Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Estensione di un tool esistente per il riconoscimento di nuovi architectural smell

Relatore: Prof.ssa Francesca Arcelli Fontana

Co-relatore: Dott.ssa Ilaria Pigazzini

Relazione della prova finale di:
Davide Rendina
Matricola 830730

Anno Accademico 2019-2020

Indice

1 Introduzione

Per decenni progettisti e sviluppatori hanno realizzato sistemi prestando maggiore attenzione ai requisiti tecnici piuttosto che all'architettura software, ma con il trascorrere degli anni la tendenza è cambiata a tal punto che oggi viene considerata un elemento fondamentale nel processo di progettazione e sviluppo di un sistema.

Con il termine architettura si indica la struttura elaborata dai progettisti per il sistema, che include la divisione del sistema in componenti differenti, le relazioni che intercorrono tra essi e la loro disposizione e proprietà. La struttura di un sistema supporta il suo intero ciclo di vita e la sua qualità impatta in modo significativo su diverse proprietà quali facilità di comprensione e di sviluppo, semplicità della manutenzione, integrazione di nuove funzioni, implementazione di politiche di sicurezza e facilità nella definizione di test. Un sistema che presenta un'architettura progettata male è soggetto a diversi problemi, che comportano un grande spreco di risorse da parte del team di sviluppo, presenza di numerosi errori e ad un lento degrado della qualità del software [11].

Una categoria di questi problemi è rappresentata dagli *architecture smells*, violazioni di *design principles* e soluzioni che impattano negativamente sulla qualità del software e sulle risorse utilizzate per la sua manutenzione ed evoluzione [3] [15]. La presenza di *smell* è dannosa per il progetto, in quanto influenza negativamente diverse caratteristiche qualitative dell'architettura e introduce *technical debt* [31], il debito accumulato durante lo sviluppo e manutenzione di un software quando vengono prese decisioni riguardanti il design errate oppure non ottimali. Se queste situazioni introdotte vengono corrette tempestivamente il debito non subisce variazioni, altrimenti continua ad incrementare e con esso anche le difficoltà nella modifica e manutenzione dei componenti sistema. La ricerca di *smell* e la loro rimozione attraverso strategie collaudate è quindi un'attività fondamentale per la manutenzione di un progetto, poiché permette la diminuzione del *technical debt* e il mantenimento di un'elevata qualità del software, attraverso un miglioramento di diverse qualità chiave quali comprensione del design, facilità nell'estensione, riusabilità e affidabilità.

Suryanarayana [31], ha proposto una classificazione degli *architecture smell* in quattro categorie differenti (*abstraction*, *encapsulation*, *modularization* e *hierarchy*), derivate dalla violazione dei principi fondamentali del *software design* introdotti da G. Booch nel suo *object model* [7]. Le tipologie rilevanti per questo elaborato sono *hierarchy smell*, in particolare lo *smell Subclasses Do Not Redefine Methods*, e *abstraction smell*, alla quale appartengono *Unutilizied Abstraction* e *Unnecessary Abstraction*. La presenza di *Subclasses Do Not Redefine Methods* ha un impatto fortemente negativo sul codice in quanto

incide sulla facilità di modifica ed estensione del codice e sul riutilizzo delle entità, dal momento che non rende possibile la sostituzione del supertipo con i suoi sottotipi senza alterare l'esecuzione del sistema. Gli *smell* relativi al principio di astrazione introducono invece nel design responsabilità non uniche e poco significative, che influenzano negativamente la facilità di comprensione del sistema e la sua affidabilità.

I tre *smell* introdotti sono stati analizzati al fine permettere la loro *detection* attraverso Arcan [11] [51], un *tool* sviluppato per effettuare analisi statiche di sistemi software che basa il suo funzionamento sui concetti di *graph database technology* e *graph computing*. Il *tool* genera un grafo delle dipendenze per la rappresentazione del sistema analizzato, che viene successivamente esaminato tramite algoritmi di *detection* per individuare gli *smell* all'interno del progetto. Al fine di implementare il riconoscimento degli smell introdotti in precedenza da parte di Arcan, è stato necessaria l'implementazione dei tre nuovi algoritmi di *detection*. Questi algoritmi hanno inoltre richiesto l'aggiunta di nuovi elementi al grafo delle dipendenze e modifiche agli algoritmi di *parsing*.

Questo lavoro è stato organizzato come segue. Nel capitolo 2 verranno discussi alcuni lavori correlati al tema del riconoscimento di *smell*, con la presentazione anche di diversi *tool* in grado di effettuare la loro *detection*. Il capitolo 3 descrive nel dettaglio il *tool* Arcan [11] e le modifiche effettuate al *parsing* e alla struttura del grafo delle dipendenze necessarie per il riconoscimento dei nuovi *smell*. Nel capitolo 4 viene effettuata la presentazione dettagliata degli *smell* introdotti in questo elaborato e delle strategie ideate per il loro riconoscimento. Il capitolo 5 riguarda l'esecuzione degli algoritmi su progetti reali, la relativa analisi dei risultati ottenuti e i problemi riscontrati. Sarà presente inoltre un confronto fra Arcan e il *tool* Designite [28]. L'ultimo capitolo conclude il lavoro svolto e suggerisce alcuni sviluppi futuri per il progetto.

2 Lavori correlati

Questo capitolo illustra diversi lavori, appartenenti alla letteratura degli *smell*, che riguardano la loro *detection*, le diverse tecniche di *refactoring* e l' influenza sulla qualità del progetto. La prima sezione presenta nel dettaglio i lavori riguardanti il tema degli *smell*, mentre saranno catalogati nella successiva numerosi *tool* in grado di effettuare il riconoscimento di *architecture smell*, con enfasi sulle loro capacità di *detection*, sulle strutture dati utilizzate e strategie di ricerca adottate.

2.1 Bad smell e refactoring

La letteratura degli *smell* contiene diversi lavori riguardanti la loro definizione, l'impatto che hanno sulla qualità del sistema e le tecniche per il loro riconoscimento. La maggioranza di questi lavori però riguarda la categoria dei *code smell* [12], in quanto meno lavoro è stato svolto nell'ambito degli *architectural smell* [3].

Martin Fowler et al. [12] hanno per primi introdotto i concetti di *code smell* e *refactoring*, cioè problemi comuni all'interno del codice e le relative tecniche di rimozione. Gli autori hanno voluto creare una guida per effettuare il *refactoring* in diverse situazioni, in modo da evitare l'introduzione di *bug* e mantenere un'elevata qualità del codice.

Diversi aspetti riguardanti le tecniche di *refactoring* e le sue differenti applicazioni sono stati anche approfonditi M. Lippert et al. [18]. I due autori trattano anche il tema degli *architecture smell*, presentando un loro catalogo e diverse strategie per effettuare il *refactoring* in maniera ottimale. Oltre al tema degli *architecture smell*, vengono discusse anche tecniche applicate in altri ambiti come *API*, *database* e progetti complessi (*Large Refactoring*).

Il tema del *refactoring* è trattato anche da M. Stal [29], che affronta il tema della prevenzione dell'erosione architetturale attraverso l'applicazione di differenti tecniche per la rimozione di *smell*.

La definizione del concetto di *architecture smell* è stata effettuata da J. Garcia et al. [15], stabilendo gli aspetti che caratterizzano questa tipologia di problemi architetturali e le differenze tra essi e gli anti-pattern architetturali. Il loro lavoro inoltre contiene una descrizione dettagliata di quattro diversi *architectural smell*, contenenti anche esempi generici di diagrammi *UML* al fine di favorire il lavoro dei progettisti e sviluppatori nella ricerca degli stessi.

G. Suryanarayana et al. [31] hanno proposto un catalogo di *architecture smell* suddivisi in 4 categorie differenti in base al principio di progettazione [7] violato dallo *smell*: *Abstraction*, *Encapsulation*, *Hierarchy* e *Modularization*. Oltre alla definizione, ogni *smell* presenta inoltre informazioni riguardanti strategie di *refactoring* consigliate, impatto sulla qualità del codice, cause

potenziali e diversi esempi.

F. Arcelli Fontana et al. [10] hanno studiato le relazioni tra diverse tipologie di *code smell* all'interno di 74 sistemi, al fine di valutare il loro impatto sul *technical debt*. Gli autori hanno riscontrato una percentuale significante di casi di correlazione tra istanze di *smell* differenti e ciò ha permesso di confermare che i *code smell* tendono a presentarsi in gruppo, influenzando negativamente la manutenzione dei progetti e il debito tecnico in maniera superiore rispetto alla loro manifestazione singola.

Una ricerca empirica riguardante l'influenza dei *code smell* sul degrado architetturale di un sistema è stata svolta anche da I. Macia et al. [19], attraverso lo studio di 40 versioni di 6 sistemi software e l'analisi di 2056 anomalie del codice. Questa ricerca ha evidenziato che il 78% dei problemi architetturali presenti nei programmi possono essere ricondotti ad anomalie del codice e che le strategie di *refactoring* non sempre contribuiscono in maniera significativa alla rimozione di problemi collegati all'architettura.

Le possibili correlazioni e dipendenze tra *code smell* e *architectural smell* sono stati approfondite da F. Arcelli et al. [2]. Attraverso l'analisi delle correlazioni tra 19 *code smell* e 4 *architectural smell*, gli autori sono stati in grado di dimostrare che le due categorie considerate possono considerarsi tra loro indipendenti e che quindi è necessario prestare attenzione al *refactoring* di tutte le categorie di *smell*.

M. Tufano et al. [32] hanno effettuato una ricerca riguardante il momento e le motivazioni dell'introduzione di *code smell* nel progetto, attraverso l'analisi di diverse *commit* effettuate su 200 progetti *open source* provenienti da differenti ecosistemi. La ricerca effettuata ha permesso agli autori la definizione di quattro differenti situazioni comuni riguardanti l'introduzione degli smell, con relativi consigli e rimedi per evitare questi scenari.

Duc Minh Le et al. [17] sono stati in grado di analizzare l'impatto degli *architectural smell* sul sistema, attraverso lo studio delle relazioni tra gli *smell* trovati e i problemi indicati dagli *issue trackers* di diversi progetti. Lo studio ha dimostrato il forte impatto che la loro presenza ha sul decadimento del software, poiché rende necessaria una quantità di risorse considerevole per garantire il mantenimento del software durante il suo ciclo di vita. Inoltre i *file* nel progetto coinvolti negli *smell* risultano maggiormente inclini agli errori e alle continue modifiche rispetto a quelli che non presentano problemi.

2.2 Tool per il riconoscimento di architectural smell

Sul mercato sono disponibili diversi *tool* per effettuare analisi statica di sistemi software, in grado di analizzare differenti linguaggi di programmazione, effet-

tuare il calcolo di numerose metriche e ricercare *architecture smell* a diverse granularità.

U. Azadi et al. [3] hanno presentato un catalogo di nove *tool* disponibili e non sul mercato, volti alla *detection* di *architectural smell*. Gli autori hanno posto molta enfasi sul confronto operativo dei vari *tool*, analizzando in particolare le differenze tra le diverse regole di *detection* e i differenti risultati ottenuti. Verranno proposti di seguito diversi *tool* per la *detection* di *architectural smell*, mettendo in risalto le loro capacità e la peculiarità del funzionamento e della ricerca di ognuno di essi, la maggior parte dei quali presenti nel lavoro di Azadi [3].

Designite [28] è un *tool* che permette la valutazione della qualità del software attraverso l'identificazione di un vasto range di *smell* (ovvero *code*, *architectural* oppure *design smell*), il calcolo di diverse metriche e l'identificazione del codice replicato nel progetto. Designite effettua il *parsing* del codice in modo da generare un *Abstract Syntax Tree (AST)*, utilizzato poi per la creazione del meta-modello necessario al funzionamento del *tool*. L'analisi di questo meta-modello permette poi di effettuare la *detection* e calcolare le metriche.

AI Reviewer [56] è un *tool* in grado di analizzare progetti C++ al fine di trovare, anche attraverso la ricerca di *architectural smell*, violazioni dei principi *S.O.L.I.D.* introdotti da Robert Martin [23] e calcolare diverse metriche a granularità differenti. Il funzionamento è basato sulla rappresentazione astratta e dettagliata del progetto derivata dal *parsing* del codice sorgente attraverso un modello, analizzato da un ulteriore componente per svolgere i differenti calcoli.

Massey Architecture Explorer (MAE) [9] è un applicazione *browser-based* per visualizzare e analizzare architetture di progetti Java. MAE estrae un modello *graph-based* dal *bytecode* Java, analizzato poi da un algoritmo apposito per la *detection* di *architecture smell* e anti-pattern architetturali.

ARCADE [16] è un software in grado di monitorare e analizzare le modifiche e il decadimento architetturale di un progetto attraverso le sue differenti versioni. ARCADE può recuperare l'architettura dal codice sorgente del progetto, per poi utilizzare le informazioni estratte per il calcolo di metriche (confrontando l'architettura estratta con quella di tutte le versioni precedenti e successive di quel software), per effettuare analisi statistiche sui dati ottenuti e ricercare diversi *architectural smell*. [3].

STatic ANalizer for Java [50] è un software per effettuare analisi strutturali di progetti Java analizzando il *bytecode* del progetto. Con STAN è possibile visualizzare le dipendenze tra classi e package del sistema, calcolare metriche per la qualità del software con anche *rating* e indicazioni sui valori ottenuti ed effettuare la detection dello smell *Cyclic Dependency*.

Sonargraph [54][33] è un software che permette il monitoraggio e l'analisi di

architettura software e metriche di un progetto, focalizzato sulla riduzione del *technical debt*. Tramite un *Domain Specific Language* (DSL) specializzato, gli sviluppatori possono definire le regole per la descrizione della loro architettura, controllate e validate in maniera automatica durante il processo di sviluppo software. Supporta diversi linguaggi di programmazione (Java, C, C, C++, Python) ed è possibile inoltre la detection di due smell architetturali [3].

