UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

ANALISI EVOLUTIVA DEI DESIGN PATTERN E CODE SMELLS IN SOFTWARE OPEN SOURCE

RELATORE

Prof. **Fabio Palomba**

SECONDO RELATORE

Dott. **Giammaria Giordano**

Università degli studi di Salerno

CANDIDATO

Aurelio Sepe

Matricola: 0512108638

Anno Accademico 2021-2022

Sommario

I design pattern sono da sempre associati ad una serie di benefici, come manutenibilità, flessibilità e estendibilità [1], nonostante ciò diversi studi in letteratura [16][2] hanno verificato che, con il passare del tempo, i design pattern tendono ad introdurre complessità ed errori all'interno del software, questo fenomeno di conseguenza può portare allo sviluppo di anti-pattern, detti anche code smells, che rappresentano difetti di qualità all'interno del codice.

L'obiettivo di questa tesi è di verificare se i design pattern sono correlati allo sviluppo di code smells .

Per raggiungere questo obiettivo è stata studiata l'evoluzione di progetti software open-source contenenti design pattern, tramite l'utilizzo di tools per il mining di repository GitHub è stato possibile ottenere una visione molto granulare dei cambiamenti all'interno dei software analizzati.

una volta individuata la storia dei cambiamenti dei progetti open source è stata condotta un analisi statistica per comprendere come i design pattern abbiano influenzato la presenza di code smells all'interno dei progetti.

Al termine dell'analisi è stato riscontrato che non è chiaro se i design pattern causino code smells, però è stato riscontrato che in alcuni casi sono in grado di prevenirne la presenza.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto Applicativo

1.1.1 Design Pattern

Il concetto di design pattern in informatica è stato per la prima volta introdotto nel famoso libro "Design Patterns: Elements of Reusable Object-Oriented Software" [7], in cui sono descritti come uno schema di classi e oggetti comunicanti che permettono di fornire una soluzione generale a un problema ricorrente.

Ogni design pattern è composto da:

- Nome: che identifica il pattern e sia quanto più rappresentativo dello stesso;
- Il Problema: ovvero la situazione in cui è possibile applicare il pattern;
- La Soluzione: ovvero gli elementi che costituiscono il pattern e le loro relazioni;
- Le Conseguenze: ovvero gli effetti sia positivi che negativi che derivano dall'uso del pattern.

I design pattern sono divisi in 3 categorie, ovvero:

- Pattern Strutturali:
  Pattern che si occupano di risolvere problemi riguardanti la struttura delle classi e degli oggetti;

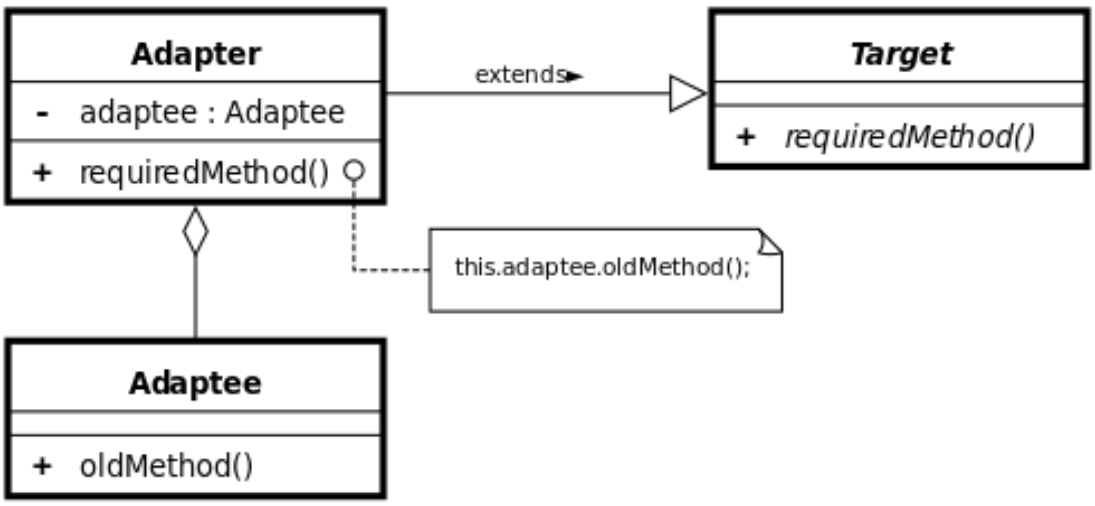


Figura 1.1: Struttura dell'Adapter

- Pattern Creazionali:
  Pattern che si occupano di risolvere problemi riguardanti l'istanziazione degli oggetti;
- Pattern Comportamentali:
  Pattern che si occupano di risolvere problemi riguardanti l'interazione fra oggetti.

