l'inizializzazione, il layout e il disegno che analizzano LinearLayouts nidificati in profondità aumenteranno esponenzialmente il tempo di calcolo.

    Refactoring: Appiattisci i layout

16. **Nome Smell:** Network IO operations in main thread

17. **Nome Smell:** No low memory resolver

    Colpisce: Efficienza della memoria, stabilità, esperienza utente

    Problema: I sistemi mobili di solito hanno poca RAM e nessuno spazio SWAP per liberare memoria. Android fornisce un meccanismo per aiutare il sistema a gestire la memoria. Il metodo sovrascrivibile Activity.onLowMemory() viene chiamato quando tutti i processi in background sono stati terminati. Cioè, prima di arrivare al punto di uccidere i processi di hosting del servizio e dell'interfaccia utente in primo piano che vorremmo evitare di fare terminare.

    Questo metodo dovrebbe pulire le cache o le risorse non necessarie. Se questo metodo non è implementato, questo è considerato uno smell, poiché può causare la chiusura anomala del programma.

    Refactoring: Override onLowMemory Efficiency()

18. **Nome Smell:** Not descriptive UI

    Colpisce: Accessibilità, conformità dell'utente, esperienza dell'utente

    Problema: Per le persone con disabilità visive o fisiche è difficile navigare nell'app e tenere traccia di ciò che viene mostrato. Per aiutare queste persone, Android ha sviluppato TalkBack, un'app che legge i contenuti.

    Quindi ogni elemento dell'interfaccia utente dovrebbe avere una descrizione del contenuto.

    Refactoring: Imposta la descrizione del contenuto

19. **Nome Smell:** Overdrawn Pixel

    Colpisce: Efficienza

    Problema: Il layout di un'interfaccia utente Android viene generalmente creato utilizzando XML. Quindi puoi impilare diversi livelli per formare interfacce utente complesse. I contenitori e gli elementi di base possono avere attributi per descrivere il loro sfondo, testo, bordo, ecc. In questo caso potrebbe essere possibile sovrascrivere un pixel. Ciò significa, ad esempio, che il contenitore radice ha uno sfondo nero e contiene molti altri elementi che hanno un altro colore di sfondo (non trasparente). E in cima a questi ci sono i pulsanti, campi di testo anch'essi colorati. Ciò significa che un pixel deve essere disegnato più volte, il che è un processo costoso.

20. **Nome Smell:** Prohibited data transfer

    Colpisce: Efficienza energetica, conformità dell'utente, esperienza dell'utente

    Problema: Non viene verificato se l'utente ha disabilitato la trasmissione dei dati in background prima della trasmissione.

    Refactoring: Controlla il trasferimento dei dati in background

21. **Nome Smell:** Public data

    Colpisce: Sicurezza, conformità dell'utente, esperienza dell'utente

    Problema: I dati privati vengono conservati in un negozio accessibile pubblicamente (da altre applicazioni).

    Refactoring: Imposta la modalità privata

22. **Nome Smell:** Rigid AlarmManager

23. **Nome Smell:** Set config changes

    Colpisce: Efficienza della memoria, conformità dell'utente, esperienza dell'utente

Problema: È considerato uno smell impostare l'attributo android:configChanges nel manifest di Android. Questo attributo definisce quali modifiche alla configurazione l'app deve gestire manualmente (altrimenti il sistema si occuperà di conservare i dati di input tra le modifiche alla configurazione negli elementi dell'interfaccia utente delle attività). Gestirli manualmente può causare bug di memoria (lasciando risorse in memoria).
Refactoring: Usa frammenti per la modifica della configurazione

24. **Nome Smell:** Slow Loop
    Colpisce: Efficienza
    Problema: Viene utilizzata una versione lenta di un ciclo for.
    Refactoring: Enhanced for

25. **Nome Smell:** Tracking Hardware Id
    Colpisce: Sicurezza, conformità dell'utente, esperienza dell'utente
    Problema: Per alcuni casi d'uso potrebbe essere necessario ottenere un identificatore di dispositivo univoco, affidabile e univoco.
    Refactoring: Utilizza un ID generato univoco

26. **Nome Smell:** Uncached Views
    Colpisce: Efficienza
    Problema: Lo scorrimento di ListView o il passaggio tra le pagine di ViewPager potrebbe essere lento.
    Refactoring: View Holder

27. **Nome Smell:** Unclosed closable
    Colpisce: Efficienza della memoria
    Problema: Un oggetto che implementa il java.io.Closeable non è chiuso
    Refactoring: Chiudi Closable

28. **Nome Smell:** Uncontrolled focus order

29. **Nome Smell:** Unnecessary permission
    Colpisce: Sicurezza, conformità dell'utente, esperienza dell'utente
    Problema: L'app richiede diverse autorizzazioni, alcune delle quali non sono necessarie, in quanto possono essere sostituite facilmente.

Più autorizzazioni richiede un'app, più sospetta sarà per l'utente.

Refactoring: Usa l'activity intent

30. **Nome Smell:** Untouchable

    Colpisce: Esperienza utente, accessibilità

    Problema: Se la dimensione di un elemento dell'interfaccia utente tattile è inferiore a 48 dp (circa 9 mm).

CAPITOLO 3

Il tool aDoctor

3.1 Prima Versione

La prima versione del tool aDoctor(acronimo per AnDrOid Code smell detecTOR) era un software per l'analisi di codice Java in grado di rilevare diversi smell specifici della piattaforma Android, presenti nel catalogo di Reimann. Il tool nasce come applicazione in grado di avviare l'analisi dell'intero codice di un progetto Java datogli in input, ritornando una tabella che elenca i vari smell presenti nel codice ed in quali classi erano stati individuati. ADoctor era in grado di compiere l'analisi dell'intero codice sorgente di un'applicazione attraverso gli Abstract Syntax Tree ("AST" in seguito), strutture esplorabili dalle classi presenti nell'API Eclipse JDT (Java Development Toolkit) [9]. Questa API viene utilizzata, principalmente per il suo package core, contenente molteplici classi per lavorare sul codice sorgente di un'applicazione Java. Con questo package è possibile convertire del codice sorgente Java in un AST, che è una struttura dati ad albero i cui nodi rappresentano i singoli costrutti del linguaggio. Gli AST ci sono particolamente utili perchè individuano gli errori semantici, compresi bug e smell. Un AST, una volta costruito, puo essere esplorato secondo il design pattern Visitor per ottenere informazioni dugli elementi di cui si compone, e quindi poter condurre la ricerca degli smell.

3.2 Seconda Versione

Dopo la prima versione di aDoctor si è arrivati all'ultima versione realizzata da Iannone et al. [48], che consente, una volta effettuata la identificazione degli smell, di effettuarne il refactoring, mostrando all'utente sia lo smell individuato all'interno del codice sia il refactoring proposto da aDoctor. In questo lavoro viene analizzato il tool aDoctor, usato come punto di partenza per lo sviluppo della sua nuova versione. La seconda versione di aDoctor (dove ora l'acronimo sta per AnDrOid Code smell detecTiOn and Refactoring) usa la stessa API usata nella prima versione del tool, ma usando altre classi, prima tra tutte ASTRwrite che è in grado di modificare i nodi di un AST e di convertire le sue modifiche in un nuovo codice sorgente Java, che rimpiazzerà quello originario, realizzando così operazioni di refactoring automatico. La seconda versione di aDoctor fornisce all'utente una nuova interfaccia grafica, semplice ed intuitiva, che guida meglio l'utente verso l'identificazione e la correzione dei code smell. Gli smell che il tool riesce ad identificare vengono mostrati allo sviluppatore tramite una lista. Da questa lista è possibile selezionarlo uno, portando aDoctor a mostrare a schermo il codice contenente lo smell affianco alla relativa proposta di refactoring. Il plugin non è in grado di correggere errori che non riguardano strettamente gli smell, quindi, affinchè il tool funzioni correttamente, è richiesto che non ci siano errori di compilazione nel codice analizzato. A differenza della prima, questa versione è in grado di analizzare un intero progetto di un app'Android e di rilevare 6 code smell, i cui dettagli sono riportati nelle sezioni seguenti.

3.2.1 Durable Wakelock

Definizione

Questo smell è stato il primo implementato nella seconda versione del tool. Un wakelock è un meccanismo che permette ad un'app di continuare ad essere eseguita anche con il dispositivo in stato di sospensione, Andrebbe rilasciato non appena terminato il task per cui è stato acquisito. Questo smell si ha se un metodo non rilascia il wakelock, una volta terminate le sue operazioni. Per evitare un consumo inutile della batteria, il dispositivo Android, che non riceve input da parte dell'utente e quindi risulta inattivo va in standby. Esistono tre livelli di sospensione:

1. Si abbassa la luminosità ordinaria del device
2. Si disattiva lo schermo

3. Si disattiva la CPU

Quando un'app deve mantenere attiva la CPU per completare un lavoro in background,l'API Android fornisce i wakelock che possono essere acquisiti per mantenere il dispositivo in attività. Un wakelock dovrebbe essere acquisito soltanto quando necessario, quindi è buona pratica specificare un timeout di rilascio o rilasciarlo esplicitamente.

Identificazione

Una classe ha un DW smell quando ha un metodo con un'istanza di PowerManager.WakeLock che acquisisce una wakelock senza impostare un timeout di rilascio e senza successivamente chiamare il metodo release() all'interno dello stesso scope. Qui di seguito uno snippet di codice dove è presente un metodo con DurableWakelock.

```
...
PowerManager pm = (PowerManager)
getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wakeLock1 =
pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "wakeLock1");
wakeLock1.acquire(); // acquire() has no timeout
// ... do work...
// missing: wakeLock1.release();
...
```

Soluzione

Per risolvere un Durable Wakelock è necessario inserire il metodo release() dopo l'ultimo statement del blocco di codice in cui è stato usato il metodo acquire() oppure inserire un timer in millisecondi all'interno del metodo acquire().

```
...
PowerManager pm = (PowerManager)
getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wakeLock1 =
pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "wakeLock1");
wakeLock1.acquire(); // acquire() has no timeout
// ... do work...
wakeLock1.release();
...
```

3.2.2 Early Resource Binding

Definizione

Spesso un'app ha bisogno di accedere ad alcune informazioni provenienti da sensori installati sul device, oppure da altri servizi forniti dal sistema operativo, chiamati System Services. L'accesso troppo anticipato alle informazione fornite dai servizi di sistema comporta uno spreco energetico. Ad esempio, accedere alle informazioni relative alla posizione in un'activity che non è ancora visibile a schermo risulta inutile, poichè non è ancora possibile mostrare all'utente mappe o coordinate. Non esiste una metrica per capire quanto presto una risorsa viene usata, tuttavia una cattiva pratica diffusa è quello di accedere ad un servizio di sistema nel metodo onCreate(Bundle) di un'activity, ovvero quando essa non è ancora visibile a schermo, cosa che accadrà solo al termine del metodo onResume().