Hotspot Detector [24] è un *tool* per la detection automatica di cinque problemi architetturali, derivati dalla teoria di progettazione di Baldwin e Clark [4] e ricercati attraverso la combinazione di informazioni storiche e architetturali. Gli stessi sviluppatori del *tool* hanno ribattezzato questi problemi come *Hotspot Patterns*.

Structure101 [53] è un *tool* che permette la visualizzazione dell'architettura del progetto in maniera modulare, gerarchica e organizzata e la *detection* di due differenti *architecture smell*, con simulazione e applicazione delle tecniche di *refactoring*. È inoltre possibile la definizione regole di dipendenza, stratificazione e visibilità attraverso elementi strutturali.

3 Estensione di Arcan per la detection dei nuovi smells

In questo capitolo viene effettuata una presentazione del *tool* Arcan e delle modifiche necessarie per il riconoscimento dei tre nuovi *architectural smells*. Oltre a una breve introduzione, vengono approfonditi diversi aspetti come la sua architettura e la struttura dati del grafo delle dipendenze. L'ultima sezione introduce le modifiche effettuate al *tool* per rappresentare nel grafo le diverse strutture necessarie per il riconoscimento dei nuovi *smell*.

3.1 Introduzione ad Arcan

Arcan (ARChitecture ANalizer) [11][6] è uno strumento per l'analisi statica di sistemi software, sviluppato dal laboratorio Essere - Università degli Studi Milano Bicocca [51]. Questo *tool* è in grado di effettuare analisi statiche di programmi scritti in diversi linguaggi (Java, C, C++ e più recentemente Python [30]) al fine di calcolare differenti metriche ed effettuare la *detection* di dieci tipologie di *architectural smell*. Non tenendo in considerazione i tre *smell* introdotti in questo elaborato, Arcan è in grado di riconoscere i seguenti *AS*:

- *Cyclic Dependency* si riferisce a sottosistemi che sono coinvolti in catene di relazioni che non rispettano la natura aciclica della struttura delle dipendenze di un sottosistema, violando così il *Acyclic Dependency Principle* [22] [11].
- *Hub-Like Dependency* si verifica quando una *abstraction* ha un alto numero di dipendenze in ingresso e in uscita, che non le permettono di mantenere accoppiamento basso e coesione alta [11].
- *Unstable Dependency* riguarda sottosistemi (componenti) che dipendono da altri sottosistemi meno stabili di loro. Cambiamenti a sottosistemi che presentano dipendenze instabili possono causare modifiche a catena nel sistema [11].
- *God Component* si manifesta quando un componente è eccessivamente largo in termini di *LOC (lines of code)* oppure numero di classi [18].
- *Insufficient Package Cohesion* descrive una situazione nella quale un entità architetturale presenta una coesione interna bassa.
- *Feature Concentration* insorge quando un'entità architetturale implementa diverse funzionalità al suo interno [1].

- *Scattered Funtionality* si presenta in un sistema dove diversi componenti sono responsabili della realizzazione delle stesse responsabilità di alto livello. [15].

3.2 Architettura di Arcan

La struttura di Arcan è composta da quattro elementi principali [11]:

1. *System Reconstructor* in grado di effettuare il *parsing* del codice sorgente del progetto fornito come input e di estrarre da esso tutte le informazioni riguardanti le dipendenze tra i vari file, grazie all'utilizzo della libreria Spoon [25]. Le informazioni estratte sono poi utilizzate per la generazione di una *Abstract Syntax Tree (AST) map*. Questo componente non è in grado di recuperare eventuali dipendenze esterne al sistema e perciò Arcan è in grado di effettuare le analisi considerando solamente gli elementi passati come input.

2. *Graph Manager* è il componente responsabile della creazione del grafo delle dipendenze partendo dall'analisi della *AST map* ricevuta dal *System Reconstructor*. Dopo l'inizializzazione del grafo, inserisce in esso tutti i nodi e gli archi seguendo le dipendenze presenti nella mappa, contenenti le diverse informazioni riguardo i componenti del sistema sotto analisi. Questo componente utilizza la libreria Apache Tinkerpop [48] per svolgere le sue mansioni.

3. *Metrics Engine* si occupa della computazione delle metriche introdotte da R. Martin [21] necessarie per la detection degli *architectural smells*. I risultati delle metriche calcolate vengono salvate poi all'interno dei componenti del grafo ai quali la metrica si riferisce.

4. *Architectural Smell Engine* contiene tutti gli algoritmi per la ricerca degli *architectural smells* nel grafo e per il filtraggio dei falsi positivi. Per ogni istanza trovata aggiunge un nuovo nodo di tipo *smell* al grafo, che viene poi messo in relazione con i diversi nodi rappresentanti i componenti coinvolti in quella particolare istanza.

3.3 Grafo delle dipendenze

Il grafo delle dipendenze è l'elemento fondamentale su cui è basato tutto il funzionamento di Arcan. Si tratta di un *graph database* che salva tutte le informazioni riguardanti il progetto analizzato attraverso gli elementi dei grafi (archi, nodi e anche le loro proprietà) e permette di svolgere *graph computing* per effettuare il calcolo di metriche e il riconoscimento di *AS*.

Questo grafo si presenta come un grafo diretto, che mette in evidenza le dipendenze tra i vari componenti del progetto rappresentati dai nodi del grafo. Ogni nodo può rappresentare un elemento del linguaggio (package, classi, interfacce, funzioni e attributi) ed è in relazione con gli altri nodi tramite uno o più archi di diverse tipologie, che rappresentano al meglio la classificazione delle relazioni tra i componenti. Nodi e archi possono presentare diverse proprietà utilizzate per il salvataggio di informazioni e metriche riguardanti un singolo componente.

Il grafo delle dipendenze dispone di numerose tipologie di nodi e archi, ma di seguito verranno presentate solamente gli elementi necessari per la comprensione del lavoro svolto. Le strutture riguardanti *Attribute* e *Function*, introdotte in questo elaborato e solamente accennate in questo elenco, hanno un approfondimento loro dedicato nelle sottosezioni 3.3.1 e 3.3.2. Riguardo i nodi porremo la nostra attenzione solamente su quattro diverse tipologie:

- *Unit*, classi concrete e astratte, interfacce ed enumerazioni. Queste tipologie di nodi sono descritti dagli attributi *name*, *filePath* e *componentType*, che forniscono informazioni riguardanti rispettivamente il suo nome qualificato (compreso di *namespace*), il percorso assoluto del file che contiene la *unit* e la tipologia rappresentata. La proprietà *componentType* può assumere i valori *class*, *abstract_ class*, *interface* oppure *enum*.

- *Smell*, tipologia di *smell* identificata in una determinata *unit*. Ogni diverso *smell* definisce il suo particolare nodo, che contiene le diverse informazioni che lo caratterizzano.

- *Function*, singola funzione definita da una *unit*.

- *Attribute*, attributo contenuto in una *unit*.

Tra le diverse tipologie di archi disponibili, quelle importanti per lo sviluppo della detection e la comprensione degli algoritmi sono:

1. *dependsOn* collega due nodi di tipo *unit*, e indica che il nodo con questo arco in uscita ha una dipendenza verso l'altra *unit*. Un esempio di dependsOn potrebbe essere una classe che al suo interno richiama i metodi di un'altra classe.

2. *isChildOf* è indice di una relazione di tipo gerarchico tra due diverse *unit*, con il supertipo che presenta questo arco in entrata.

3. *isImplementationOf* raffigura l'implementazione da parte di una *unit* di un'interfaccia. Nel dettaglio la *unit* che ha l'arco isImplementationOf in uscita implementa l'interfaccia rappresentata dall'altra *unit*.

4. *containedIn* definisce una relazione tra una *unit* e un attributo che essa definisce.

5. *implementedBy* che rappresenta l' implementazione di una funzione da parte di una *unit* specifica.

6. *affects* indica che l'istanza di uno *smell*, che possiede questo arco in uscita, è stata trovata in una determinata *unit*.

7. *archi di tipologie particolari*, definite dagli *smell* per rappresentare diverse situazioni particolari. Vengono utilizzati in combinazione con l'arco *affects*.

3.4 Modifiche effettuate al grafo

Al fine di implementare la *detection* degli *smell* introdotti, è stata necessaria la modifica degli algoritmi di *parsing* esistenti e l'aggiunta di nuove strutture al grafo delle dipendenze. Nello specifico sono state effettuate modifiche al *parser* Java per il recupero di informazioni riguardanti funzioni e attributi e introdotte nel grafo le strutture necessarie per la loro rappresentazione.

3.4.1 Parsing e rappresentazione di funzioni

Il ruolo delle funzioni nelle strategie di identificazione è fondamentale per tutti e tre gli algoritmi presentati.

Arcan disponeva già della struttura necessaria per il *parsing* delle funzioni, perciò è stato necessario solamente effettuare l'implementazione degli algoritmi per il recupero delle informazioni utilizzando le API fornite dalla libreria Spoon [25].

Le modifiche al grafo per la rappresentazione delle funzioni hanno comportato l'introduzione di due nuovi tipi di componenti, un nodo *function* e un arco implementedBy. Il nodo *function* rappresenta la singola funzione definita da una classe o interfaccia, dove il valore della proprietà *name* del nodo corrisponde al nome della funzione rappresentata dallo stesso. Per rappresentare la definizione di una funzione da parte di una *unit* è stato introdotto un arco di tipo *implementedBy*, in uscita dalla funzione verso la *unit* che la contiene. Un esempio di questa struttura si può identificare nella figura 1 dove la funzione è rappresentata dal nodo verde e la *unit* che la implementa da quello di colore beige.

Durante la modifica del *parsing* è stato necessario effettuare due scelte principali: la rappresentazione delle funzioni non per *signature* ma per nome e la tecnica di recupero dei metodi definiti.

La scelta di rappresentare le funzioni attraverso il nome è stata effettuata al fine di favorire la detection dello *smell Subclasses Do Not Redefine Method* (sezione 4.1), dove viene controllato che il sottotipo ridefinisca almeno un metodo del suo supertipo. In particolare si desiderava l'inclusione anche di tutti i casi di *overloading*, dove un figlio non ridefinisce il comportamento di un metodo che il padre implementa ma aggiunge un nuovo comportamento per un metodo già esistente, rappresentato da supertipo e sottotipo che condividono un metodo con lo stesso nome ma con *signature* differente.

La libreria Spoon [25] utilizzata per il *parsing* presenta inoltre due differenti tipologie di recupero delle funzioni da una determinata classe e/o interfaccia del sistema:

1. Recupero dei metodi definiti direttamente dal componente.

2. Recupero delle funzioni che possono essere chiamati su un istanza di quel componente, inclusi quindi anche tutti i metodi derivati dalle superclassi.

È stato preferito il primo approccio poiché il secondo presentava diverse criticità. Nel linguaggio Java tutte le classi derivano dalla classe *Object* e quindi le funzioni di quest'ultima sarebbero risultate implementate da tutte le *unit*, con conseguenza che nessuna di esse sarebbe risultata come *smell Subclasses Do Not Redefine Methods* (sezione 4.1). Inoltre questa soluzione avrebbe portato a un eccessivo appesantimento del grafo e di Arcan in generale, poiché oltre ai metodi di *Object* aggiunti a tutte le classi ogni funzione sarebbe stata replicata per tutte le *unit* che la ereditano.

3.4.2 Parsing e rappresentazione di attributi

Arcan non disponeva di nessuna struttura o algoritmo dedicata al *parsing* degli attributi, perciò tutte le classi e algoritmi sono stati implementati da zero, considerando comunque la struttura degli altri elementi simili per garantire continuità nel codice. La presenza nel grafo di informazioni riguardanti gli attributi definiti dalle *unit* è stata necessaria per il riconoscimento dello *smell Unnecessary Abstraction*. Questa esigenza ha portato all'introduzione nel grafo di due nuovi elementi: un nodo di tipo *attribute* e un arco *containedIn*. Il nodo *attribute* rappresenta un attributo definito da una particolare classe, collegato con un arco in ingresso verso la *unit* che lo definisce. Un esempio della struttura è presentato dalla figura 1, dove il nodo di colore beige rappresenta la *unit* mentre quello rosa l'attributo.

Un attributo può essere inoltre identificato univocamente all'interno del grafo grazie alla proprietà *name*, formata dalla concatenazione del nome della classe che lo definisce e quello dell'attributo stesso. Questi nodi presentano inoltre tre proprietà aggiuntive rispetto agli altri:

- *Attribute type*, tipo dell'attributo rappresentato dal nodo (può essere sia un tipo primitivo che il nome qualificato di una classe)
- *Constant attribute*, *flag* che indica se è un valore costante o meno
- *Default value*, *flag* che specifica la presenza o meno di un valore assegnato

3.4.3 Modifiche alla classe Unit

Un ulteriore elemento che ha subito modifiche è la classe Unit, rappresentante di un nodo *unit* del grafo, che ha visto l'aggiunta di diverse procedure per il recupero di metodi e attributi definiti. È stato introdotta la funzione *getAllMethods*, che produce in output tutti i metodi concreti o astratti definiti nella gerarchia di una *unit*, al fine di considerare per lo *smell Subclasses Do Not Redefine Methods* anche tutte le funzioni ereditate. In Java inoltre le classi astratte che implementano un'interfaccia non sono obbligate alla definizione di tutti i suoi metodi e la loro implementazione viene lasciata alle classi concrete che estendono quella astratta. È stato deciso quindi di considerare queste funzioni come definite comunque dalle classi astratte, come fossero metodi *abstract*, e vengono quindi recuperate dalla procedura *getAllMethods*.

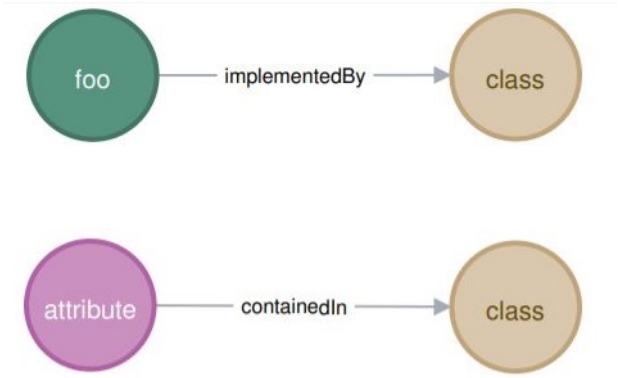

Figure 1: Esempio di strutture per attributi e funzioni

4 Riconoscimento di nuovi architectural smell con Arcan

Obiettivo di questo capitolo è l'introduzione e approfondimento dei tre nuovi *architectural smell (AS)* introdotti. Le sezioni successive sono orientate alla descrizione dei tre nuovi *smell*, rappresentati secondo una struttura ben definita:

1. Introduzione con descrizione delle cause più comuni che portano al suo inserimento nel sistema e piccola descrizione del nodo introdotto nel grafo
2. Impatto sulla qualità del codice e possibili strategie di *refactoring*
3. Strategie di identificazione ideate
4. Algoritmo di *detection* in pseudocodice

Per favorire la comprensione degli *smell* è necessaria l'introduzione del concetto di *abstraction*, riferito a classi (concrete o astratte) e interfacce che fanno parte del progetto. Questo termine deriva dal *Abstraction Principle* [7], che sostiene la semplificazione delle entità attraverso l'eliminazione di dettagli non necessari e la generalizzazione delle caratteristiche condivise tra esse. Un altro principio importante per la comprensione del lavoro svolto è il *Hierarchy Principle* [7], che supporta la creazione di un'organizzazione gerarchica di *abstraction* attraverso l'utilizzo di tecniche come classificazione, generalizzazione e sostituibilità.

Obiettivo dell'estensione di Arcan è stato fornire al *tool* la capacità di riconoscere tre nuovi *architectural smells*: *Subclasses Do Not Redefine Methods*, *Unutilizied Abstraction*, e *Unnecessary Abstraction*. Tutti gli *smell* introdotti violano i principi presentati in precedenza. In dettaglio, lo smell *Subclasses Do Not Redefine Methods* è responsabile della violazione del *Hierarchy Principle* mentre *Unutilizied Abstraction* e *Unnecessary Abstraction* non rispettano il *Abstraction Principle*.

4.1 Subclasses Do Not Redefine Methods

Subclasses Do Not Redefine Methods (*SR*), definito da M. Lippert et al. [18], afferma che se le sottoclassi non ridefiniscono i metodi delle loro superclassi può essere indice del fatto che attraverso l'ereditarietà non è espressa nessuna astrazione ma è più che altro ereditarietà implementativa [18].

Questo può essere indice del fatto che attraverso la gerarchia non è espressa nessuna astrazione e che non c'è quindi alcun motivo per cui questa gerarchia debba essere presente nel design. La presenza di questo *smell* può essere

causata principalmente dalla tendenza all'utilizzo delle gerarchie per il riutilizzo delle funzionalità del padre, senza però che la classe e il suo supertipo condividano una relazione IS-A.

Nodo di tipo smell nel grafo Il nodo dello *smell* SR dispone di due o più archi in uscita verso altrettante *unit*:

- Presenta la dipendenza *affects* verso la *unit* colpita dallo *smell*.
- È in relazione con almeno una *unit* attraverso un arco di tipo *fatherInvolved*, che indica i supertipi della classe che presenta lo *smell*.

4.1.1 Impatto sulla qualità del codice e refactoring

La presenza di *Subclasses Do Not Redefine Methods* può influenzare in maniera negativa la qualità del codice attraverso la compromissione di proprietà quali comprensibilità, riusabilità, facilità nella modifica, estensione e affidabilità [31].

Analizzando nel dettaglio le qualità influenzate negativamente dalla presenza di questo *smell*, si può affermare che:

- Quando le classi in una relazione gerarchica non condividono una relazione concettuale IS-A, può portare a molta confusione e ridurre la *comprensibilità* del sistema, confondendo gli sviluppatori e i progettisti.
- Se le *unit* non condividono una relazione IS-A il riutilizzo del codice potrebbe essere compromesso poiché non è possibile la sostituzione delle istanze dei sottotipi con i loro supertipi. Questa situazione renderebbe difficile il *riutilizzo* dell'intera gerarchia in un altro contesto. Questo potrebbe causare diversi problemi ulteriori relativi alla difficoltà nella *modifica ed estensione* della gerarchia, poiché un cambiamento potrebbe avere un grosso impatto sul codice.
- La presenza di classi che non condividono una relazione IS-A potrebbe portare a diversi problemi di *affidabilità* del codice, poiché lo scambio tra un'istanza di una superclasse con quella di una sottoclasse potrebbe causare errori indesiderati.

Per il *refactoring* di questo *smell* l'applicazione di *Replace Inheritance With Delegation* è la scelta consigliata e più diffusa [31]. Questa strategia consiste nella trasformazione della relazione IS-A tra le due classi in una relazione di utilizzo, in modo che la ex-sottoclasse abbia al suo interno un oggetto dell'altra per l'utilizzo dei suoi metodi.

4.1.2 Strategia di identificazione

Devo controllare che in ogni relazione gerarchica presente nel programma analizzato non si verifichi la ridefinizione di almeno un metodo del supertipo da parte del sottotipo. Per fare questo, è necessario controllare che l'intersezione dei metodi definiti dalle due classi sia vuota e quindi non abbiano almeno un metodo in comune. Nella ricerca dei metodi definiti dalla superclasse viene utilizzata la funzione *getAllMethods* definita in precedenza poiché, se i metodi delle interfacce non fossero considerati, una situazione analoga a quella definita dalla figura 2, dove il sottotipo (nodo beige) ridefinisce un metodo (nodo verde) definito solamente dall'interfaccia (nodo azzurro) ma non implementato dalla classe astratta (nodo rosso), verrebbe considerata come *smell*. È stato deciso però di non ritenere questa situazione come tale poiché le due classi condividono una relazione IS-A e il metodo dell'interfaccia che la classe concreta implementa è derivato comunque dal suo supertipo.

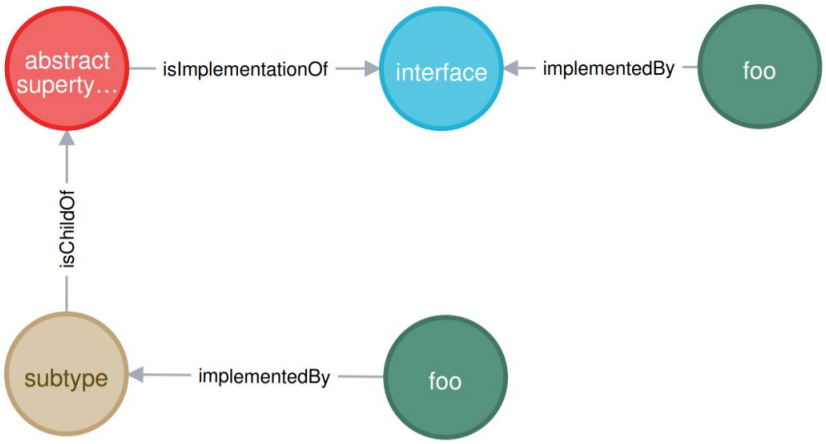

Figure 2: Esempio interfaccia implementata da una classe astratta

Dalle relazione *isChildOf* considerate vengono escluse tutte quelle che coinvolgono:

- Due *unit interface*, poichè nelle gerarchie tra interfacce non avviene la ridefinizione di metodi ma si verifica solamente l'estensione del comportamento attraverso nuove funzioni, pertanto tutti questi casi risulterebbero falsi positivi.
- Una *unit* di tipo classe che rappresenta una *exception* (oppure anche *error*), poiché è molto comune che abbiano come supertipo una *exception* e richiamino solamente il costruttore utilizzando parametri diversi. In

questi casi non c'è ridefinizione dei metodi ma non è stato giudicato comunque come *smell*, in quanto la relazione IS-A è presente.

In termini di grafo è necessario considerare tutti i nodi che presentano un arco *isChildOf* in uscita dove almeno un nodo di tipo *function* in ingresso al nodo considerato non condivida la proprietà *name* con un'altro nodo *function*:

- Definito da uno dei supertipi della classe analizzata, dove per supertipi si intendono i nodi che hanno in ingresso gli archi *isChildOf* provenienti dalla classe sotto analisi.
- Presente nella gerarchia di uno dei supertipi. Per verificare questo si seguono tutti gli archi *isChildOf* e *implementedBy* a partire dai supertipi al fine di controllare anche tutti i metodi ereditati.

4.1.3 Algoritmo

Input un sottografo del grafo principale, considerando solamente i nodi che rappresentano classi (concrete e astratte) coinvolte in una dipendenza del tipo *isChildOf* e le relative funzioni definite dalle classi. Oltre all'arco *isChildOf* sarà considerato anche *implementedBy*.

Output una lista di classi che presentano lo *smell* e i relativi supertipi coinvolti.

Algoritmo

function SUBCLASSES-DOES-NOT-REDEFINE-METHODS-DETECTOR( )
Prendo tutti i nodi che hanno almeno un arco in entrata del tipo isChildOf, escludendo le interfacce e le enumerazioni
for Ogni vertice trovato, che rappresenta il supertipo **do**
Prendo la lista delle funzioni definite ed ereditate dalla unit, includendo anche quelli definiti dalle interfacce implementate
if La lista dei metodi considerata non è vuota **then**
for Ogni sottotipo con isChildOf in uscita verso il supertipo **do**
Prendo la lista di tutte le funzioni definite dal sottotipo
if L'intersezione tra le liste dei metodi è l'insieme vuoto **then**
Aggiungo il vertice alla lista degli smell

4.2 Unutilizied Abstraction

Unutilizied Abstraction (*UUA*), definito da G. Suryanarayana et al. [31], si manifesta quando una *abstraction* viene lasciata inutilizzata, cioè non diret-

tamente usata o non raggiungibile. Questo *smell* si può manifestare in due forme:

- *Unreferenced Abstraction*, classi concrete che non sono utilizzate da nessuno.
- *Orphan Abstraction*, classi astratte e interfacce che non hanno nessuna abstraction derivata.

Il principio di astrazione afferma che le *abstraction* dovrebbero avere loro assegnate responsabilità singole e limitate. La presenza di classi e interfacce senza uno scopo specifico nel design e quindi inutilizzate viola il principio, introducendo nel design questo *smell*. Un altro principio violato oltre a quello di astrazione è il principio *YAGNI (You Aren't Gonna Need It* [52], che raccomanda di evitare l'aggiunta di funzionalità non strettamente necessarie al design.

Le cause che possono portare all'introduzione di *UNA* nel design sono:

- *Design speculativo*: l'introduzione nel design di funzionalità e strutture che potrebbero servire in futuro può portare alla violazione del principio *YAGNI* [52] e all'introduzione di *abstraction* attualmente non utilizzate.
- *Cambio di Requisiti*: il cambiamento di requisiti del progetto potrebbe avere influenza anche sulle *abstraction* impiegate, perciò alcune potrebbero rimanere orfane e non utilizzate.
- *Cancellazione parziale delle abstraction*: quando la *manutenzione* di un progetto viene effettuata senza cancellare *abstraction* vecchie e non più utili per il design, lasciando così nel sistema diverse *unutilizied abstraction*.
- *Paura di rompere il codice*: la cancellazione di *abstraction* potrebbe portare diversi problemi ai programmatori poiché spesso essi decidono di non rimuoverle per paura che vengano ancora utilizzate nel codice. L'introduzione dello *smell* causata da questo timore si verifica soprattutto in progetti che presentano un grande numero di classi e linee di codice.

Nodo di tipo smell nel grafo Il nodo dello *smell* UUA si presenta come un nodo di tipo *smell* con una singola dipendenza verso la classe che presenta il problema.

4.2.1 Impatto sulla qualità del codice e refactoring

Unutilizied Abstraction ha un impatto fortemente negativo su due qualità del progetto, ovvero affidabilità e facilità di comprensione del sistema. La presenza di molte classi inutilizzate infatti potrebbe portare ad errori a run-time, se per esempio una di queste dovesse venire erroneamente invocata portandosi dietro piccoli *bug*. Inoltre l'inquinamento del design da parte di molte classi e interfacce diminuisce la comprensione del progetto aumentandone il carico cognitivo, causando anche problemi secondari come difficoltà nello studio del funzionamento del sistema oppure estensione e manutenzione del codice complicate.

La tecnica di *refactoring* consigliata è l'eliminazione del progetto di tutte le *abstraction* non più necessarie. Nel caso di API pubbliche però l'eliminazione potrebbe non essere la soluzione opportuna, poiché alcune di esse potrebbero ancora essere utilizzate da qualche *client*. La soluzione in questo caso è la segnalazione delle *unutilizied abstraction* come deprecate.

4.2.2 Strategie di identificazione

Per l'identificazione dello *smell Unutilizied Abstraction* è necessaria l'identificazione di tutte le *abstraction* del progetto inutilizzate oppure non utilizzate per il loro naturale scopo. In particolare, come già analizzato nella sua introduzione, bisogna considerare tutte quelle che appartengono a due differenti categorie:

- *Unreferenced Abstractions*, classi concrete non utilizzate da nessuno e senza alcun riferimento all'interno del progetto, interpretato nel grafo come i nodi che non presentano in ingresso alcun arco *dependsOn* proveniente da una classe esterna. Quest'ultima precisazione viene specificata poiché nel grafo di Arcan le *inner class* hanno sempre una relazione *dependsOn* verso il loro contenitore e, senza questa specificazione, le classi *unreferenced* contenenti almeno una classe interna non utilizzata non sarebbero state erroneamente considerate come *smell*. L'utilizzo di una classe interna da parte di una esterna genera invece nel grafo due differenti archi *dependsOn*, uno in ingresso alla classe utilizzata e l'altro a quella che la contiene. In questo caso, anche se la classe contenitore non viene utilizzata, non viene giustamente considerato come *smell*.