Identificazione

Si ha Early Resource Binding se nel metodo onCreate(Bundle) si accede alle informazioni di un servizio di sistema. Nello snippet di codice qui di seguito viene mostrato un Early Resource Binding dovuto dal fatto che nel metodo onCreate(Bundle) è presente un'istanza di LocationManager tramite la quale viene invocato il metodo requestLocationUpdates().

```
private LocationManager lm;
protected void onCreate(Bundle savedInstanceState) {
   lm = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
   if(lm.getAllProviders().contains(LocationManager.GPS_PROVIDER))
   {
      if(lm.isProviderEnabled(LocationManager.GPS_PROVIDER))
         {
            lm.requestLocationUpdates
               (LocationManager.GPS_PROVIDER,1000,0,
                  (LocationListener) this);
         }
   }
}
public void onResume() {
// ... do work...
}
```

Refactoring

Si risolve un ERB smell impedendo all'onCreate(Bundle) di accedere al servizio di sistema e di farlo nel metodo onResume() della stessa activity. E' comunque permesso istanziare e preparare il servizio nel metodo onCreate(Bundle).

```
private LocationManager lm;
protected void onCreate(Bundle savedInstanceState) {
   lm =
      (LocationManager)getSystemService(Context.LOCATION_SERVICE);
      if(lm.getAllProviders().contains(LocationManager.GPS_PROVIDER))
         {
            if(lm.isProviderEnabled(LocationManager.GPS_PROVIDER))
            {
            //eventuale controllo che indica la disponibilit del servizio
            }
         }
}
public void onResume() {
// ... do work...
   lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,1000,
      0, (LocationListener) this);
}
```

3.2.3 Internal Setter

Definizione

I metodi setter sono una componente fondamentale dell'OOP. Di solito accettano un solo solo argomento assegnato a una variabile di istanza. Un metodo non statico della stessa classe che chiama un setter con una sola assegnazione, compie uno sforzo computazionale inutile, in quanto ha i diritti di accesso per effettuare un'assegnazione diretta su quella proprità, che potrebbe causare una perdita di energia.

Identificazione

Si ha un Internal Setter quando viene usato un metodo setter su un istanza dichiarata all'interno della classe in cui si sta lavorando, come mostrato dal seguente snippet di codice.

```
...
private int count;
public void setCount(int count)
{
   this.count=count;
}
@Override
protected void onCreate(Bundle savedInstanceState) {
   setCount(3);
...
}
```

Refactoring

Un Internal Setter viene risolto aggiornando il valore dell'istanza direttamente anzichè usare metodi setter, come mostrato nel seguente snippet di codice.

```
...
private int count;
public void setCount(int count)
{
   this.count=count;
}
@Override
protected void onCreate(Bundle savedInstanceState) {
   this.count=3;
...
}
```

3.2.4 Leaking Thread

Definizione

Android RunTime (ART) tratta un istanza di thread attivo come una radice di Garbage Collector(GC), il che significa che la memoria non può essere recuperata. Ogni qualvolta che un thread viene interrotto (dalle chiamate stop() o interrupt()), cessa di essere trattato come una radice di un GC, diventando idoneo per la Garbage Collection.

Identificazione

Una sottoclasse di un Activity ha un LT smell quando mostra, in un metodo di classe, una variabile di istanza variabile di un thread che chiama il metodo start() ma non interrupt(). Nel seguente snippet viene identificato un Leaking thread.

```
Thread t;
@Override
protected void onCreate(Bundle savedInstanceState) {
   t= new Thread();
   t.start();
}
public void onDestroy() {
   super.onDestroy();
}
...
```

Refactoring

Un Leaking Thread si risolve chiamando il metodo .interrupt() all'istanza Thread utilizzata, come fatto nel seguente snippet di codice.

```
...
Thread t;
@Override
protected void onCreate(Bundle savedInstanceState) {
   t= new Thread();
   t.start();
}
public void onDestroy() {
   super.onDestroy();
   t.interrupt();
}
...
```

3.2.5 Member Ignoring Method

Definizione

Un metodo statico è piu veloce dell'equivalente non statico. Un metodo statico non accede ad alcuna proprietà interna (cioè variabili d'istanza e metodi non statici). Pertanto, se un metodo non statico non fa accedere a qualsiasi proprietà interna della sua classe di appartenenza, dovrebbe essere impostato come statico.

Identificazione

Una classe ha un MIM smell quando 1. ha un metodo non statico e non vuoto che non accede a variabili d'istanza 2. non usa "this" e la parola chiave "super" 3. non effettua Override di metodi Di seguito viene mostrato uno snippet di codice dove compare questo smell.

```
public class metodign {
   public void foo() {
      int i=0;
      for (i;i<5;i++){
         System.out.println("tesi di Antonio de Matteo");
      }
   }
}
```

Refactoring

Un MIM smell può essere corretto aggiuggendo la parola chiave "static" alla dichiarazione del metodo.

```
public class metodign {
   public static void foo() {
      int i=0;
      for (i;i<5;i++){
         System.out.println("tesi di Antonio de Matteo");
      }
   }
}
```

3.3 Funzionalità di aDoctor

I requisiti funzioni di aDoctor si possono riassumere schematicamente in questo modo:

1. Permettere all'utente di avviare il tool dalla sezione "refactoring" di Android Studio.
2. Permettere all'utente di iniziare l'analisi dell'intero codice del progetto aperto in quel momento.
3. Far scegliere all'utente quale smell individuare dalla lista degli smell implementati in aDoctor.
4. Mostrare all'utente il codice da lui scritto contenente quei particolari smell affiancato dal codice che ha subito il refactoring.
5. Dar la possibilità di scegliere la classe da visualizzare tra quelle contenenti smell.
6. Dar la possibilità all'utente di applicare la proposta di refactoring,modificando quindi il codice scritto.

Il tool è stato progettato con l'obiettivo di essere il usabile possibile, cercando di far avere al sistema il minor numero di interazioni con l'utente. L'utente da quando avvia il tool riesce sempre a sapere cosa sta accadendo, grazie ad un interfaccia semplice ed intuitiva. Un altro importante requisito non funzionale é quello della sopportabilitá: il sistema dovrà essere facilmente manutenibile ed estendibile, ovvero favorire le modifiche correttive ed evolutive del codice. Questo permette di implementare altri smell con un numero minimale di modifiche al codice giá presente.

3.4 Flusso di esecuzione

Il flusso di esecuzione di aDoctor è molto semplice. Attraverso una serie di figure, viene illustrato tale flusso. Per avviare il plugin è necessario accedere al menu "Refactor" dell'IDE situato nella barra in alto come mostrato nella figura 3.1.

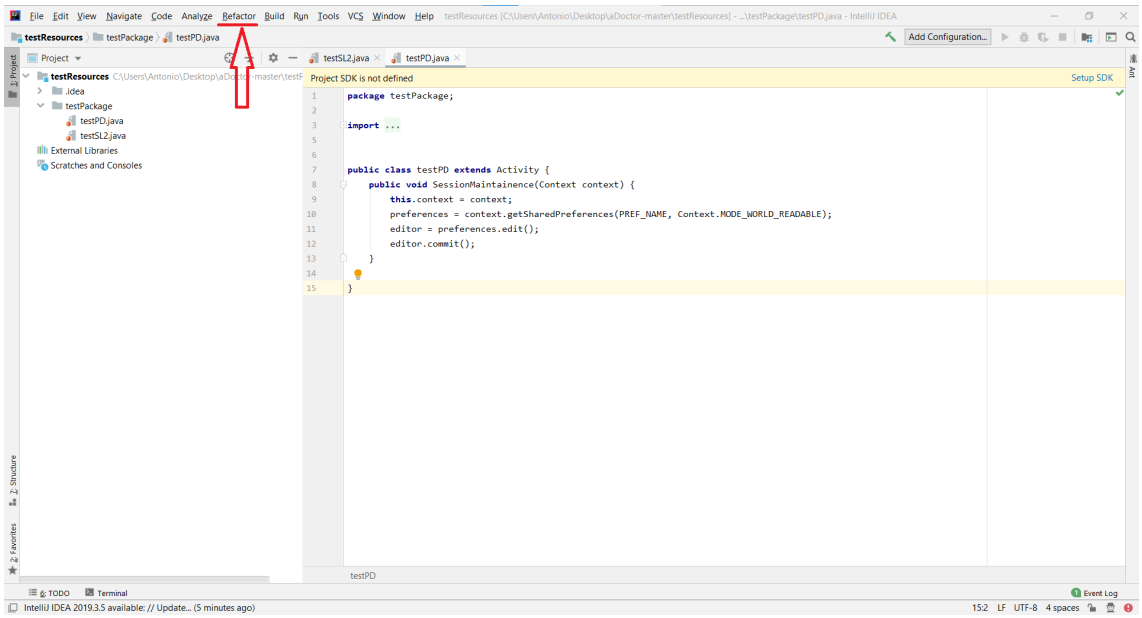

Figura 3.1: Schermata principale di Intellij

Successivamente, dal menu di "Refactor" si clicca su aDoctor per avviarne l'esecuzione come mostrato nella Figura 3.2.