- *Orphan Abstractions*, classi astratte o interfacce che non vengono implementate o estese. Per quanto riguarda le classi astratte, bisogna ricercare nel grafo tutti i nodi che le rappresentano senza nessun arco *isChildOf* in ingresso; i nodi di tipo interfaccia invece non devono presentare in ingresso, oltre ad archi *isChildOf*, nemmeno nessuna relazione di tipo *isImplementationOf*. In questo caso non si deve porre molta attenzione

alle classi interne, poiché per la *detection* di questo caso non vengono presi in considerazione gli archi del tipo dependsOn.

4.2.3 Algoritmo

Input un sottografo del grafo principale considerando solamente i nodi che rappresentano classi e interfacce e gli archi *isChildOf*, *dependsOn* oppure *isImplementationOf*.

Output una lista di unit che presentano lo *smell*.

Algoritmo

function UNUTILIZIED-ABSTRACTION-DETECTOR
Prendo la lista di tutti i nodi unit
for Ogni vertice della lista **do**
if Il vertice è di tipo classe o enum **then**
Considero tutti i nodi con un arco dependsOn in uscita verso il vertice
if Il vertice è di tipo classe astratta **then**
Considero tutti i nodi con un arco isChildOf verso il vertice
if Il vertice è di tipo interfaccia **then**
Considero tutti i nodi con un arco dependsOn oppure isChildOf verso il vertice
if I nodi considerati sono 0 oppure sono tutti classi interne **then**
Aggiungi il nodo alla lista degli smell

4.3 Unnecessary Abstraction

Unnecessary Abstraction (*UNA*), definito da G. Suryanarayana et al. [31], si verifica quando un *abstraction* che non è in realtà necessaria (e quindi potrebbe essere evitata) viene introdotta nel design. Questo *smell* viola il principio di astrazione, poiché si verifica l'introduzione nel design di *abstraction* con responsabilità limitate oppure nulle.

Si possono riassumere tre cause principali che portano l'introduzione di questo *smell* nel design:

- *Utilizzo improprio di feature del linguaggio: abstraction* non necessarie possono essere introdotte nel design solamente per convenienza, utilizzando *feature* del linguaggio in maniera impropria. Il caso più diffuso è rappresentato dalle *constant placeholder*, interfacce o classi utilizzate dal programmatore solamente per contenere valori costanti. La generazione

di queste *abstraction* consente al programmatore di implementare o estendere il *placeholder* nella classe desiderata in modo da utilizzare una costante senza la necessità di specificarne il tipo ma solamente attraverso il suo nome.

- *Over-engineering:* vengono definite *over engineered* le *abstraction* introdotte nel design che risultano superflue e prive di un grande significato associato. Un esempio di classe over engineered è mostrato dalla figura 3, dove sono presenti una classe *Customer* contenente un attributo ID che, invece di essere di tipo *String*, è incapsulato da una classe *CustomerID* dedicata, superflua e non necessaria per il sistema.

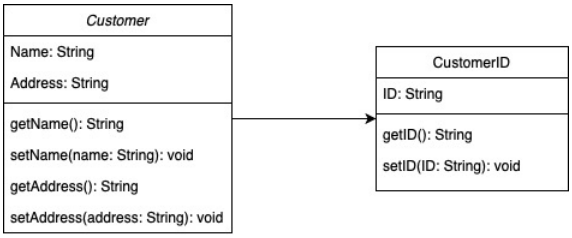

Figure 3: Esempio di classe over engineered

- *Procedural thinking:* se uno sviluppatore, spesso affacciato recentemente allo sviluppo Object Oriented, tende a pensare le classi in maniera procedurale e non dotate di responsabilità e compiti, può introdurre nel codice *abstraction* non necessarie. Questa tendenza si traduce in classi che "svolgono procedure" invece di "essere qualcosa" violando il principio di astrazione, a causa di responsabilità multiple e poco definite. Un esempio di classe procedurale possono essere le classi di *utilities*.

Nodo di tipo smell nel grafo Il nodo dello *smell* presenta un'unica dipendenza verso la sospetta *Unnecessary Abstraction*. Inoltre ha una proprietà chiamata *unnecessaryCase*, che indica quale delle tre casistiche di *UNA* è stata riscontrata nella *unit*.

4.3.1 Impatto sulla qualità del codice e refactoring

Unnecessary Abstraction ha un impatto significativo sulla possibilità di riutilizzo del codice e sulla comprensione del progetto. Riguardo la comprensione, la presenza di numerose interfacce non necessarie incide in maniera negativa poiché aumenta la complessità del design, causando una maggiore difficoltà nell'interpretazione del design e nella chiarezza del progetto.

Anche il riutilizzo del codice subisce un influenza negativa dalla presenza di *Unnecessary Abstraction* poiché *abstraction* senza responsabilità uniche e ben definite e specializzate per un design particolare risultano molto difficili da riutilizzare in contesti differenti.

Se inoltre viene utilizzata un'interfaccia come *constant placeholder* possono verificarsi ulteriori problemi:

- Le classi derivate da quelle che implementano l'interfaccia possono risultare inquinate da costanti che non sono rilevanti per loro.
- Si verifica una violazione dell'incapsulamento in quanto vengono mostrati, attraverso l'interfaccia, dettagli implementativi.
- Quando le costanti sono presenti nelle interfacce, cambiamenti ad esse possono creare problemi ai *client* esistenti.
- Le interfacce rappresentano un protocollo che le classi che lo implementano devono rispettare e l'utilizzo come *constant placeholder* è un abuso del meccanismo di astrazione.

Al fine di rimuovere questo problema dal codice ed aumentare la qualità dello stesso, sono suggerite tre diverse strategie [31] di *refactoring*:

- Le classi procedurali dovrebbero essere eliminate, secondo quanto proposto da Fowler [12].
- Si suggerisce l'eliminazione delle *constant placeholder* per favorire l'utilizzo di costrutti forniti dal linguaggio che si adattano meglio a questa esigenza (un esempio possono essere le enumerazioni).
- Per le classi *over engineered* Fowler consiglia [12] l'applicazione del *Inline Class Refactoring*, che consiste nell'unione delle due classi in una sola. Riferendoci all'esempio presentato in precedenza, l'azione suggerita è l'aggiunta di un parametro *CustomerID* di tipo *String* alla classe *Customer*, eliminando la classe *CustomerID* non necessaria.

4.3.2 Strategia di identificazione

Le regole di identificazione di *Unnecessary Abstraction* sono state divise in tre differenti casistiche, in base alle cause della presenza di questo *smell* definite nella sezione 4.3. Vengono considerate per la valutazione della presenza di *UNA*:

- *Placeholder abstraction* classi oppure interfacce senza alcuna funzione definita al loro interno che contengono solamente attributi costanti. La

ricerca di queste classi ha visto l'esclusione di due categorie principali di *unit*, in quanto risultanti come falsi positivi. Non sono state considerate le enumerazioni, poiché sono per definizione contenitori di costanti, e tutte le classi *exception* o *error*, poiché anche se presentano le caratteristiche delle *constant placeholder* non possono essere considerate come tali. Sono state escluse inoltre tutte le classi che presentano un supertipo, con le uniche eccezioni di padri vuoti oppure contenenti solamente costanti.

Nel grafo le *placeholder abstraction* vengono rappresentate da tutti i nodi senza archi in ingresso di tipo *implementedBy* e, per ogni arco in ingresso del tipo *definedBy*, il nodo corrispondente all'attributo deve avere i parametri *constantAttribute* e *defaultValue* aventi il valore *true*. Vengono esclusi dalla ricerca tutti i nodi *unit* che presentano *componentType* con valore *enum* oppure con nomi che terminano con le stringhe *"Error"* oppure *"Exception"*. Inoltre per le *unit* che presentano archi *isChildOf* in uscita viene controllato che i supertipi non presentino alcun arco in ingresso oppure che presentino le stesse caratteristiche definite in precedenza.

- *Over-engineered* classi concrete che definiscono solamente un attributo e due funzioni, rappresentanti i metodi *getter* e *setter*. Inoltre queste classi devono essere definite come attributo in solamente un'altra *abstraction* e non possono presentare alcun supertipo che non sia vuoto, poiché altrimenti la classe subirebbe una modifica a causa degli elementi ereditati. Per la definizione di queste regole si è replicato il caso descritto dalla figura 3. È stato deciso inoltre di inserire il limite di solamente un utilizzo come parametro da un'altra *abstraction* poiché altrimenti la classe potrebbe avere significato nel sistema di riferimento, come avviene ad esempio nell'applicazione del pattern *Object Identifier* [8].

  Le condizioni per l'identificazione si traducono nella ricerca di nodi del grafo che presentano in ingresso un solo arco di tipo *definedBy* e massimo due *implementedBy*. Viene controllato anche che il nome di ogni funzione rappresentata dall'arco *implementedBy* sia effettivamente una funzione *getter* o *setter* per l'attributo della classe, ovvero presenti un nome del tipo {get/set}{nomeAttributo}. Inoltre il nodo deve presentare solamente un arco in ingresso di tipo dependsOn da un'altra *unit*, che a sua volta ha in ingresso un'arco *containedIn* da un attributo dello stesso tipo della classe sotto esame. Se infine il nodo presentasse un arco in uscita di tipo *isChildOf*, la *unit* del supertipo non dovrebbe aver alcun arco in ingresso *definedBy* oppure *implementedBy* ed anche eventuali supertipi di questa *unit* non dovrebbero presentare nessun arco di queste due tipologie. Un esempio di questa struttura può essere osservato nella

figura 4 (dove alla classe Customer è stato aggiunto un ulteriore attributo *name*).

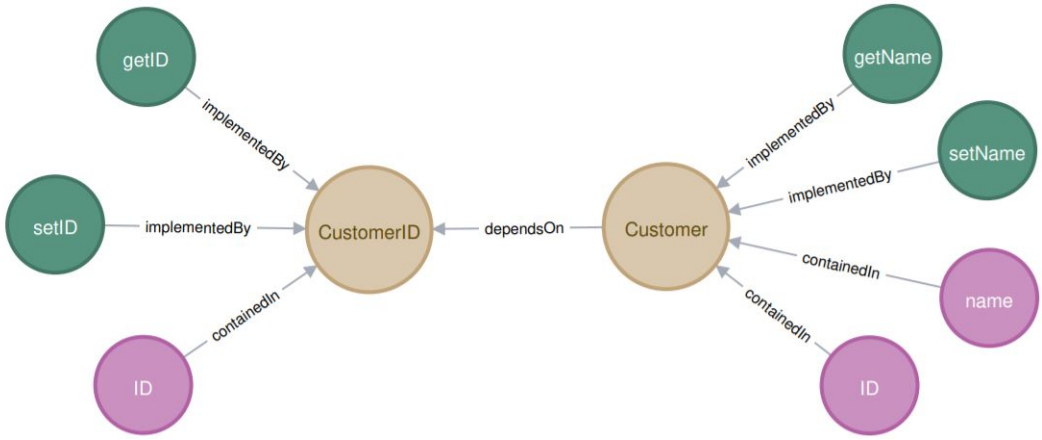

Figure 4: Esempio classi overengineered nel grafo

- *Procedural class* classi concrete o astratte senza attributi e con solamente una oppure due funzioni definite al loro interno. Non sono state considerate nella ricerca tutte quelle classi che presentano dei supertipi, per due motivi principali. In primo luogo, se la classe avesse dei supertipi potrebbe ereditare metodi e attributi che potrebbero far sì che non rispetti più i vincoli appena definiti. Inoltre le classi procedurali sono spesso introdotte da persone non esperte nella programmazione orientata agli oggetti e prive di significato nel design, cosa che potrebbe non verificarsi se è presente una gerarchia.

  Per quanto riguarda il grafo, bisogna ricercare i nodi con *componentType* di valore *class* oppure *abstract_class* che presentano in ingresso uno oppure due archi di tipo *implementedBy* e nessun arco di tipo *containedIn* in ingresso oppure *isChildOf* in uscita.

4.3.3 Algoritmo

Input un sottografo del grafo principale, contenente i nodi *unit* e le *function* e *attribute* che esse definiscono. Gli archi considerati sono di tipo *dependsOn*, *implementedBy* e *definedIn*.

Output la lista delle classi affette da questo *smell* e la relativa causa che ha portato alla loro identificazione.

Algoritmo

function UNNECESSARY-ABSTRACTION-DETECTION(G)
Prendo la lista di tutti i vertici di tipo unit
for Ogni vertice nella lista **do**
if IS-PROCEDURAL-CLASS(vertice) **then**
Aggiungo il vertice alla lista degli smell
if IS-CONSTANT-PLACEHOLDER(vertice) **then**
Aggiungo il vertice alla lista degli smell
if IS-OVER-ENGINEERED(vertice) **then**
Aggiungo il vertice alla lista degli smell

function IS-PROCEDURAL-CLASS(vertice)
if Il vertice è una classe concreta o astratta **then**
if Il vertice non ha archi in ingresso containedIn **then**
if Il vertice non ha archi in uscita isChildOf **then**
if Il vertice ha 1-2 archi in ingresso implementedBy **then**
return la classe è una Unnecessary Abstraction

function IS-CONSTANT-PLACEHOLDER(vertice)
if Il vertice non rappresenta enumerazioni o classi di errore **then**
if La classe non ha archi in ingresso implementedBy **then**
if La classe ha archi in uscita isChildOf **then**
if Almeno un supertipo non è una constant placeholder oppure non è vuoto **then**
return la classe non è una Unnecessary Abstraction
if Tutti i nodi con arco containedIn verso questo vertice hanno gli attributi constantAttribute e defaultValue true **then**
return la classe è una Unnecessary Abstraction

function IS-OVER-ENGINEERED(vertice)
if Il vertice ha un solo arco in ingresso containedIn **then**
if Il vertice ha max 2 archi implementedBy in ingresso con nomi delle funzioni collegate corrispondono a getNomeAttributo oppure setNomeAttributo **then**
if Il vertice ha solo un'arco in ingresso dependsOn da un vertice che definisce un attributo dello stesso tipo della classe rappresentata dal vertice **then**
return la classe è una Unnecessary Abstraction

5 Riconoscimento degli architectural smell su progetti reali

Questo capitolo presenta nel dettaglio tutte le attività svolte riguardanti l'esecuzione e la validazione degli algoritmi sviluppati su diversi progetti open-source. Dopo una descrizione dei progetti, saranno analizzate nel dettaglio l'esecuzione degli algoritmi, le diverse attività di validazione svolte e i risultati ottenuti. Sarà inoltre effettuato un confronto sulle differenze della *detection* di Arcan rispetto a quella del *tool* Designite [28]. Un'ultima sezione esporrà poi i differenti problemi affrontati durante lo svolgimento di queste attività. Prima della presentazione delle tematiche descritte in precedenza, sarà effettuata una breve digressione riguardo le attività di *testing* svolte per la verifica degli algoritmi durante il loro sviluppo

Approccio Test Driven Development Durante lo sviluppo degli algoritmi sono stati effettuati i loro test seguendo l'approccio *Test Driven Development (TDD)* [5]. Questa strategia prevede la scrittura dei test anticipata rispetto all'implementazione degli algoritmi, in modo che lo sviluppo dell'applicativo sia orientato alla soddisfazione degli stessi. Al fine di applicare il *TDD* è stato utilizzato il framework JUnit [44].

L'implementazione dei test unitari è stata fondamentale per la definizione degli obiettivi da raggiungere da parte degli algoritmi di *detection*. Attraverso queste verifiche è stato possibile effettuare due tipologie di analisi differenti:

- *Analisi di strutture sintetiche*, cioè grafi generati manualmente all'interno del codice dei test, utilizzati al fine di valutare il comportamento dell'algoritmo considerate diverse situazioni in termini di nodi e relazioni tra essi. In particolare attraverso le strutture sintetiche sono stati analizzati tutti i casi limite riguardanti le strutture del grafo, come ad esempio il numero di nodi coinvolti oppure le loro tipologie. L'obiettivo della scrittura di questi test è stato quindi il test su piccola scala, testando la capacità dell'algoritmo a riconoscere gli *smell* in differenti scenari con un numero basso di nodi e archi e strutture ben definite.

- *Analisi di piccoli progetti open source*, sono stati direttamente analizzati anche due progetti Java *open source* di piccole dimensioni, Junit 4.13 [44] e Jsoniter [43]. Lo scopo dell'analisi di questi due progetti è stata la verifica, direttamente dal grafo generato da Arcan tramite la piattaforma Neo4J [58], che il numero degli *smell* trovati attraverso *query* Cypher [57] effettuate sul grafo fosse equivalente a quello trovato dagli algoritmi di *detection*. Questi test quindi avevano come obiettivo la verifica che le

strutture sintetiche riconosciute dall'algoritmo fossero individuate anche all'interno di progetti complessi e organizzati.

5.1 Descrizione progetti utilizzati

Per le attività di individuazione degli *smell* e validazione del lavoro svolto sono stati selezionati dieci progetti *open-source* differenti, sviluppati da *Apache Software Foundation* [49] e disponibili sulla piattaforma *GitHub* [42]. Una descrizione sommaria dei progetti è fornita dalla tabella 1, presentando informazioni riguardanti numero della release analizzata, dominio applicativo e collegamento al progetto disponibile su *Github*.

A causa di alcuni problemi derivati dalle librerie utilizzate per il *parsing* non è stato possibile analizzare tutti e dieci i progetti completi in tutti i loro componenti, perciò per alcuni di essi è stata necessaria l'analisi solamente del modulo principale del sistema. Nello specifico, per i progetti *Druid*, *Flink* e *Geode* è stato utilizzato il modulo *core* mentre per *Beam* il modulo *beam-runners core-java*. Per l'analisi inoltre sono state escluse tutte le classi di test, in quanto considerate non molto significative ai fini del lavoro di ricerca degli *smell* e loro validazione.

Informazioni più dettagliate sui progetti riguardanti modulo analizzato, righe di codice (LOC) e numero di unit e package analizzati sono verificabili nella tabella 2.

Progetto	**Link**	**Release**	**Dominio applicativo**
Accumulo	[34]	2.0.0	Distributed key/value data store
Beam	[35]	2.20.0	Unified programming model
Bookkeeper	[36]	4.10.0	Storage service
Cassandra	[37]	3.11.6	Distributed NoSQL DBMS
Druid	[39]	0.18.0	Distributed data store
Flink	[40]	1.10.0	Distributed processing engine
Geode	[41]	1.12.0	In-memory data management system
Kafka	[45]	2.5.0	Distributed platform
Skywalking	[46]	7.0.0	Application performance monitor system
Zookeeper	[47]	3.6.0	Services for distributed system

Table 1: Informazioni progetti analizzati

Progetto	Modulo	N° classi	N° package	N° LOC
Accumulo	-	3 789	212	43 682 818
Beam	core-java	201	8	52 414
Bookkeeper	-	2 036	252	1 495 659
Cassandra	-	3 910	116	33 968 645
Druid	druid-core	736	63	369 487
Flink	flink-core	915	49	148 600
Geode	geode-core	4 293	190	2 236 679
Kafka	-	2 389	138	575 967
Skywalking	-	979	526	141 674
Zookeeper	-	755	52	269 378

Table 2: Dettaglio progetti analizzati

5.2 Risultati della detection

L'analisi dei progetti è stata effettuata attraverso l'esecuzione di Arcan, in grado di generare in *output* diversi file in formato *csv* per ogni sistema analizzato. I file generati per un singolo progetto forniscono informazioni riguardanti:

- I valori delle metriche (come *instability, LOC*) calcolate su tutti gli elementi presenti nell'applicativo.
- Le unit colpite dallo *smell* (con relazione *affects* verso il nodo che lo rappresenta) e la tipologia riscontrata.
- Tutte le unit collegate ad un nodo di tipo *smell*, con indicazione sulla tipologia di relazione tra i due elementi.

L'analisi di questi tre *file* ha permesso lo studio e la validazione dei risultati ottenuti dagli algoritmi.

La *detection* degli *smell* sui progetti *Apache* ha evidenziato la presenza di numerose istanze. In particolare sono stati rilevate 708 *Subclasses Do Not Redefine Methods*, 2033 *Unutilizied Abstraction* e 2491 *Unnecessary Abstraction*. Il dettaglio riguardante il numero di *smell* trovati nei vari progetti è presentato nella tabella 3.

Come si evince dai risultati, *Subclasses Do Not Redefine Methods* è lo *smell* che presenta il numero minore di istanze; la causa principale di ciò può essere identificata nel numero differente di elementi analizzati per la ricerca. Infatti *UUA* e *UNA* effettuano la loro *detection* controllando ogni *abstraction* all'interno del progetto, mentre questo *smell* considera solamente le gerarchie presenti (e le relative *unit* coinvolte), che sono sicuramente in numero minore rispetto alle classi e interfacce del sistema.

Progetto	Subclasses Do Not Redefine Methods	Unutilizied Abstraction	Unnecessary Abstraction
Accumulo	98	177	1113
Beam	5	32	13
Bookkeeper	52	189	96
Cassandra	96	137	639
Druid	6	128	52
Flink	66	185	47
Geode	143	304	166
Kafka	138	134	119
Skywalking	65	683	208
Zookeeper	39	64	38
Totale	708	2033	2491

Table 3: Numero di istanze individuate nei progetti

Per quanto riguarda *Unutilizied Abstraction* invece si può affermare che la presenza di un alto numero di istanze può essere causato da due fattori principali. La semplicità con la quale questo *smell* può essere introdotto nel design è sicuramente uno di questi, poiché le cause della sua manifestazione sono molto comuni nello sviluppo software. Inoltre la presenza di *UUA* può verificarsi anche in seguito alla manifestazione "effetto collaterale" della presenza di *Unnecessary Abstraction*. Un esempio può essere la creazione di una classe *constant placeholder*, quando la *abstraction* è rappresentata da un'interfaccia che non viene implementata ma solamente utilizzata via *dot notation*. In questo caso il nodo non avrà alcun arco in ingresso ad eccezione di *dependsOn*, facendo risultare l'interfaccia sia come *unnecessary* che *unutilizied*.

In merito a *Unnecessary Abstraction* ritengo che la definizione non molto dettagliata dello *smell* e soprattutto la necessità di identificare il caso delle classi procedurali abbiano portato ad un elevato numero di istanze individuate. La situazione descritta da *UNA* è un po' ambigua, siccome non espone come gli altri una scenario ben definito ma si riferisce genericamente a classi "non necessarie per il design". Attraverso le cause che lo introducono [31] è stata possibile la definizione di tre sue diverse tipologie, ma è comunque presente il caso delle *procedural class* che risulta molto difficoltoso da identificare, poiché le classi procedurali derivano principalmente dalle intenzioni dello sviluppatore piuttosto che dalla sua struttura e dai metodi e attributi definiti. La loro ricerca ha influenzato in maniera significativa l'alto numero di *Unnecessary Abstraction* presenti, come evidenziato dai dati riportati nella tabella 4, dove si evince che il 92% dei casi riportati sono dovuti proprio a questa categoria di classi (2275 istanze su 2491 totali), a differenza di *constant placeholder* e *over engineered* che hanno un incidenza sul totale rispettivamente del 7% e

1% (con 178 e 17 casi individuati). L'alto di numero di classi procedurali è a sua volta influenzato fortemente dal progetto Accumulo, dove sono presenti in totale 1113 classi identificate come *smell*, che rappresentano il 44% dei casi totali riscontrati su tutti i progetti. Anche qui la casistica più frequente è quella delle *procedural class*, con ben 996 istanze trovate.

Tipologia	**Istanze smell**	**Incidenza totale**
Constant placeholder	178	7%
Over engineered	17	1%
Procedural class	2275	92%

Table 4: Dettaglio istanze smell Unnecessary Abstraction

5.3 Validazione dei risultati

Questa sezione descrive in dettaglio la fase di validazione degli algoritmi, con descrizione delle attività svolte e discussione dei risultati ottenuti. Prima di approfondire questi aspetti è però necessaria l'introduzione di alcuni concetti fondamentali, al fine di favorire la comprensione delle attività descritte.

I primi concetti introdotti sono *true positive* (vero positivo, VP) e *false positive* (falso positivo, FP). Con il termine *true positive* si indicano le istanze di smell trovate dagli algoritmi che risultano come problemi reali mentre *false positive* indica il contrario, ovvero tutte le istanze individuate che però non risultano tali.

Il numero di veri e falsi positivi trovati da un algoritmo è fondamentale per il calcolo della metrica *Precision* [20]. Questa metrica è indice della precisione dell'algoritmo sviluppato in termini di numero di veri positivi in rapporto al totale di casi analizzati e può essere descritta dalla formula seguente:

$$Precision = \frac{false\ positives}{true\ positives + false\ positives}$$

Per lo svolgimento delle attività di validazione sono state selezionate un numero fisso di 50 classi per ogni *smell* in ogni progetto, segnalate dall'algoritmo di *detection*. Nei casi in cui il numero di istanze fosse minore, sono state prese in considerazione tutte quelle presenti anche in numero inferiore.

Ogni elemento coinvolto è stato analizzato manualmente attraverso l'osservazione del codice sorgente, allo scopo di verificare la presenza effettiva dello *smell* e analizzare le cause che hanno portato l'elemento a essere considerato come tale. Per la validazione di *Unnecessary Abstraction* sono state prese, ove possibile, un numero uguale di classi per ogni tipologia presente. Le successive analisi presenteranno i risultati ottenuti dalla *detection* di *UNA* considerando sia i valori generici ottenuti per lo *smell* sia i tre casi distinti.

Lo studio dei falsi positivi riscontrati ha fatto emergere la necessità di effettuare modifiche agli algoritmi e agli algoritmi di *parsing*, al fine di rimuovere alcune tipologie ricorrenti di falsi positivi e migliorare la *precision* della *detection*.

Al fine di garantire una maggior precisione nella validazione degli algoritmi, è stata effettuata un'attività di validazione incrociata con un'altra laureanda [27]. Questa attività ha portato a un'analisi e validazione doppia dei falsi positivi, con successive valutazioni e verifica dei casi dove il risultato delle due non coincideva.

La tabella 5 presenta in maniera sintetica i risultati ottenuti dalla validazione delle *detection* degli algoritmi *SR*, *UNA* e *UUA*, e fornisce informazioni sul numero totale di *smell* analizzati e il numero di falsi positivi riscontrati. Un'ulteriore colonna indica il valore della metrica *Precision* per l'algoritmo considerato.

Smell	**Analizzati**	**VP**	**FP**	**Precision**
Subclasses Do Not Redefine M.	400	359	41	89.75%
Unutilizied Abstraction	460	439	21	95.43%
Unnecessary Abstraction	448	330	118	73.66%

Table 5: Panoramica validazione

Le tre sottosezioni successive analizzeranno nel dettaglio la validazione di ogni singolo *smell*. Verrà posta maggior enfasi sulle tecniche utilizzate, sui *tool* impiegati come supporto e saranno approfondite le diverse cause della presenza di falsi positivi.