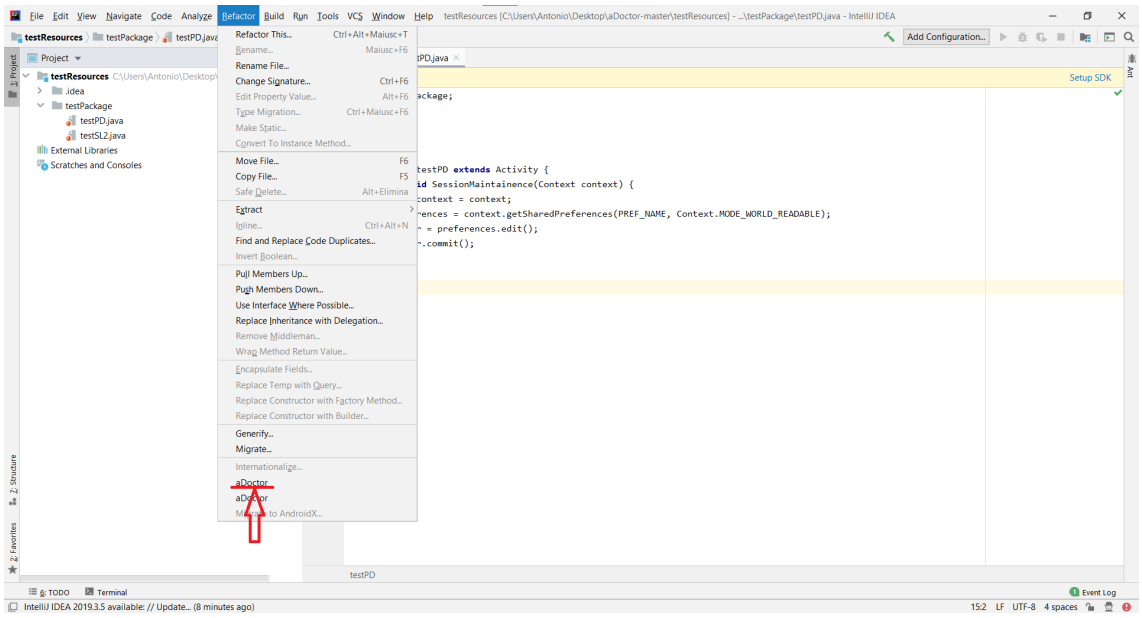

Figura 3.2: Menu di Refactor

Dopo aver avviato l'esecuzione, viene mostrata una finestra dove è possibile selezionare gli smell che si vogliono trovare, una volta scelti, bisogna premere "Run", come mostrato in Figura 3.3.

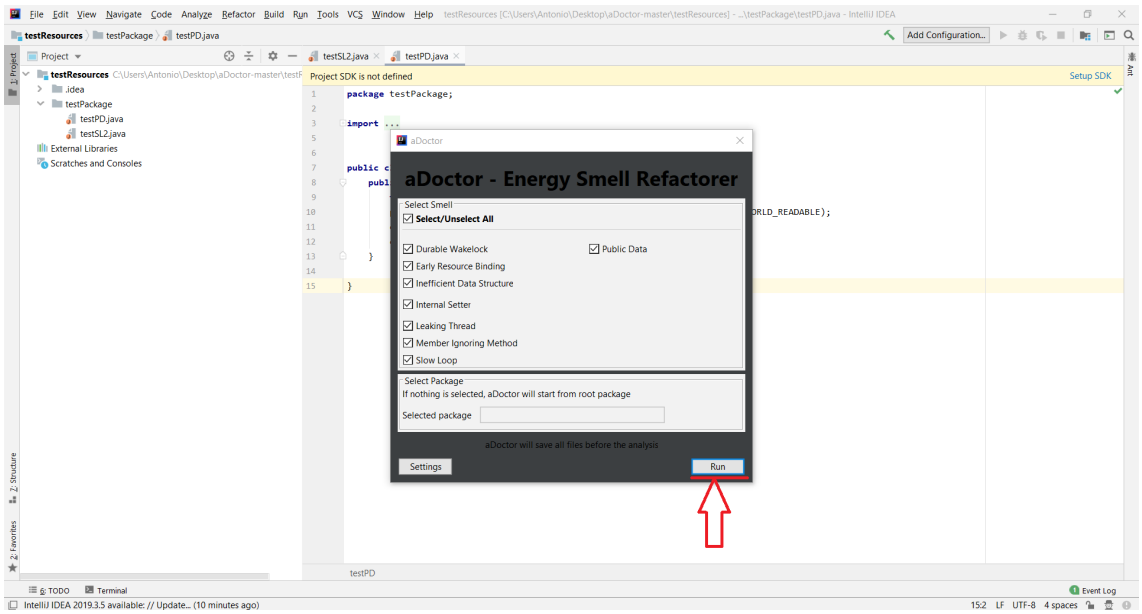

Figura 3.3: Menu principale di aDoctor

Sulla desta viene mostrato da una parte il codice scritto e dall'altra il codice dopo il refactoring. (2) L'utente una volta aver capito l'errore, preme su "Apply" ed effettua le modifiche. Come mostrato in figura 3.4.

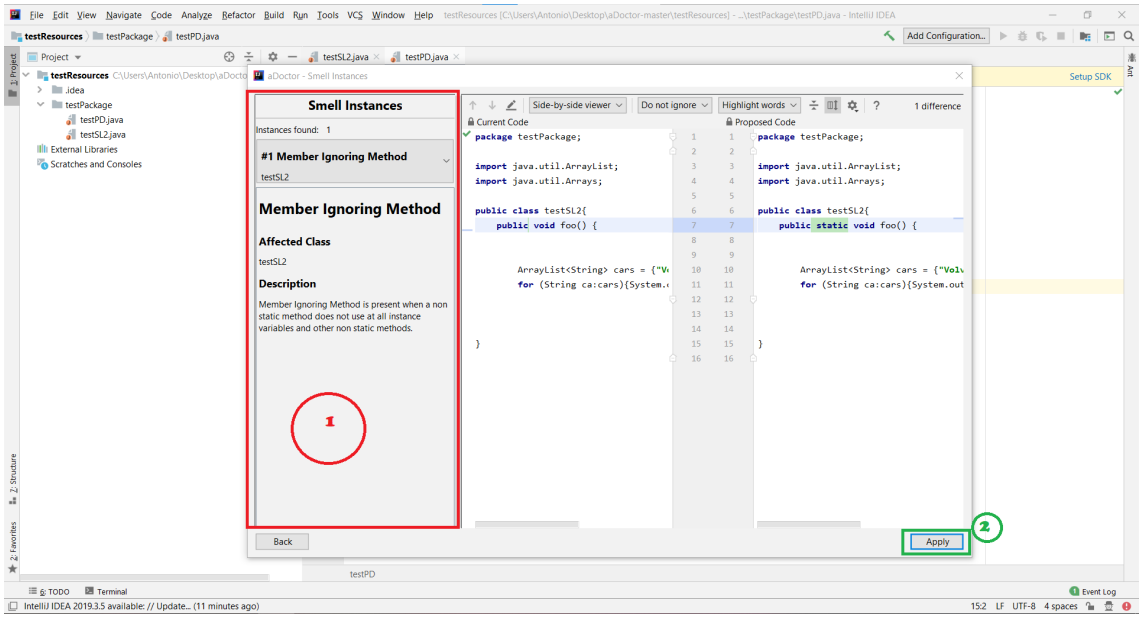

Figura 3.4: Finestra dove vengono applicate le modifiche

Terminato con successo il refactoring, viene mostrata questa finestra, dalla quale è possibile rifare un'altra analisi del codice o terminare aDoctor, come mostrato in Figura 3.5

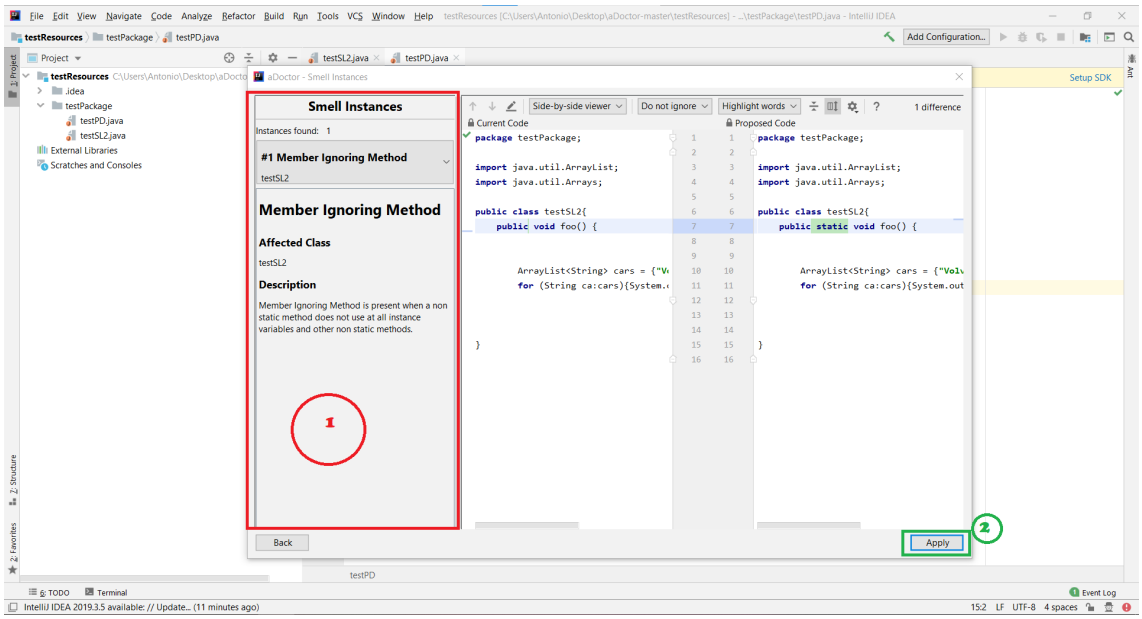

Figura 3.5: Finestra dove vengono applicate le modifiche

3.5 Casi d'uso

3.5.1 Selezione Smell

Use case name	Selezione smell.
Participating Actors	Utente
Flow of events	1. L'utente accede ad Android Studio e premendo sulla voce refactoring del menù, avvia aDoctor. 2. Il sistema visualizza una finestra con al di sopra tutti i possibili smell risolvibili con aDoctor, al di sotto dove voler iniziare ad analizzare in mancanza di selezione. 3. L'utente seleziona lo/gli smell e avvia aDoctor.
Entry condition	• L'utente avvia il tool.
Exit condition	• L'utente preme il tasto "run".
Exceptions	• L'utente non ha selezionato nulla