5.3.1 Subclasses Does Not Redefine Methods

Per effettuare la validazione di *Subclasses Do Not Redefine Methods* sono state analizzate le superclassi di ogni unit segnalata dall'algoritmo come istanza, al fine di verificare che le due classi condividessero almeno un metodo. Fondamentale è stato il supporto fornito dall'ambiente di sviluppo IntelliJ IDEA [55], in grado di fornire informazioni riguardanti l'*override* delle funzioni all'interno delle diverse classi o interfacce.

La *precision* calcolata su questo algoritmo è del 89.75%, con 359 veri positivi su un totale di 400 unit analizzate. Una prima validazione di *SR* aveva mostrato un valore di *precision* del 73%, con 418 istanze validate di cui 307 veri positivi. Al termine di questa fase è stata notata la presenza di numerosi falsi positivi, che ha portato allo studio dei casi riscontrati al fine di effettuare

modifiche per favorire la diminuzione del loro numero e migliorare la precisione dell'algoritmo. A tal proposito, è stata introdotta un'unica modifica alle strategie di detection: nella ricerca dei metodi ereditati dal supertipo, sono stati considerati anche tutti i metodi delle interfacce implementate da classi astratte. Questa modifica ha permesso l'aumento del valore di *precision* di circa 17 punti percentuali, passando da 307 VP su 418 istanza analizzate a 359 VP su un totale di 400. La correzione effettuata ha portato inoltre alla eliminazione di 166 istanze totali, di cui 78 falsi positivi e 5 veri positivi (dei restanti non si hanno informazioni perché non selezionati per la validazione).

Analisi dei falsi positivi Nonostante le modifiche effettuate, è stata riscontrata la presenza di ulteriori situazioni di istanze risultate come false positive, che per diversi motivi non è stato possibile eliminare. Tra le differenti situazioni, verrà approfondita solamente quella relativa alla estensione delle classi esterne al sistema, poiché è stata riscontrata con una certa rilevanza durante la validazione. Siccome i componenti di Arcan, in particolare il *System Reconstructor*, costruiscono il grafo solamente a partire dal codice sorgente ricevuto come input, diverse dipendenze esterne non vengono considerate nella generazione del grafo. Può verificarsi il caso di presenza nel sistema di una gerarchia, che presenta come primo elemento una classe di sistema. Se una classe di questa gerarchia, non figlia direttamente della classe di sistema, ridefinisce solamente i suoi metodi, viene considerata erroneamente come *smell* poiché quest'ultima non è presente nel grafo in quanto dipendenza esterna. Se invece si considera una classe figlia direttamente di una classe esterna, la relazione tra le due classi non viene considerata nella ricerca dello *smell* poiché non viene rappresentata nel grafo. Facendo un esempio del primo caso, se una classe B estende una classe A che a sua volta è sottotipo di Beans [59] e B effettua *override* del metodo *instantiate* di Beans, la classe B risulterebbe erroneamente come contenente lo *smell SR*.

5.3.2 Unutilizied Astraction

La validazione di *Unutilizied Abstraction* è stata effettuata verificando gli utilizzi di ogni classe e interfaccia segnalata come istanza dello *smell* all'interno del progetto. I casi risultati come falsi positivi hanno comportato un'ulteriore analisi della struttura del grafo, al fine di comprendere le motivazioni che hanno portato alla loro *detection*.

Uno strumento fondamentale durante la validazione è stato l'ambiente di sviluppo IntelliJ IDEA [55]. Grazie allo strumento messo a disposizione da questo *IDE* per la ricerca delle *references* di una classe o interfaccia, è stato possibile ottenere informazioni riguardanti gli utilizzi dell'elemento analizzato all'interno del progetto.

Progetto	Analizzati	Veri positivi	Falsi positivi
Accumulo	50	50	0
Beam	5	4	1
Bookkeeper	50	45	5
Cassandra	50	43	7
Druid	6	6	0
Flink	50	49	1
Geode	50	48	2
Kafka	50	50	0
Skywalking	50	49	1
Zookeeper	39	15	24
Totale	400	359	41

Table 6: Risultati validazione Subclasses Do Not Redefine Methods

La *precision* riscontrata sull'algoritmo è del 95.43%, con 418 veri positivi su 439 istanze analizzate. Grazie all'alta percentuale riscontrata, non è stato ritenuto necessario effettuare modifiche agli algoritmi successive alla prima validazione.

Analisi di falsi positivi La maggioranza delle istanze riscontrate è riferito a classi di esempio, che sono parte del progetto ma non vengono utilizzate da nessuno. Un ulteriore causa identificata che ha portato la presenza di falsi positivi è legata al *parsing*, poiché alcune *references* di classi trovate nel codice non hanno trovato riscontro nel grafo delle dipendenze. È stato notato in particolare che gli utilizzi all'interno di classi anonime non vengono rilevati dal *tool*, in quanto nel grafo non sono presenti i nodi rappresentanti classi anonime e le relative dipendenze.

5.3.3 Unnecessary Abstraction

Le attività e i risultati della validazione di *Unnecessary Abstraction* verranno presentati sia considerando lo *smell* dal punto di vista generico che analizzando i tre casi distinti dello stesso. I valori di *precision* calcolati per le tre differenti categorie possono essere osservati nella tabella 8.

La validazione di questo *smell* ha presentato diversi problemi, a causa di numerosi casi particolari e difficoltà nell'ideazione di strategie e sviluppo di algoritmi, testimoniate dal valore di *precision* di 73.66%, il minore tra i tre calcolati. Come mostrato dalla tabella 8, la precisione dell'algoritmo è fortemente influenzata dal caso *procedural class*, che presenta un alto numero di

Progetto	Analizzati	Veri positivi	Falsi positivi
Accumulo	50	44	6
Beam	32	32	0
Bookkeeper	50	50	0
Cassandra	50	50	0
Druid	50	49	1
Flink	50	50	0
Geode	50	48	2
Kafka	50	41	9
Skywalking	50	47	3
Zookeeper	28	28	0
Totale	460	439	21

Table 7: Risultati validazione Unutilizied Abstraction

istanze analizzate e una precisione del 65.33%. La precisione dei casi *constant placeholder* e *over engineered* invece assume i valori rispettivamente del 86.75% e 87.50%.

Le tre casistiche dello *smell* hanno richiesto differenti attività per la validazione dei risultati. Per i casi *constant placeholder* e *over engineered*, sono state analizzate le classi per verificare che seguissero i vincoli descritti nel capitolo precedente. Per le *procedural class* invece il controllo riguardava non solo il rispetto o meno dei vincoli, ma anche le motivazioni dello sviluppo della classe per comprendere se la stessa potesse essere considerata come *smell* o meno. Per fare questo, sono stati analizzati gli attributi utilizzati dalle diverse funzioni: venivano giudicate *procedural class* tutte quelle che svolgevano una banale trasformazione dei dati in ingresso e producevano in uscita un risultato, mentre classi che effettuavano trasformazioni al sistema oppure ad altri oggetti in modo permanente non venivano considerate tali. Un ulteriore verifica comprende l'eventuale struttura nella quale era inserita, i package e le interfacce implementate, i commenti alla classe e alle funzioni e i nomi delle funzioni stesse.

Anche questa *detection*, come già effettuato per *Subclasses Do Not Redefine Methods*, ha subito un adattamento al termine della prima fase di validazione. Prima delle modifiche agli algoritmi, la *precision* calcolata su questo algoritmo era 59.66%, con 278 veri positivi su 466 totali. Si è verificato quindi un incremento di circa 14 punti percentuale, portandolo all'attuale valore di 73.66%. Il numero di istanze segnalate da Arcan è inoltre calato di 722 unità, con la rimozione di 134 istanze selezionate precedentemente per la prima *validation*, tra le quali si registrano 8 veri positivi e 126 falsi positivi (94% dei casi validati e rimossi).

Nelle tre casistiche introdotte per lo *smell*, sono state effettuate le modi-

fiche elencate di seguito:

- Per la ricerca delle *constant placeholder* non sono state più considerate le classi che possiedono un supertipo. Questa modifica ha comportato un incremento della *precision* da 78.57% a 86.75%, con un passaggio da 136 VP su 168 totali a 144 VP su 166 attuale.

- L'introduzione di limiti più stringenti per la *detection* di *over engineered* ha fatto registrare un aumento della precision da 27.59% a 87.50%. La differenza molto grande in termini di punti percentuale è dovuta al numero esiguo di istanze trovate, con un numero di casi considerati prima e dopo le modifiche effettuate rispettivamente di 29 e 8. I cambiamenti introdotti riguardano il controllo del nome delle funzioni, che viene fatto per intero e non solamente controllando che il nome inizi con le stringhe *"get"* oppure *"set"*. Viene verificato inoltre che le classi sospette non ereditino nulla da eventuali supertipi.

- La rimozione delle classi che presentavano supertipi nella ricerca delle *procedural class* ha comportato un aumento della metrica *precision* da 51.30% a 65.33%, passando da 138 VP su 269 istanze a 179 VP su 274.

Caso	**Analizzati**	**VP**	**FP**	**Precision**
Constant Placeholder	166	144	22	86.75%
Over-engineered	8	7	1	87.50%
Procedural classes	284	179	95	65.33%

Table 8: Dettaglio casistica Unnecessary Abstraction

Analisi di falsi positivi Le cause della presenza di numerose istanze risultate come falsi positivi può essere ricercata in tre fattori fondamentali:

1. A differenza degli altri due *smell*, *UNA* presenta un gran numero di casi particolari e situazioni limite, emerse soprattutto durante le fasi di validazione. Di conseguenza le regole di detection sono risultate molto più difficili da implementare, soprattutto nell'ottica di inserire limiti stringenti per le varie casistiche.

2. Le difficoltà riscontrate nella detection del caso *procedural class* hanno un impatto considerevole sul valore di *precision* calcolato per l'algoritmo di detection, in quanto è il caso più rappresentato dei tre con 62% delle istanze validate di *Unnecessary Abstraction* e presenta il valore di *precision* minore (65.33%).

Al fine di approfondire le cause della presenza di falsi positivi nel dettaglio, saranno ora analizzati i risultati della validazione delle diverse tipologie singolarmente.

La causa più comune della presenza di falsi positivi *constant placeholder* è rappresentata da classi che presentano pochi attributi, generalmente da 1 a 3, non considerate come placeholder in quanto contenenti campi di significati e tipologie differenti o diverse istanze di classi anonime. Spesso le classi presentano solamente un solo attributo come unica istanza di un oggetto, una situazione simile al design pattern *Singleton* [13]. Una possibile soluzione per la loro rimozione potrebbe essere l'introduzione di una soglia minima di attributi per essere considerata un *placeholder*, che porterebbe però a una possibile limitazione dell'algoritmo siccome potrebbero rimanere escluse dall'algoritmo diverse classe considerate ad oggi come vere positive.

L'esiguo numero di istanze *over engineered* dello *smell* e di falsi positivi riscontrati (rispettivamente 8 e 1) non rendono possibile alcuna analisi su di essi. L'unico falso positivo è relativo a una classe che presenta tutte le caratteristiche che le fanno considerare *over engineered* ma contiene un'altra classe al suo interno. Questo causa anche che il nodo della dipendenza in ingresso alla classe sia in realtà dato dalla sua classe interna (nel grafo ogni classe interna ha un arco *dependsOn* verso la unit che la contiene), rendendola così un falso positivo.

Infine le *procedural class* presentano caratteristiche molto comuni anche per classi non procedurali, causando così la *detection* di numerose istanze considerate poi come falsi positivi. Anche se le classi presentano tutte le caratteristiche necessarie per essere considerate procedurali, può succedere che molto spesso non siano comunque ritenute tali poiché sono presenti diversi fattori esterni alla sola struttura della classe difficoltosi da analizzare attraverso un algoritmo di analisi su un grafo. Durante la valutazione delle istanze di questo algoritmo vengono infatti presi in considerazione aspetti quali nomi assegnati agli elementi, codice ed eventuali commenti presenti, che influiscono in maniera significativa sulla valutazione della classe come procedurale o meno. Inoltre in un progetto la presenza di alcune classi procedurali è comune, come ad esempio le classi che rappresentano *Thread* oppure quelle che che implementano il metodo *main*, ma queste vengono individuate erroneamente da Arcan come *smell*, generando casi di falsa positività.

Progetto	Analizzati	Veri positivi	Falsi positivi
Accumulo	50	37	13
Beam	17	11	6
Bookkeeper	50	38	12
Cassandra	50	26	24
Druid	50	41	9
Flink	47	38	9
Geode	50	35	15
Kafka	50	43	7
Skywalking	50	40	10
Zookeeper	38	21	17
Totale	452	330	122

Table 9: Risultati validazione Unnecessary Abstraction

5.4 Confronto dei risultati ottenuti con il tool Designite

Questa sezione propone un confronto tra i *tool* Arcan e Designite [28] , attraverso la *detection* degli *smell SR, UUA e UNA* sui progetti Apache [49] presentati nella sezione 5.1.

Il confronto non affronterà solamente la differenza tra la detection tra i due *smell* in termini di istanze trovate, ma verranno effettuate anche analisi delle strategie di riconoscimento adottate dai due tool. Le strategie e gli algoritmi di riconoscimento sono stati esaminati attraverso ispezione del codice sorgente di Designite disponibile sulla piattaforma GitHub [38].

È stato selezionato il tool Designite in merito a due considerazioni fondamentali:

1. È in grado di effettuare la detection di *Unutilizied Abstraction*, *Unnecessary Abstraction* e di *Broken Hierarchy (BH)*, uno *smell* analogo a *Subclasses Do Not Redefine Methods* citato come alias di *BH* da G. Suryanarayana [31].

2. La sua natura *open-source* permette l'analisi del codice in maniera semplice e veloce.

Per il confronto è stata preferita la presentazione degli algoritmi singolarmente, al fine di evidenziare in maniera chiara la discrepanza tra le strategie adottate e i valori ottenuti. La tabella 10 mette in risalto le diversità riscontrate nella detection dei progetti analizzati. Per lo *smell Unnecessary Abstraction* è indicato tra parentesi il numero di *constant placeholder* trovati poiché Designite ricerca solamente questo particolare caso.