Figura 3.6: Selezione Smell

3.5.2 Ricerca classe

Use case name	Ricerca classe
Participating Actors	Utente
Flow of events	1. L'utente si trova sulla finestra di refactoring 2. Il sistema mostra nella finestra di refactoring sul lato destro, la sezione contenenti le classi 3. L'utente seleziona il campo 4. Il sistema mostra tutte le classi del progetto con relativo nome e smell trovato 5. L'utente seleziona la classe cercata 6. Il sistema mostra la classe con codice scritto dall'utente affiancata a quella con codice dopo il refactoring
Entry condition	• L'utente si trova nella pagina di refactoring
Exit condition	• L'utente seleziona la classe voluta

Figura 3.7: Ricerca classe

3.5.3 Applicazione proposta di refactoring

Use case name	Applicazione proposta di refactoring
Participating Actors	Utente
Flow of events	1. L'utente preme il tasto relativo alla conferma 2. Il sistema effettua il refactoring del codice.
Entry condition	• L'utente attiva la funzione di conferma
Exit condition	• Il sistema corregge il codice

Figura 3.8: Applicazione proposta di refactoring

3.6 Progettazione e architettura del sistema

L'infrastruttura di aDoctor è stata implementata con IntelliJ Platform SDK che, rende disponibile un insieme di librerie che consentono l'estensione della piattaforma IntelliJ

creando plugin, supporto a linguaggi personalizzati o ad un IDE personalizzata [1]. In questo caso aDoctor risulta essere un plugin per Android Studio che ha accesso ai tool built-in e menu della piattaforma IntelliJ. La figura 3.9 mostra l'architettura del plugin divisa in tre layer. Il layer di Presentazione chiamato presentation, si occupa delle interazioni utente realizzate tramite le finestre di dialogo costruite con i componenti Swing di Java, supportati dall'IntelliJ Platform SDK. Il controllo della logica globale è centralizzato in una singola classe, chiamata CoreDriver, che seleziona i giusti dialoghi da mostrare, spostando i dati tra di loro tramite callback interface. Il layer della logica applicativa chiamato application contiene la logica di business. Il sottosistema Analysis permette la rilevazione dei code smell e prepara le proposte di refactoring. aDoctor non ha bisogno di salvare nessun dato in modo persistente, quindi, attraverso IntelliJ Platform SDK, accede solamente al file system locale, scrivendo e leggendo i file. aDoctor è basato sulla libreria Eclipse JDT CORE per il parsing dei file java estraendone i loro AST . Nel layer presentation abbiamo i seguenti sottosistemi:

1. Presentation: layer di presentazione, si occupa delle interazioni con l'utente, realizzate attraverso delle finestre di dialogo costruite con Java Swing. In questo layer abbiamo i seguenti sottoinsiemi:

    - action, si occupa dell'avvio del plugin attraverso il menu a tendina di refactoring dell'IDE.
    - dialog, contiene tutte le diverse finestre di dialogo con le quali l'utente interagisce.

2. Application: layer della logica applicativa

    - analysis, si occupa di analizzare il codice sorgente dell'app al fine di cercare gli smell
    - proposal, si occupa di calcolare le proposte di refactoring sulla base degli smell individuati.
    - refactoring, si occupa di applicare le proposte di refactoring, riscrivendo il codice.
    - bean, si occupa di rappresentare come oggetti gli smell e le proposte di refactoring, associando loro informazioni aggiuntive.

3. Storage layer: layer dei dati persistenti

[1] https://www.jetbrains.com/idea/

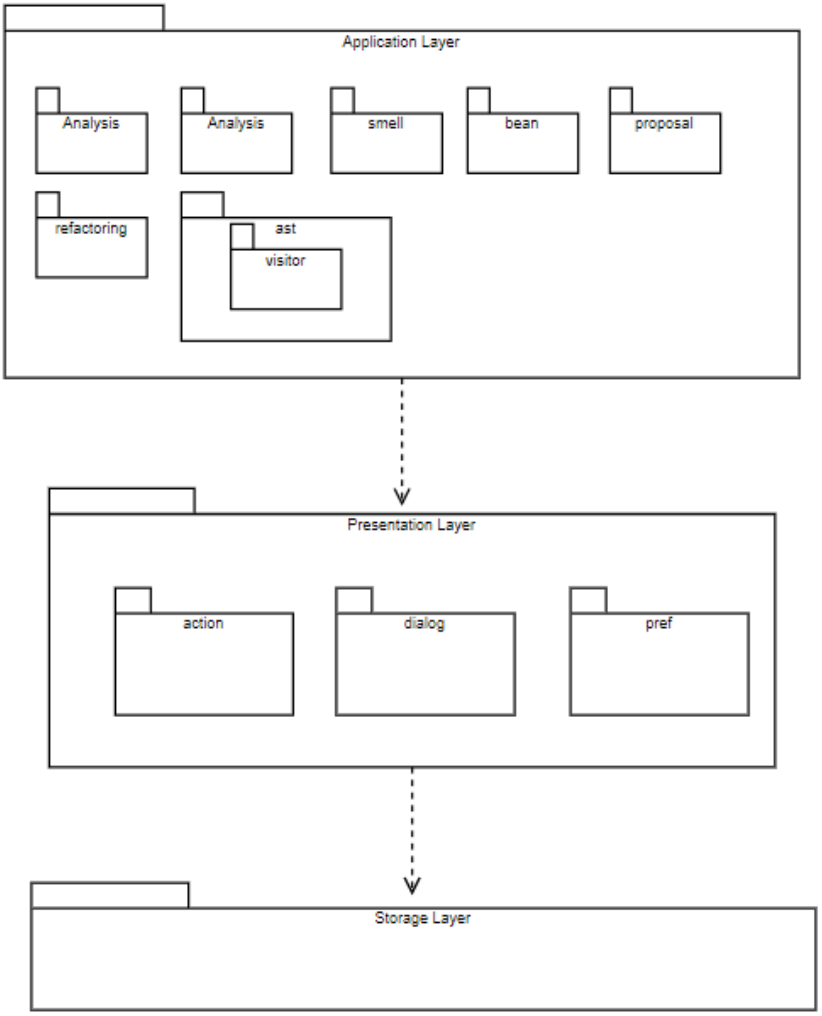

Figura 3.9: Architettura di aDoctor

3.6.1 Analyzer

Il package Analyzer, è il package di aDoctor fondamentale per le azioni di detection, nella figura 3.10 sottostante viene mostrato il class diagram.

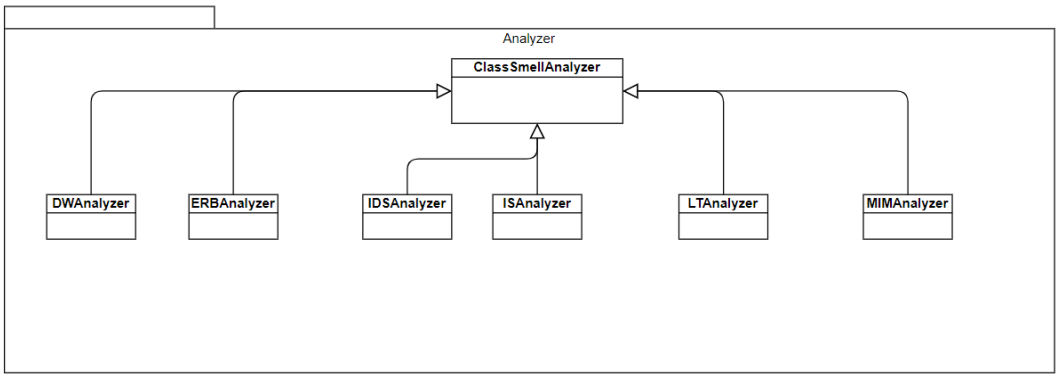

Figura 3.10: Class Diagram Analyzer

Ognuna delle 6 classi di detection che vanno a comporre il package Analyzer è apposita per uno specifico smell e va ad estendere la superclasse ClassSmellAnalyzer.

3.6.2 Proposer

Il package Proposer, è il package di aDoctor fondamentale per le azioni di refactoring, nella figura 3.11 sottostante viene mostrato il class diagram.

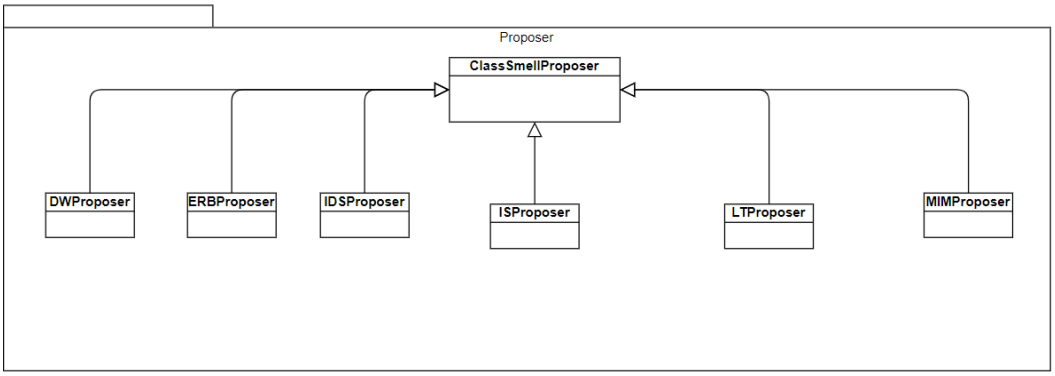

Figura 3.11: Class Diagram Proposer

Ognuna delle 6 classi di refactoring che vanno a comporre il package è apposita per uno specifico smell e va ad estendere la superclasse ClassSmellProposer.

3.6.3 Pattern Visitor

Entrambi i package sopra descritti sono supportati dal pattern Visitor. Il pattern Visitor è gia utilizzato nella libreria JDT Core e in aDoctor è usato nella classe ASTUtilies per effettuare analisi sul codice. Il Visitor [48] è un design pattern comportamentale utilizzato nell'OOP. Permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa. Visitor è utile quando:

- una struttura di oggetti è costituita da molte classi con interfacce diverse ed è necessario che l'algoritmo esegua su ogni oggetto un'operazione differente a seconda dalla classe concreta dell'oggetto stesso,
- è necessario eseguire svariate operazioni indipendenti e non relazionate tra loro sugli oggetti di una struttura composta, ma non si vuole sovraccaricare le interfacce delle loro classi. Riunendo le operazioni correlate in ogni Visitor è possibile inserirle nei programmi solo dove necessario.
- le classi che costituiscono la struttura composta sono raramente suscettibili di modifica, ma è necessario aggiungere spesso operazioni sui rispettivi oggetti. Ogni intervento sulle operazioni richiede la modifica o l'estensione di un Visitor, mentre ogni modifica alle classi della struttura vincola alla ridefinizione delle interfacce di tutti i Visitor, compito che può risultare estremamente complesso nei progetti di una certa dimensione.

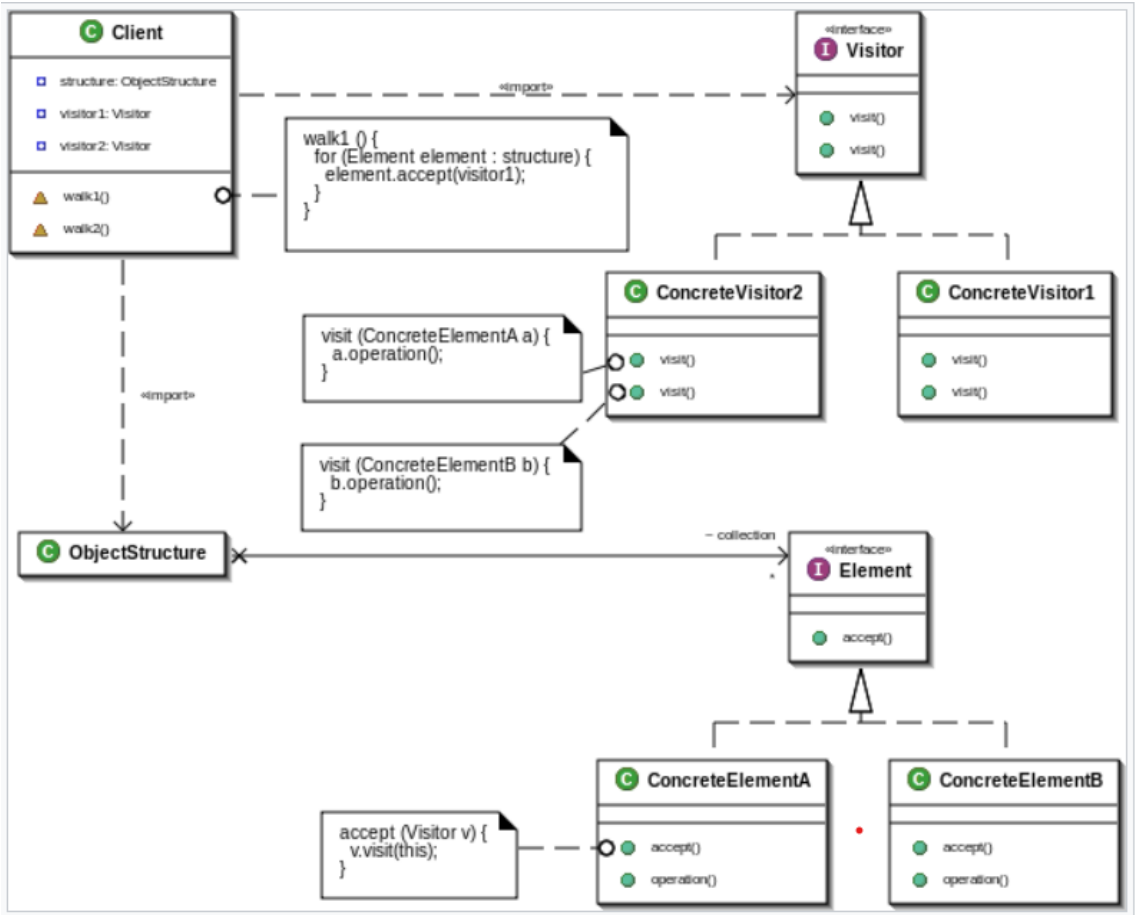

Figura 3.12: Class Diagram Design pattern Visitor

Visitor

Visitor dichiara un metodo visit per ogni ConcreteElement appartenente alla struttura di oggetti, in modo che ogni oggetto della struttura possa invocare il metodo visit appropriato passando un riferimento a sé (this) come parametro. Questo permette al Visitor di identificare la classe che ha chiamato il metodo visit, eseguire il comportamento corrispondente e accedere all'oggetto attraverso la sua specifica interfaccia.

ConcreteVisitor

ConcreteVisitor implementa le operazioni visit dichiarate da Visitor perché agiscano come desiderato sulle rispettive classi. Inoltre fornisce il contesto dell'algoritmo e ne mantiene in memoria lo stato, che spesso accumula i risultati parziali ottenuti durante l'attraversamento della struttura.

Element

Element definisce un'operazione accept utilizzata per "accettare" un Visitor passato come parametro.

ConcreteElement

ConcreteElement implementa la accept definita da Element. In generale accept chiama il metodo visit del Visitor ricevuto, passando come parametro un riferimento a sé.

ObjectStructure

ObjectStructure contiene ed elenca gli elementi. Quando necessario può fornire un'interfaccia d'alto livello che permetta al Visitor di visitare i singoli Element. Può essere implementata applicando il pattern Composite, oppure utilizzando una collezione come ad esempio un array o qualsiasi altra struttura dati.

CAPITOLO 4

Estensione di ADoctor

In questo capitolo viene spiegato nel dettaglio i nuovi smell introdotti in questa versione: Slow Loop Smell (SL) e Public Data (PD) descrivendo il problema, proponendo poi una possibile proposta di detection e refactoring con le sue limitazioni. In questa versione di aDoctor vengono implementati i nuovi smell Slow Loop (SL) e Public Data (PD).

4.1 Slow Loop

Definizione

Quando si programma un' applicazione Android, usare gli standard for loop come ad esempio *for(int i=0;i< struttura.size();i++* potrebbe causare rallentamenti nell'esecuzione non garantendo la giusta efficienza [1] del codice dell'app. Un for_statement è composto da tre parti: index, expression, updater.

- L'index è la parte iniziale dove si inizializza la variabile del for, viene eseguito una volta all'inizio del for
- Quando l'expression di terminazione è uguale a *false*, il for termina
- L'updater è invocato dopo ogni iterazione attraverso il for; in questa espressione si può incrementare o decrementare il valore.

[1] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html

Identificazione

Il sistema individua un SL smell nel momento in cui trova, una versione standard di for che opera su una struttura complessa come un Array,ArrayList, etc... Di seguito verranno presentati gli snippet di codice per ogni parte della detection, divisi nelle tre parti in cui è composto un for_statement.

Individuazone del for statement e rilevazione dell'indice

```
for (MethodDeclaration methodDecl : methods) {
   List<Statement> stat =
       (List<Statement>)methodDecl.getBody().statements();
      for (Statement s : stat) {
         if (s.getNodeType() == ASTNode.FOR_STATEMENT) {
            ForStatement forstat = (ForStatement) s;
            if(forstat.initializers().size()==1) {
               List<Expression> inits=
                   (List<Expression>)forstat.initializers();
               if(inits.get(0).getNodeType()==
                  ASTNode.VARIABLE_DECLARATION_EXPRESSION){
                  VariableDeclarationExpression t=
                      (VariableDeclarationExpression)
                      forstat.initializers().get(0);
                  List <VariableDeclarationFragment> frag=
                      (List<VariableDeclarationFragment>) t.fragments();
                  if(frag.get(0).getInitializer()
                     .getNodeType()==ASTNode.NUMBER_LITERAL){

                     if(frag.get(0).getInitializer()
                        .toString().equals("0")){
                        SimpleName simpleName= frag.get(0).getName();
                        slSmell.setIndex(simpleName);
```

Rilevazione dell'expression

```
String type = t.getType().toString();
if(type.equals("int")) {
```

```
if (forstat.getExpression().getNodeType() ==
   ASTNode.INFIX_EXPRESSION) {
   InfixExpression infixExpression = (InfixExpression)
      forstat.getExpression();
   if(infixExpression.getLeftOperand().toString()
      .equals(simpleName.toString())){
      if(infixExpression.getOperator() ==
         InfixExpression.Operator.LESS) {
         if(infixExpression.getRightOperand().getNodeType()==
            ASTNode.QUALIFIED_NAME) {
            QualifiedName qualiName= (QualifiedName)
               infixExpression.getRightOperand();
            if(qualiName.getName().getIdentifier()
               .equals("length")){
               if(qualiName.getQualifier().resolveTypeBinding()
                  .isArray()) {
                  slSmell.setStructure(qualiName.getQualifier());
               }
            }
         }
         if(infixExpression.getRightOperand().getNodeType()
         == ASTNode.METHOD_INVOCATION) {
            MethodInvocation methodInvocation= (MethodInvocation)
               infixExpression.getRightOperand();
            if(methodInvocation.getName().getIdentifier()
             .equals("size")){
               slSmell.setStructure(methodInvocation
                .getExpression());
```

Rilevazione dell'updater

```
if(forstat.updaters().size()==1){
   List<PostfixExpression> postfix= (List<PostfixExpression>)
      forstat.updaters();
   if(postfix.get(0).getOperand().toString().equals(simpleName.
    toString())){
```

```
if(postfix.get(0).getOperator()==
    PostfixExpression.Operator.INCREMENT){
    slSmell.setClassBean(classBean);
    slSmell.setForStatement(forstat);
    return slSmell;
```

Refactoring

Il refactoring di questo smell risulta molto semplice ed intuitivo. Quando aDoctor identifica lo smell effettua un cambio, sostituendo lo standard for con un enhanced for. Un esempio di enhanced for è il seguente *for(int i: struttura)*, questo tipo di for è particolarmente utile quando si hanno delle strutture come ad esempio degli Array o degli ArrayList. aDoctor effettua questo refactoring con step immediati, crea una nuova struttura *enhancedForStatement* per poi creare e modificare i campi del for originario con il fine di poterli sostituire con quelli del nuovo for creato. Un'importante distinzione che bisogna fare in questo refactoring è quella nelle implementazioni nel caso in cui il tool si trovasse ad operare su un array o su una struttura ArrayList.