Tool	Nº SR	Nº UUA	Nº UNA
Arcan	708	2033	2491 (178)
Designite	1623	6153	440

Table 10: Differenza in numero assoluto per smell

Come mostrato dai dati presentati dalla tabella 10, il *tool* Designite è in grado di trovare un numero maggiore di istanze rispetto ad Arcan in merito a *Subclasses Do Not Redefine Methods* e *Unutilizied Abstraction*, rispettivamente con una differenza di 915 e 4120. Per quanto riguarda *Unnecessary Abstraction* è Arcan a trovare un numero di *smell* maggiore con 2051 istanze in più ma, limitando solamente il confronto al caso *constant placeholder*, il numero di trovato da Designite è maggiore di 262 unità.

Il confronto tra questi *smell* verrà approfondito nei tre paragrafi successivi. L'analisi è stata svolta seguendo la stessa struttura: a una piccola introduzione seguono considerazioni e differenze tra gli algoritmi di *detection*, per concludere con la valutazione dei risultati ottenuti.

5.4.1 Subclasses Do Not Redefine Methods

Come anticipato in precedenza Designite è in grado di fare la *detection* di *Broken Hierarchy*, analogo allo *smell Subclasses Do Not Redefine Methods* e presentato anche come suo alias [31].

Broken Hierarchy si verifica quando un supertipo e il suo sottotipo non condividono una relazione IS-A con conseguente interruzione della sostituibilità. Lo *smell* può presentarsi in tre differenti forme:

- I metodi del supertipo sono ancora applicabili o rilevanti nel sottotipo, sebbene non sia condivisa una relazione IS-A.
- Il sottotipo eredita funzioni del supertipo che non sono rilevanti o accettabili per i sottotipi.
- L'implementazione del sottotipo rifiuta esplicitamente metodi irrilevanti o inaccettabili ereditati dal supertipo.

Nonostante la descrizione di *SR* e *BH* possa sembrare differente, in tutte e due gli *smell* non si verifica la condivisione della relazione IS-A da parte delle due classi e viene interrotta la sostituibilità.

Strategie di identificazione Il controllo effettuato da Designite per la ricerca di *Broken Hierarchy* è analogo a quello effettuato da Arcan, poiché viene effettuata una ricerca di tutte le classi che non condividono almeno un

metodo con i loro supertipi. Di seguito è illustrato l'algoritmo utilizzato da Designite per la *detection* di Broken Hierarchy:

function DESIGNITE-BH-DETECTOR( )
  if Il *type* analizzato ha *supertypes* e almeno un metodo pubblico **then**
    for Ogni *supertype* del *type* analizzato **do**
      if *type* e *supertype* non condividono almeno un metodo **then**
        return l'elemento è una Broken Hierarchy

Si può notare una differenza tra i due algoritmi nella ricerca dei metodi, poiché Designite è in grado di effettuare una loro distinzione per modificatore (*public, private* oppure *protected*) e valuta solamente i metodi effettivamente ereditati da una classe, mentre Arcan considera tutte le funzioni definite dai supertipi indipendentemente dalla loro visibilità. Alcune analisi riguardanti il funzionamento e la struttura del codice di Designite hanno mostrato che per il controllo di *Broken Hierarchy* e in particolare per verificare le condivisione di un metodo da parte di due classi viene utilizzato il nome della funzione e non la *signature*. Anche Designite quindi considera i casi di *overloading* come ridefinizione del metodo, analogamente a quanto è stato deciso per la *detection* in Arcan.

Analisi risultati Nonostante le analogie tra le strategie e gli algoritmi di *detection*, il numero di istanze trovate da Designite (1623) è maggiore rispetto ad Arcan (708), con un divario di 915 unità. È complicato effettuare una ricerca delle cause di questa discrepanza ma una di queste può essere identificata con la differenza nella scelta delle classi da analizzare, poiché Arcan effettua un filtraggio delle *abstraction* rimuovendo tutte le interfacce e le classi di *exception* che Designite invece esamina (si tratta di 286 istanze).

Arcan potrebbe inoltre aver valutato come non affette dallo *smell* diverse *abstraction* che condividono il nome del metodo con una funzione definita nella gerarchia del padre ma dotata di modificatore *private*, non ereditabile quindi dai suoi sottotipi. Designite invece, essendo in grado di discriminare le funzioni ereditate in base al modificatore, non soffre di questo problema.

5.4.2 Unutilizied Abstraction

Designite è in grado di effettuare la *detection* anche dello *smell Unutilizied Abstraction*, rendendo così possibile il confronto con Arcan. I due *tool* presentano strategie di identificazione leggermente differenti, che causano una discrepanza sostanziale nel numero di istanze identificate con 4120 unità in più riscontrate da Designite.

Progetto	Nº smell Arcan	Nº smell Designite
Accumulo	98	71
Beam	5	5
Bookkeeper	52	149
Cassandra	96	247
Druid	6	86
Flink	66	112
Geode	143	533
Kafka	138	243
Skywalking	65	109
Zookeeper	39	68
Totale	708	1623

Table 11: Confronto istanze Subclasses Do Not Redefine Methods

Strategie di identificazione La *detection* di Unutilizied Abstraction da parte del *tool* Designite comporta piccole differenze con le strategie di identificazione individuate per Arcan. L'algoritmo definito da Designite per la *detection* dello *smell* è stato definito come segue:

function DESIGNITE-UUA-DETECTOR( )
  if il tipo analizzato non ha dipendenze in ingresso **then**
    return l'elemento è una Unutilizied Abstraction
  else
    if L'elemento analizzato ha supertipi **then**
      if Tutti i supertipi non hanno dipendenze in ingresso **then**
        return l'elemento è una Unutilizied Abstraction

Si può notare che non vengono valutati separatamente i due casi *Unreferenced Abstraction* e *Orphan Abstraction* (presentati nella sezione 4.2) a differenza di Arcan che invece effettua controlli differenti a seconda della tipologia di *abstraction* considerata. Non essendoci nessuna distinzione tra tipologia della classe e delle relazioni tra esse, vengono considerati come *smell* tutte le *abstraction* che non presentano dipendenze in ingresso.

Viene effettuato inoltre un altro controllo per stabilire se una classe è inutilizzata o meno, attraverso una valutazione dei supertipi della classe sotto analisi. Se la *abstraction* che stiamo considerando presenta dei supertipi, la presenza dello *smell* è verificata, oltre che dalla eventuale assenza di dipendenze in ingresso, anche dalle dipendenze dei supertipi stessi. Se nessuno di essi ne presenta in ingresso allora anche la classe sotto analisi è considerata *unutilizied*. Più in generale si può affermare che Designite considera come tali tutte le *abstraction* figlie di classi senza dipendenze in ingresso.

Analisi risultati Nella tabella 12 vengono riportati i dati relativi alla *detection* effettuata con i due diversi *tool*. Da questi si evince che le istanze dello *smell* trovate da Designite sono in numero maggiore rispetto ad Arcan, con una differenza di 4120 unità. Un'analisi approfondita del funzionamento dei due *tool* ha dimostrato che la causa principale di questo divario può essere ricercata nel *parsing* degli elementi e nelle strutture dati utilizzate. La verifica è stata effettuata selezionando le *abstraction* individuate come *smell* da Designite ma non considerate tali da parte di Arcan, e analizzando successivamente la struttura del grafo delle dipendenze al fine di valutare le relazioni della classe dal punto di vista di Arcan. È stata riscontrata la presenza di diverse classi nel grafo con archi in ingresso e nessun supertipo, perciò a causa delle relazioni presenti non sono state individuati da Arcan come *unutilizied* e la loro rilevazione da parte di Designite non è causata del controllo sui supertipi. Di conseguenza si può concludere che con molta probabilità sia presente una differenza tra le strutture dati e gli algoritmi di *parsing* utilizzati, con le dipendenze che vengono identificate in maniera diversa causando un grande divario di istanze dello *smell* tra i due *tool*.

Inoltre un altro fattore che potrebbe influenzare il numero delle istanze è la considerazione come *smell* da parte di Designite di tutti i figli di classi inutilizzate indistintamente dal loro utilizzo o meno. Se queste ultime fossero utilizzate, non verrebbero infatti considerate da Arcan come problema, creando una discrepanza nei risultati.

Progetto	**N° smell Arcan**	**N° smell Designite**
Accumulo	177	1955
Beam	32	22
Bookkeeper	189	906
Cassandra	137	1927
Druid	128	203
Flink	185	431
Geode	304	1557
Kafka	134	1118
Skywalking	683	1182
Zookeeper	64	336
Totale	2033	6153

Table 12: Confronto istanze Unutilizied Abstraction

5.4.3 Unnecessary Abstraction

Attraverso il *tool* Designite è possibile anche la *detection* dello *smell Unnecessary Abstraction*, sebbene con la limitazione di identificare solamente il caso *constant placeholder*, a differenza di Arcan che individua anche *procedural class* e le *over engineered*. Questa diversità tra le capacità dei due *smell* crea un grande divario anche per quanto riguarda il numero di istanze, con Arcan che ne ha individuate 2051 in più.

Il confronto delle strategie di identificazione sarà effettuato analizzando solamente il caso *constant placeholder* mentre l'analisi dei risultati verrà condotta esaminando anche tutti i casi presenti. Nello specifico, saranno inizialmente considerati tutti i casi di *Unnecessary Abstraction*, per poi proseguire con l'analisi solamente di *constant placeholder*.

Strategie di identificazione Per la ricerca delle *abstraction constant placeholder* gli sviluppatori di Designite hanno effettuato una ricerca di tutte le classi senza metodi e con un numero di attributi minore o uguale a una determinata soglia, che al momento della stesura di questo elaborato assumeva il valore 5. Segue una rappresentazione in pseudocodice dell'algoritmo di *detection*:

```
function DESIGNITE-UNA-DETECTOR( )
    def SOGLIA = 5
    if L'elemento analizzato ha un numero di attributi <= SOGLIA then
        if L'elemento analizzato non ha metodi then
            return l'elemento è una Unnecessary Abstraction
```

Mentre Arcan considera come *constant placeholder* le *abstraction* che presentano solamente attributi costanti a prescindere dal loro numero, Designite introduce una soglia minima di attributi che deve definire una classe per essere considerata una *constant placeholder*, poiché gli sviluppatori del tool devono aver ritenuto che classi con pochi costante non possano essere considerati *placeholder*. Il problema delle classi costanti con pochi attributi è stato riscontrato in Arcan, dove è emerso che la causa principale della presenza di falsi positivi nel caso *constant placeholder* fosse proprio la loro presenza nel sistema.

Un ulteriore differenza tra i due algoritmi si può ricercare nel controllo degli attributi poiché, mentre Arcan verifica che gli siano costanti e con un valore già assegnato, Designite si limita solamente a verificare che la *abstraction* considerata non contenga altro oltre ad essi, effettuando così la *detection* di classi che non hanno tutti valori costanti.

Infine la struttura di Arcan ha reso necessario anche il filtraggio delle classi rappresentanti errori, mentre per Designite non è stato necessario poiché

probabilmente l'introduzione della soglia esclude queste classi che solitamente contengono massimo due attributi costanti.

Analisi risultati Essendo condotta l'analisi nei due modi differenti, nella tabella riepilogativa delle istanze è stata inserita una nuova colonna, denominata *N° constant placeholder*, che indica il numero di istanze di questa tipologia trovate da Arcan nei vari progetti.

Dai dati presentati nella tabella 13 emergono i diversi approcci adottati dai due tool. La differenza di 2051 istanze in favore di Arcan è in soprattutto conseguenza della ricerca delle *procedural class* che ha un forte impatto sul numero totale di *smell* trovati.

Progetto	**N° smell Arcan**	**N° constant placeholder**	**N° smell designite**
Accumulo	1113	24	77
Beam	13	24	2
Bookkeeper	96	15	74
Cassandra	639	4	33
Druid	52	5	7
Flink	47	25	28
Geode	166	19	34
Kafka	119	19	51
Skywalking	208	36	128
Zookeeper	38	7	6
Totale	2491	178	440

Table 13: Confronto istanze Unnecessary Abstraction

Se anche per Arcan non venissero considerati i due casi *over engineered* e *procedural class*, si può notare come Designite sia in grado di trovare un numero maggiore di istanze. Un'analisi dettagliata delle classi individuate da quest'ultimo ha permesso di evidenziare le possibili cause principali che hanno portato a questa discrepanza:

- Il *tool* Designite non effettua il controllo che tutti gli attributi della classe siano costanti e con valori di default assegnati e l'analisi effettuata ha confermato questa differenza, in quanto tra le istanze erano presenti numerose classi che presentavano attributi non costanti.
- C'è una differenza per quanto riguarda gli attributi considerati all'interno delle classi, poiché mentre Arcan effettua l'analisi solamente degli attributi della *abstraction* presa sotto analisi, Designite considera anche

tutti quelli ereditati da eventuali supertipi. Questa diversità porta quindi la *detection* da parte di Designite di classi vuote che sono sottotipi di *costant placeholder*, che non verrebbero invece individuate da Arcan.

5.4.4 Conclusioni

La sezione 5.4 ha illustrato un confronto tra i due *tool* Arcan e Designite, in grado di effettuare la *detection* degli smell *Subclasses Do Not Redefine Methods*, *Unutilizied Abstraction* e *Unnecessary Abstraction*.

Designite ha dimostrato una capacità maggiore di riconoscimento di istanze per gli smell *SR* e *UUA* ma poi ne ha individuato un numero minore per quanto riguarda lo smell *UNA*. Non essendo presenti i valori di *precision* per Designite, non è stato possibile fare un confronto approfondito riguardo questo aspetto degli algoritmi. In generale, è stata constatata sia la presenza di diversi falsi positivi di Arcan non segnalati da Designite sia istanze individuate da Designite che non sono risultate come smell.