Array

```
if (slSmell.getStructure().resolveTypeBinding().isArray()) {
        ITypeBinding fieldiTypeBinding = slSmell.getStructure()
        .resolveTypeBinding().getElementType();
        String nameClassString = fieldiTypeBinding.getName();
        Type t = targetAST.newSimpleType(targetAST
        .newSimpleName(nameClassString));
        singleVariableDeclaration.setType(t);
...
if (slSmell.getStructure().resolveTypeBinding().isArray()) {
        List<ArrayAccess> arrayAccesses = ASTUtilities.getArrayAccesses
        (enhancedForStatement.getBody());
        for (ArrayAccess arrayAccess : arrayAccesses) {
           if (arrayAccess.getArray().toString().equals
           (slSmell.getStructure().toString())) {
              SimpleName newSimpleName = targetAST.newSimpleName(ind);
```

```
                astRewrite.replace(arrayAccess, newSimpleName, null);
```

ArrayList

```
 ITypeBinding[] typeArray =
     slSmell.getStructure().resolveTypeBinding().getInterfaces();
    if (typeArray.length > 0) {
       ITypeBinding[] typeArgument = slSmell.getStructure().
       resolveTypeBinding().getTypeArguments();
       if (typeArgument.length == 1) {
          ITypeBinding a = typeArgument[0];
          String argumentName = a.getName();
          Type t = targetAST.newSimpleType
          (targetAST.newSimpleName(argumentName));
          singleVariableDeclaration.setType(t);
...
List<MethodInvocation> methodInvocations =
    ASTUtilities.getMethodInvocations
(enhancedForStatement.getBody());
    for (MethodInvocation methodInvocation : methodInvocations) {
       if (methodInvocation.getExpression() != null) {
          if (methodInvocation.getExpression().toString()
          .equals(slSmell.getStructure().toString())) {
             if (methodInvocation.getName().getIdentifier()
             .equals("get")) {
                List<Expression> arguments = (List<Expression>)
                   methodInvocation.arguments();
                if (arguments.size() == 1) {
                   if(arguments.get(0).toString().equals
                   (slSmell.getIndex().getIdentifier())) {
                            SimpleName newSimpleName =
                               targetAST.newSimpleName(ind);
                            astRewrite.replace(methodInvocation,
                               newSimpleName, null);
```

4.1.1 Scenario

1. Viene presentata ad aDoctor una classe affetta da Slow Loop Smell: Come mostrato nella figura 4.1
2. Selezionare nell'elenco degli smell la casella relativa a Slow Loop e cliccare sul pulsante "Run" come mostrato nella figura 4.2.
3. aDoctor mostrerà la proposta di refactoring, e l'utente accetterà. Come mostrato nella figura 4.3.
4. Il codice verrà cambiato. Come mostrato nella figura 4.4.

```
public class testSL2{
   public static void foo() {


      ArrayList<String> cars = {"Volvo", "BMW", "Ford", "Mazda"};
      for (int i=0; i<cars.size();i++){
         System.out.println(cars.get(i));
      }
   }
}
```

Figura 4.1: Classe affetta da SL smell

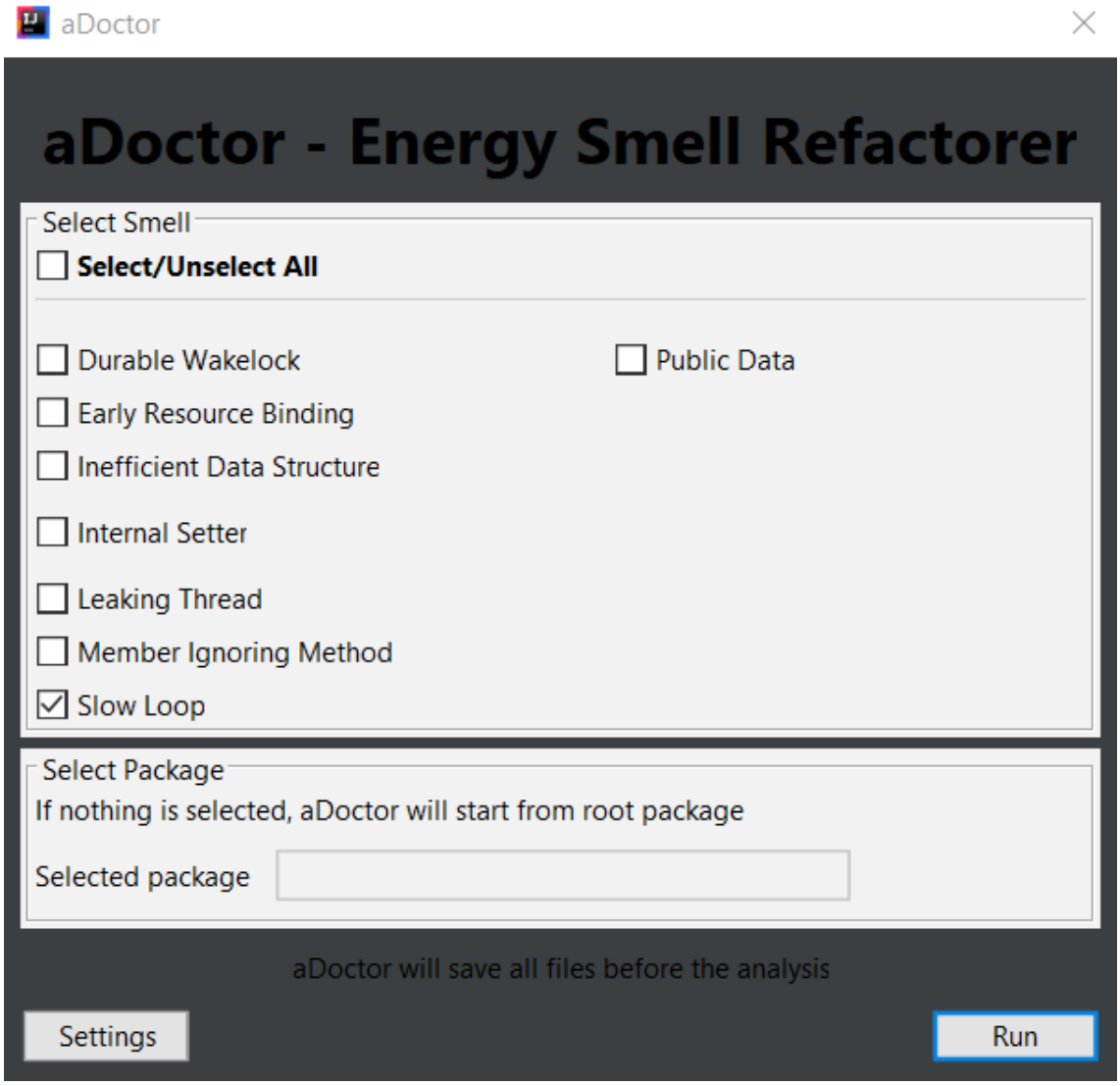

Figura 4.2: Selezione Slow Loop dalla schermata di aDoctor

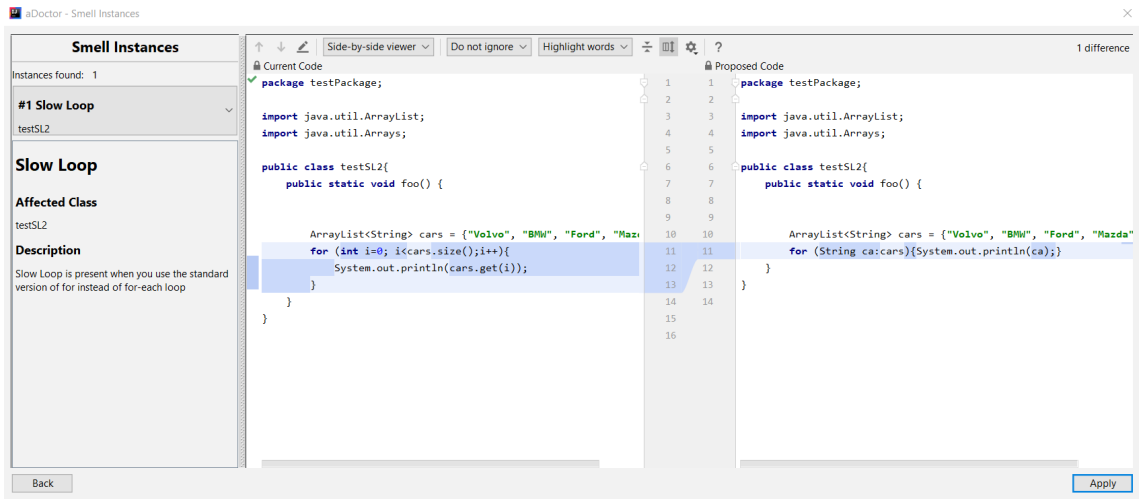

Figura 4.3: Proposta di refactoring

```
public class testSL2{
   public static void foo() {
      ArrayList<String> cars = {"Volvo", "BMW", "Ford", "Mazda"};
      for (String ca:cars){
         System.out.println(ca);}
   }
}
```

Figura 4.4: Codice dopo il refactoring

4.2 Public Data

Definizione

Questo smell si verifica quando i dati privati vengono conservati in una zona di memoria accessibile pubblicamente da altre applicazioni, potenzialemente minacciando la sicurezza dell'app.

Identificazione

In Android, questo smell viene identificato dal Context della classe come privato, nello specifico quando è presente il comando Context.MODE_WORLD_READABLE=True e Context.MODE_WORLD_WRITEABLE=True. Un Context è un'interfaccia per informazioni globali su un ambiente applicativo. Questa è una classe astratta la cui implementazione è fornita dal sistema Android. Consente l'accesso a risorse e classi specifiche dell'applicazione, nonché chiamate per operazioni a livello di applicazione come broadcasting e ricezione di intents, ecc. [2] In questa detection aDoctor dovrà visitare tutto il codice andando a cercare la presenza delle linee di codice colpevoli. Di seguito viene riportata la porzione di codice di detection.

```
if(superclassType.equals("Activity")){
   for(MethodDeclaration methodDeclaration: methods){
      Block body= methodDeclaration.getBody();
      List<Assignment> assignments= ASTUtilities.getAssigments(body);
      for(int i=0; i<assignments.size();i++){
         if(assignments.get(i).getOperator().toString()
```

[2] https://developer.android.com/reference/android/content/Context

```
.equals("=")){
  if(assignments.get(i).getLeftHandSide().toString()
      .equals("Context.MODE_WORLD_READABLE")||
      assignments.get(i).getLeftHandSide().toString()
          .equals("Context.MODE_WORLD_WRITEABLE") ){
      if(assignments.get(i).getRightHandSide().toString()
      .equals("true")){
          pdSmell.setClassBean(classBean);
          pdSmell.setAssignment(assignments.get(i));
          return pdSmell;
```

Refactoring

Il refactoring di questo smell è immediato, quando aDoctor rileva lo smell, dovrà semplicemente eliminare la linea di codice incriminata.

4.2.1 Scenario

1. Viene presentata ad aDoctor una classe affetta da Public Data Smell. Come mostrato nella figura 4.5
2. Selezionare nell'elenco degli smell la casella relativa a Slow Loop e cliccare sul pulsante "Run" come mostrato nella figura 4.6
3. aDoctor mostrerà la proposta di refactoring, l'utente dovrà accettarla o tornare indietro. Come mostrato nella figura 4.7.
4. Il codice verrà cambiato. Come mostrato nella figura 4.8.

```
public class testPD extends Activity {
   public void SessionMaintainence(Context context) {
      Context.MODE_WORLD_READABLE=true;
      this.context = context;
      preferences = context.getSharedPreferences(PREF_NAME,
         Context.MODE_WORLD_READABLE);
      editor = preferences.edit();
      editor.commit();
   }

}
```

Figura 4.5: Classe affetta da PD smell

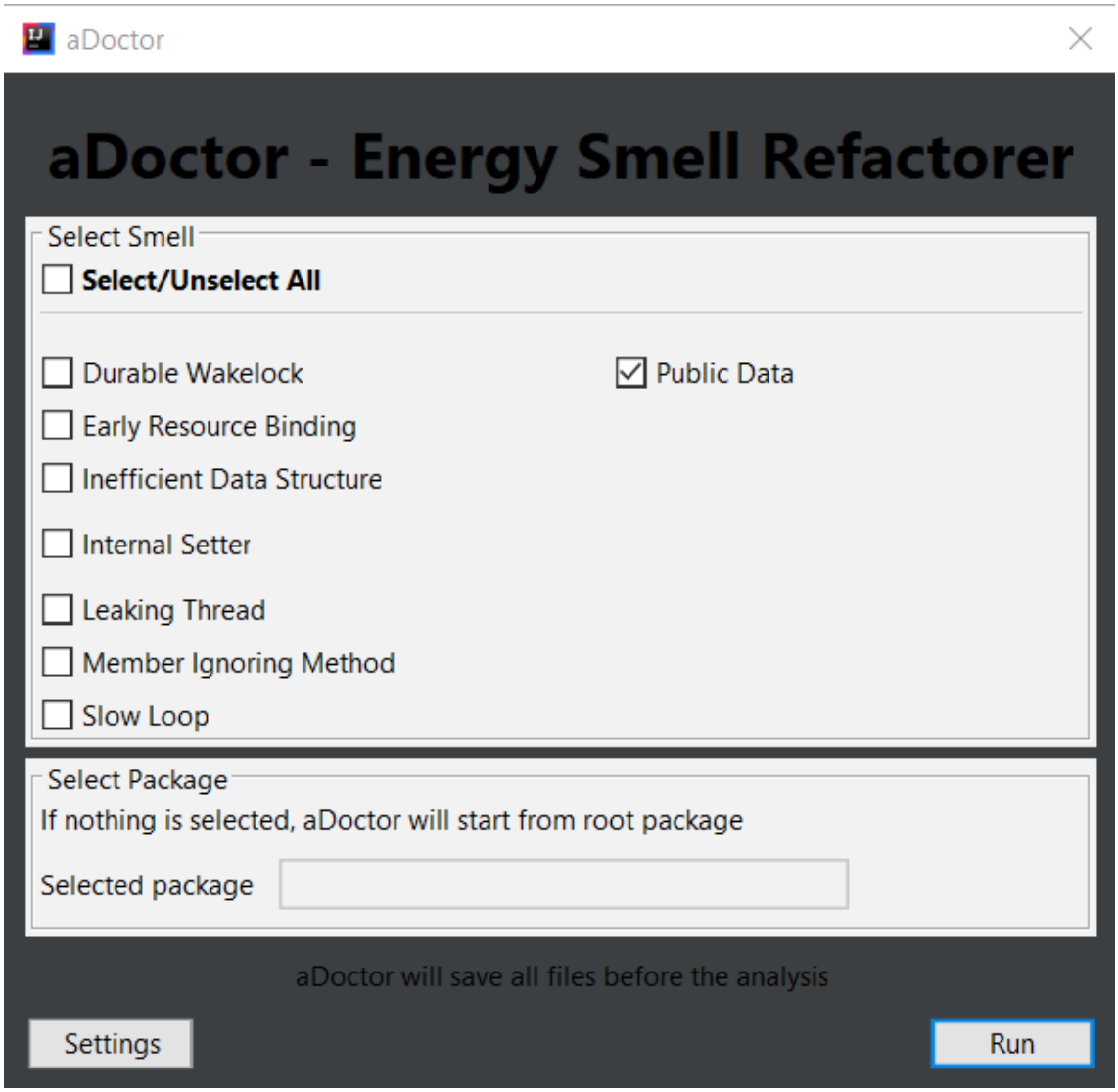

Figura 4.6: Selezione Public Data dalla schermata di aDoctor

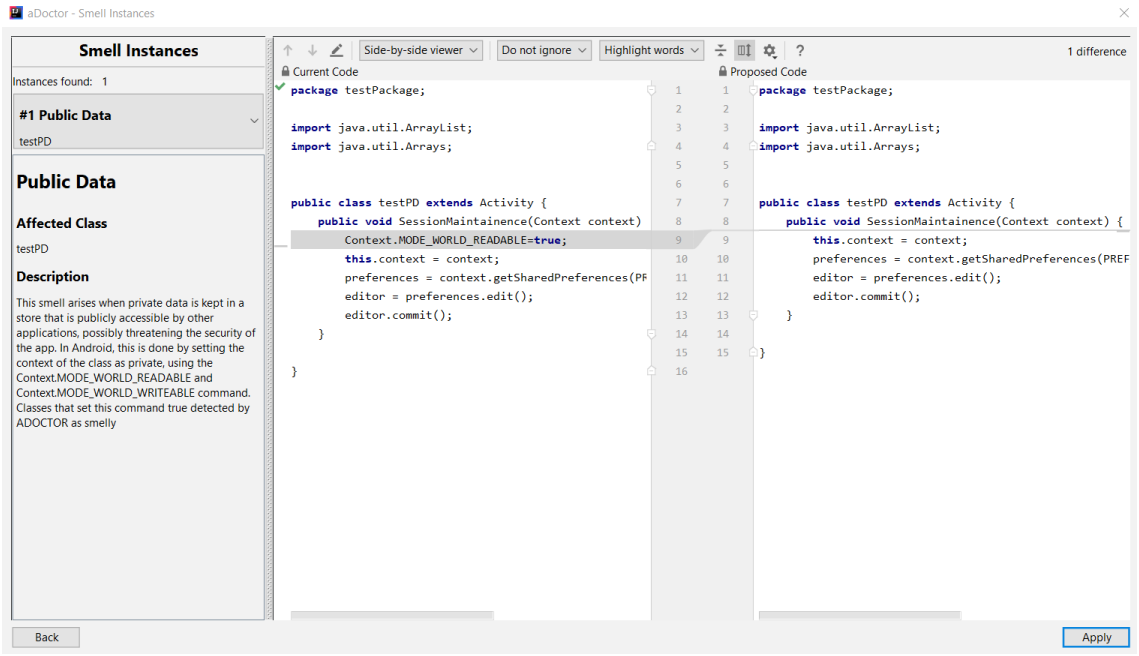

Figura 4.7: Proposta di refactoring

```
public class testPD extends Activity {
   public void SessionMaintainence(Context context) {
      this.context = context;
      preferences = context.getSharedPreferences(PREF_NAME,
          Context.MODE_WORLD_READABLE);
      editor = preferences.edit();
      editor.commit();
   }

}
```

Figura 4.8: Codice dopo il refactoring

Il class diagram per gli analyzer e per i proposer dopo aver introdotto queste nuove classi sarà il seguente, presente, rispettivamente, nelle figura 4.9. e 4.10