Riguardo *Subclasses Do Not Redefine Methods*, Designite è in grado di discriminare i metodi ereditati in base al modificatore mentre Arcan li considera tutti, anche se ritengo un caso abbastanza isolato la presenza in una gerarchia di due metodi *private* definiti da classi diverse che condividono lo stesso nome. Arcan inoltre non considera nella ricerca diversi falsi positivi relativi a *exception*, riconosciuti invece da Designite. Per questi motivi introdotti, ritengo che nessun *tool* prevalga sull'altro in considerazione alla qualità e capacità della *detection*.

In merito a *Unutilizied Abstraction*, Designite non effettua alcuna distinzione tra la tipologia di classi analizzate e le relazioni che intercorrono tra esse, a differenza di quanto effettuato da Arcan. Questa situazione porta quindi a non considerare il caso delle *orphan abstractions*, introdotto nella definizione dello smell come una delle due tipologie di manifestazione (sezione 4.2). Molte interfacce e classi astratte, che dovrebbero risultare come smell, non vengono erroneamente considerate da Designite. Inoltre la ricerca dei figli di classi inutilizzate introduce diverse istanze in più non prese in considerazione da Arcan. Il controllo di Arcan sembra quindi più completo, a fronte di un'analisi maggiormente approfondita su classi e relazioni e una fedeltà superiore alla definizione dello smell rispetto a Designite.

Infine, relativamente a *Unnecessary Abstraction* si può affermare che Arcan effettua un controllo più dettagliato e accurato, poiché vengono ricercati nel codice tutti e tre i casi di classi non necessarie presentati, sacrificando però la *precision* (73.66%) influenzata soprattutto dalle classi procedurali non considerate da Designite. Anche valutando solamente il caso *constant placeholder*, Arcan esegue verifiche più approfondite riguardo le *abstraction* sotto

analisi.

Concludendo, si può affermare che nonostante la ricerca di Designite sia in grado di trovare un maggior numero di istanze, Arcan effettua controlli più dettagliati sulle classi e interfacce sotto analisi. Non disponendo però dei dati di *precision* di tutte e due i *tool*, non può essere effettuato un confronto sulla differenza della reale efficacia dei due algoritmi.

5.5 Problemi riscontrati

Durante lo sviluppo delle nuove strutture e algoritmi di *detection* sono stati affrontati diversi problemi, riguardanti sia lo studio di soluzioni adeguate ai problemi sia l'effettiva implementazione e correzione degli errori.

L'attività di validazione del lavoro svolto è stata l'attività più complessa e impegnativa poiché i controlli effettuati sui differenti progetti hanno comportato modifiche alle strutture e algoritmi implementati in precedenza. L'analisi approfondita delle diverse classi individuate ha rivelato la presenza di diversi falsi positivi, causati da casi particolari che non erano stati considerati e da situazioni che non era ben chiaro se fossero incluse nelle definizioni degli *smell* o meno. Per alcuni di questi casi particolari è stata sufficiente la modifica degli algoritmi di *detection* e *parsing* oppure della struttura del grafo delle dipendenze, per migliorare la precisione e filtrare i casi positivi. Per la soluzione di altre situazioni invece è stato necessario, prima di procedere con un'eventuale modifica del software, un confronto con la mia tutor, per comprendere quale fosse il metodo migliore per la loro gestione.

Per gli *smell Subclasses Do Not Redefine Methods* e *Unnecessary Abstraction* le modifiche apportate hanno inoltre comportato lo svolgimento di una seconda fase di validazione, per definire i nuovi valori di *precision* degli algoritmi.

5.6 Osservazioni finali

In conclusione, rispetto agli *architectural smell* di cui è stata sviluppata la *detection* e dopo aver eseguito Arcan su dieci differenti sistemi *open source*, Arcan presenta una *precision* riguardo *Subclasses Do Not Redefine Methods, Unutilizied Abstraction e Unnecessary Abstraction* rispettivamente del 89.75%, 95.43% e 73.66%.

6 Conclusioni e sviluppi futuri

Questo elaborato ha presentato il mio lavoro svolto per l'estensione del tool Arcan [11], riguardante l'implementazione degli algoritmi di *detection* per l'analisi di tre nuovi *architectural smells*, di cui uno relativo a problemi di gerarchia (*Subclasses Do Not Redefine Methods*) e due riguardanti *smell* che violano il principio di astrazione (*Unutilizied Abstraction* e *Unnecessary Abstraction).* Oltre agli *smell* e alle strategie di identificazione e algoritmi ideati, sono state presentate anche le modifiche effettuate al *parsing* e al grafo delle dipendenze, necessarie per la rappresentazione di informazioni riguardanti funzioni e attributi richiesti dai nuovi *smell.*

È stata analizzata nel dettaglio la fase di validazione degli algoritmi, con analisi dei falsi positivi riscontrati e del valore di *precision* degli stessi. In particolare, è stato calcolato un valore di questa metrica pari a 89.75% per quanto riguarda *Subclasses Do Not Redefine Methods*, mentre *Unutilizied Abstraction* e *Unnecessary Abstraction* hanno registrato una *precision* rispettivamente di 95.43% e 73.66%.

È presente infine una sezione riguardante il confronto tra Arcan e Designite, in grado di riconoscere tutti e tre gli *smell* introdotti in questo elaborato, che ha visto un'analisi approfondita delle differenze riguardanti le strategie di detection adottate dai *tool* e i risultati ottenuti.

La struttura di Arcan lo rende molto flessibile dal punto di vista dell'aggiunta di nuove caratteristiche, come può essere per nuovi algoritmi di *detection*, *parsing* di nuovi linguaggi e modifiche al grafo delle dipendenze, e potrà continuare quindi ad essere esteso per ampliare il suo raggio di azione e le sue capacità.

Ulteriore lavoro invece può essere svolto riguardo agli *smell* da me introdotti. Sarebbe possibile infatti studiare nuovi filtri per la rimozione di casi di falsa positività con conseguente aumento della *precision* degli algoritmi. Inoltre potrebbero essere introdotte nuove casistiche nell'ambito di *Unnecessary Abstraction*, previo studio di nuovi problemi che possono introdurre classi non necessarie al design.

Un'ulteriore modifica ad Arcan potrebbe essere l'aggiunta del *refactoring* automatico di alcuni *smell* trovati, proponendo all'utente la soluzione più congeniale per la risoluzione dello stesso. Nel caso di progetti Java infatti, grazie alle librerie utilizzate è possibile la modifica di tutti gli elementi che compongono un programma. Questa possibilità di *refactoring* automatico può adattarsi molto bene ad esempio nei casi *over-engineered* di *Unnecessary Abstraction.*

Bibliografia

[1] Hugo Andrade, Eduardo Almeida, and Ivica Crnkovic. Architectural bad smells in software product lines: An exploratory study. 04 2014.

[2] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. Are architectural smells independent from code smells? an empirical study. *Journal of Systems and Software*, 154, 04 2019.

[3] U. Azadi, F. Arcelli Fontana, and D. Taibi. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97, 2019.

[4] Carliss Baldwin and Kim Clark. *Design Rules Volume I: The Power of Modularity*, volume 1. 01 2000.

[5] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[6] A. Biaggi, F. Arcelli Fontana, and R. Roveda. An architectural smells detection tool for c and c++ projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 417–420, 2018.

[7] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

[8] K Brown and B Whitenack. Pattern languages of program design, vol. 2. *Reading, MA: Addisson-Wesley*, 1996.

[9] Jens Dietrich. Upload your program, share your model. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 21–22, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Vincenzo Ferme, Francesca Arcelli Fontana, and Marco Zanoni. Toward assessing software architecture quality by exploiting code smell relations. 05 2015.

[11] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285, 2017.

[12] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[14] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.

[15] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, 2009.

[16] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 235–245, 2015.

[17] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 176–17609, 2018.

[18] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[19] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 277–286, 2012.

[20] David Martin and Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[21] Robert C. Martin. Oo design quality metrics. *An analysis of dependencies*, 12(1):151–170, 1994.

[22] Robert C. Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[23] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[24] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60, 2015.

[25] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. 46:1155–1179, 2015.

[26] D. T. Ross, J. B. Goodenough, and C. A. Irvine. Software engineering: Process, principles, and goals. *Computer*, 8(5):17–27, 1975.

[27] Alessandra Rota. *Identificazione di architectural smells legati alle gerarchie di sistemi object oriented.* PhD thesis, Università degli Studi Milano Bicocca, 2020.

[28] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, BRIDGE '16, page 1–4, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Michael Stal. Refactoring software architectures. In *Agile Software Architecture*, pages 63–82. Elsevier, 2014.

[30] Beatrice Stropeni. *Identificazione di smell architetturali in progetti Python.* PhD thesis, Università degli Studi Milano Bicocca, 2020.

[31] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. Chapter 3 - abstraction smells. In Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma, editors, *Refactoring for Software Design Smells*, pages 21 – 60. Morgan Kaufmann, Boston, 2015.

[32] 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*, volume 1, pages 403–414, 2015.

[33] A. von Zitzewitz. Mitigating technical and architectural debt with sonargraph. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 66–67, 2019.

Sitografia

[34] Accumulo v2.0.0. https://github.com/apache/accumulo/tree/rel/2.0.0. Ultimo accesso 08/2020.

[35] Beam v2.20.0. https://github.com/apache/beam. Ultimo accesso 08/2020.

[36] Bookkeeper v4.10.0. https://github.com/apache/bookkeeper/tree/release-4.10.0. Ultimo accesso 08/2020.

[37] Cassandra v3.11.6. https://github.com/apache/cassandra. Ultimo accesso 08/2020.

[38] Designite GitHub repository. https://github.com/tushartushar/DesigniteJava.

[39] Druid v0.18.0. https://github.com/apache/druid. Ultimo accesso 08/2020.

[40] Flink v1.10.0. https://github.com/apache/flink/tree/release-1.10. Ultimo accesso 08/2020.

[41] Geode v1.12.0. https://github.com/apache/geode. Ultimo accesso 08/2020.

[42] Github. https://github.com.

[43] Jsoniter. http://jsoniter.com/java-features.html.

[44] Junit 5. https://junit.org/junit5/.

[45] Kafka v2.5.0. https://github.com/apache/kafka/tree/2.5.0. Ultimo accesso 08/2020.

[46] Skywalking v7.0.0. https://github.com/apache/skywalking/tree/v7.0.0. Ultimo accesso 08/2020.

[47] Zookeeper v3.6.0. https://github.com/apache/zookeeper/tree/release-3.6.0. Ultimo accesso 08/2020.

[48] Apache Software Foundation. Apache Tinkerpop. https://tinkerpop.apache.org. Ultimo accesso 09/2020.

[49] Apache Software Foundation. Apache website. https://www.apache.org/. Ultimo accesso 09/2020.

[50] Bugan IT Consulting UG. STAN - Static Analizer for Java. http://stan4j.com/. Ultimo accesso 07/2020.

[51] ESSeRE Lab. Evolution of Software SystEms and Reverse Engineering Lab. https://essere.disco.unimib.it. Ultimo accesso 09/2020.

[52] Martin Fowler. You Aren't Gonna Need It (YAGNI) Principle. https://www.martinfowler.com/bliki/Yagni.html#footnote-origin. Ultimo accesso 09/2020.

[53] Headway Software Technologies Ltd. Structure101. https://structure101.com/help/java/studio/#intro/welcome-studio.html%3FTocPath%3DStudio%7C_____0.

[54] hello2morrow. Sonargraph. https://www.hello2morrow.com/products/sonargraph. Ultimo accesso 07/2020.

[55] Jetbrains. Intellij IDEA. https://www.jetbrains.com/idea/.

[56] Logarix. AI Reviewer. http://www.aireviewer.com/doc/. Ultimo accesso 07/2020.

[57] Neo Technology, Inc. Cypher Query Language. https://neo4j.com/developer/cypher/.

[58] Neo Technology, Inc. Neo 4J Graph Platform. https://neo4j.com.

[59] Sun Microsystems. Java Documentation. https://docs.oracle.com/javase/7/docs/api/. Ultimo accesso 09/2020.

Ringraziamenti

Ritengo doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione dello stesso.

In primo luogo ringrazio il mio relatore Prof.ssa Francesca Arcelli per i suoi consigli fondamentali, la sua infinita disponibilità e soprattutto per avermi dato questa opportunità. Un ringraziamento particolare va anche alla mia co-relatrice Dott.ssa Ilaria Pigazzini, che sin dall'inizio della mia attività di tirocinio è stata disposta ad aiutarmi guidandomi attraverso l'implementazione dell'applicativo e la stesura dell'elaborato.

Ringrazio infinitamente la mia famiglia per aver costantemente sostenuto le mie scelte e per aver creduto in me. Dedico un grazie speciale a mia mamma Giovanna per la sua dolcezza che mi è sempre stata di conforto. A mio papà Angelo, che con la sua simpatia contagiosa rende questa casa più allegra. A mio fratello Daniele, disponibile ad aiutarmi in qualsiasi momento e situazione. Vi voglio bene.

Voglio ringraziare anche Giorgia, supporto indispensabile durante tutto il percorso universitario, pronta ogni volta ad ascoltarmi e spronarmi. Grazie per essere al mio fianco.

Non posso non menzionare i miei amici di una vita, compagni di viaggio e presenze costanti anche nelle difficoltà. So di poter sempre contare su di voi. Rivolgo anche un ringraziamento ai miei colleghi di università, "i fighi dell'U4", per aver condiviso questo percorso e trascorso tante giornate a fare molte cose, escluso studiare. Conserverò questi ricordi con molta gioia.

Inoltre dedico questa tesi a me stesso, ai miei sacrifici, al mio impegno e determinazione, che mi hanno permesso di arrivare fin qui e spero mi porteranno lontano.

Infine un ultimo pensiero lo rivolgo a te, Lore. Non sarai mai dimenticato.