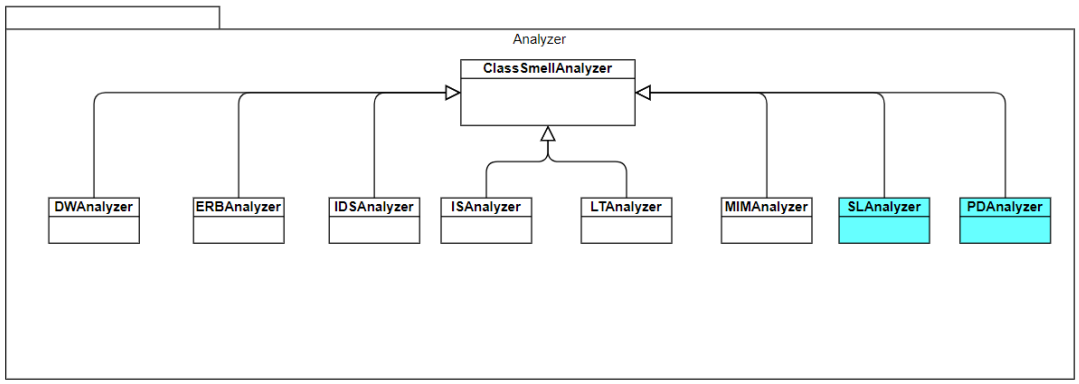

Figura 4.9: Class Diagram Analyzer con nuove classi evidenziate

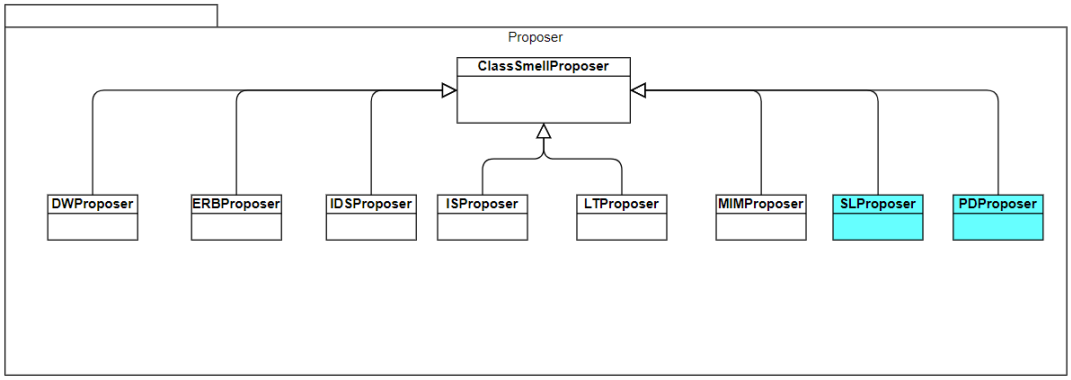

Figura 4.10: Class Diagram Proposer con nuove classi evidenziate

Nella nuova versione di aDoctor, come sopra descritto vengono create quattro nuove classi:

- **SLAnalyzer** nel Package Analyzers
- **PDAnalyzer** nel Package Analyzers
- **SLProposer** nel Package Proposers
- **PDProposer** nel Package Proposers
- **PDSmell** nel package Smell
- **SLSmell** nel package Smell

PDSmell e SLSmell sono due classi, nelle quali vengono descritte le principale informazioni relative agli smell in questione, che compaiono nella finestra di refactoring, quali: Nome completo, Short Name e Descrizione; inoltre vengono inseriti in queste classi tutti i metodi propri degli smell. Inoltre vengono effettuate modifiche al **Package Dialog** facente parte del presentation layer, in particolare:

- Start Dialog : aggiunta di checkbox per la selezione dei due nuovi smell
- Analysis Dialog: Dopo aver selezionato gli smell in StartDialog, aDoctor istanzia gli analyzer e proposer specifici e li memorizza in una sua struttura dati interna. Creazione dei due analyzer specifici per i due nuovi smell
- Smell Dialog: creazione dei due proposer specifici per i due nuovi smell

4.2.2 Uso del Pattern Visitor

aDoctor usa il pattern Visitor per un analisi sul codice. In particolare questo viene implementato nel package visitor appartenente al package ast, nella classe ASTUtilities. In questa versione di aDoctor per garantire una corretta analisi degli smell Public Data e SLow Loop si è proceduto alla creazione di una classe Visitor: ArrayAccessVisitor e creare due nuovi metodi nella classe ASTUtilities. Di seguito viene mostrato lo snippet di codice relativo all'uso del pattern Visitor per i due proposer.

Uso Visitor per Slow Loop: Classe ArrayAccessVisitor

```
public class ArrayAccessVisitor extends ASTVisitor {
   private ArrayList<ArrayAccess> arrayAccesses;
   public ArrayAccessVisitor(ArrayList<ArrayAccess> ArrayAccesses)
   {
      this.arrayAccesses = ArrayAccesses;
   }

   @Override
   public boolean visit(ArrayAccess arrayAccess) {
      arrayAccesses.add(arrayAccess);
      return true;
   }
}
```

Uso Visitor per Slow Loop: metodo getArrayAccesses, classe ASTUtilities

```
public static List<ArrayAccess> getArrayAccesses(ASTNode node) {
      if (node == null) {
         return null;
      } else {
         ArrayList<ArrayAccess> arrayAccesses = new ArrayList<>();
         node.accept(new ArrayAccessVisitor(arrayAccesses));
         return arrayAccesses;
      }
   }
```

Uso Visitor per Public Data: metodo getParentBlock, classe ASTUtilities

```
public static Block getParentBlock(ASTNode node) {
      ASTNode parent = node;
      while (!(parent instanceof Block)) {
         parent = parent.getParent();
      }
      return (Block) parent;
   }
```

CAPITOLO 5

Conclusioni e sviluppi futuri

Nel tool sono diversi gli aspetti migliorabili, anzitutto la precisione della rilevazione degli smell già implementati, poichè in alcune situazioni potrebbero esserci dei falsi positivi o dei falsi negativi. Questo è dovuto al fatto che non esistono metriche di correttezza delle operazioni di identificazione e di refactoring. Vanno implementati altri energy smell, quali: Data Transmission Without Compression, Debuggable Release, Inefficient Data Format and Parser, Inefficient SQL Query, Leaking Inner Class, No Low Memory Resolver, Rigid AlarmManager, Unclosed Closable. L'interazione utente è semplice ed intuitiva, tuttavia è possibile migliorare l'estetica e guidare ancora meglio l'utente nell'uso principale del plugin, fornendogli un tutorial e maggiori spiegazioni a riguardo degli smell.

Bibliografia

[1] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells—5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering, 2018.

[2] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology, 108:115–138, 2019.

[3] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering, 43(11):1063–1088, 2017.

[4] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 4–15, 2016. (

[5] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. In Proceedings of the 4th Workshop on Refactoring Tools, pages 33–36, 2011.

[6] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In 2010 Seventh International Conference on the Quality of Information and Communications Technology, pages 106–115. IEEE, 2010.

[7] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In 2012 16th European Conference on Software Maintenance and Reengineering, pages 411–416. IEEE, 2012.

[8] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In 2009 3rd international symposium on empirical software engineering and measurement, pages 390–400. IEEE, 2009.

[9] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology, 99:1–10, 2018.

[10] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In 2017 IEEE international conference on software maintenance and evolution (ICSME), pages 1–12. IEEE, 2017.

[11] Fabio Palomba and Andy Zaidman. The smell of fear: On the relation between test smells and flaky tests. Empirical Software Engineering, 24(5):2907–2946, 2019.

[12] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3):1188–1221, 2018.

[13] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class changeand fault-proneness. Empirical Software Engineering, 17(3):243–275, 2012.

[14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 101–110. IEEE, 2014.

[15] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In 2012 28th IEEE international conference on software maintenance (ICSM), pages 306–315. IEEE, 2012.

[16] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. How developers perceive smells in source code: A replicated study. Information and Software Technology, 92:223–235, 2017.

[17] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In Proceedings of the 38th International Conference on Software Engineering, pages 225–236, 2016.

[18] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: a software-based tool for estimating the energy profile of android applications. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, ICSE (Companion Volume), pages 3–6. IEEE Computer Society, 2017.

[19] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology, 105:43–55, 2019.

[20] Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999.

[21] Jan Reimann, Martin Brylski, and Uwe Aßmann. A tool-supported quality smell catalogue for android developers. Softwaretechnik-Trends, 34(2), 2014.

[22] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1):20–36, 2009.

[23] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., pages 350–359. IEEE, 2004.

[24] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering, 35(3):347–367, 2009.

[25] Michele Lanza and Radu Marinescu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science Business Media, 2007.

[26] Matthew James Munro. Product metrics for automatic identification of" bad smell" design problems in java source-code. In 11th IEEE International Software Metrics Symposium (METRICS'05), pages 15–15. IEEE, 2005.

[27] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 93–104. IEEE, 2019.

[28] E Alpaydın. Introduction to machine learning. the mit press. 50. 2014.

[29] Jochen Kreimer. Adaptive detection of design flaws. Electronic Notes in Theoretical Computer Science, 141(4):117–136, 2005.

[30] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In 2015 IEEE 26th international symposium on software reliability engineering (ISSRE), pages 261–269. IEEE, 2015.

[31] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In 2009 16th Working Conference on Reverse Engineering, pages 145–154. IEEE, 2009.

[32] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, and Esma Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In 2012 19th Working Conference on Reverse Engineering, pages 466–475. IEEE, 2012.

[33] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 278–281. IEEE, 2012.

[34] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In 2009 Ninth International Conference on Quality Software, pages 305–314. IEEE, 2009.

[35] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. Journal of Systems and Software, 84(4):559–572, 2011.

[36] Salima Hassaine, Foutse Khomh, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Ids: An immune-inspired approach for the detection of software design smells. In 2010 Seventh International Conference on the Quality of Information and Communications Technology, pages 343–348. IEEE, 2010.

[37] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gael Gueheneuc. Numerical signatures of antipatterns: An approach based on b-splines. In 2010 14th European Conference on Software Maintenance and Reengineering, pages 248– 251. IEEE, 2010.

[38] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 87–98. IEEE, 2016.

[39] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Classification model for code clones based on machine learning. Empirical Software Engineering, 20(4):1095–1125, 2015.

[40] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3):1143–1191, 2016.

[41] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. Code smell detection: Towards a machine learning-based approach. In 2013 IEEE International Conference on Software Maintenance, pages 396–399. IEEE, 2013.

[42] Francesca Arcelli Fontana and Marco Zanoni. Code smell severity classification using machine learning techniques. Knowledge-Based Systems, 128:43–58, 2017.

[43] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner), pages 612–621. IEEE, 2018.

[44] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, SANER, pages 103–114. IEEE Computer Society, 2017.

[45] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in android apps. In Aharon Abadi, Danny Dig, and Yael Dubinsky, editors, MOBILESoft, pages 148–149. IEEE Computer Society, 2015.

[46] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of android smells. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, SANER, pages 115–126. IEEE Computer Society, 2017.

[47] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Lightweight detection of android-specific code smells: The adoctor project. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, SANER, pages 487–491. IEEE Computer Society, 2017.

[48] Emanuele Iannone, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. Refactoring android-specific energy smells: A plugin for android studio. 2018.

[49] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, 2002, ISBN 88-7192-150-X.

Ringraziamenti

INSERIRE RINGRAZIAMENTI QUI