Un esempio pratico di design pattern è l'Adapter, è uno dei primi design pattern strutturali ad essere descritti nel libro [7].
L'Adapter, detto anche Wrapper, è utilizzato nelle situazioni in cui in un progetto software è necessario l'utilizzo di librerie che presentano un interfaccia non perfettamente compatibile con quella richiesta, di conseguenza l'Adapter entra in gioco creando una classe che fa da tramite fra l'interfaccia della libreria con l'interfaccia desiderata.

1.1.2 Code Smells

I code smells sono delle tipologie di anti-pattern, ovvero uno schema ricorrente all'interno di classi o metodi che generalmente porta a conseguenze negative, nel caso dei code smells essi portano ad un peggioramento della qualità del codice.

Un esempio pratico può essere il code smell Long Method [6], ovvero quando all'interno di una classe è presente un metodo eccessivamente lungo, questo è considerato negativo poiché porta ad avere un metodo con molte responsabilità e difficilmente leggibile.

1.2 Motivazioni e Obiettivi

La maggiore manutenibilità, flessibilità e estendibilità del codice sono fra i benefici più comuni ad essere citati come vantaggi dell'utilizzo dei design pattern [7].
Nonostante questo lo studio della letteratura ha mostrato come, all'evolversi del progetto in cui sono presenti, le classi appartenenti ai design pattern tendono ad essere proprio quelle più soggette a cambiamenti ed errori, in contrasto con quello che suggerisce la teoria.
L'obiettivo è quindi quello di verificare se l'uso dei design pattern non sia in realtà controproducente, e che quindi il degradamento progressivo dei design pattern non li porti a far emergere code smells all'interno del codice.
Per poter ottenere risultati significativi sarà necessario analizzare un numero elevato di campioni, di conseguenza un altro obiettivo di questa tesi è quello di automatizzare quanto più possibile il processo di raccolta dati, creando un sistema basato sul repository mining per la raccolta di dati relative alle release di progetti software presenti su GitHub.

1.3 Risultati

Lo studio effettuato ha portato alla creazione di un sistema automatizzato di raccolta dati basato sul repository mining, tramite questo sistema è possibile estrarre i dati relativi ai design pattern e code smells di repository presenti su GitHub.
Inoltre grazie ai dati raccolti è stato possibile effettuare un analisi per poter comprendere la relazione fra design pattern e code smells.
Dall'analisi è risultato che non è chiaro se i design pattern causino code smells, però è stato riscontrato che in alcuni casi sono in grado di prevenirne la presenza.

1.4 Struttura della tesi

- **Introduzione**:
  In questo capitolo è presente una breve spiegazione di cosa sono i design pattern e code smells, vengono spiegate le motivazioni e gli obbiettivi che hanno spinto la realizzazione di questa tesi e infine è presente un breve riassunto dei risultati ottenuti;

- **Stato dell'arte**:
  In questo capitolo sono descritti degli studi in letteratura che hanno studiato i possibili effetti negativi dell'utilizzo dei design pattern;

- **Metodologia**:
  In questo capitolo viene descritto in che modo è stato portato avanti il lavoro di analisi, in particolare viene descritto il processo utilizzato per la raccolta dei dati e il modo in cui sono state condotte delle analisi statistiche su di essi;

- **Risultati**:
  In questo capitolo vengono descritti i risultati ottenuti dalle analisi statistiche effettuate;

- **Conclusioni**:
  In questo capitolo vengono descritte le conseguenze dei risultati ottenuti e i possibili studi futuri.

CAPITOLO 2

Stato dell'arte

Questo capitolo illustra come precedenti studi abbiano analizzato la correlazione fra design pattern e la qualità dei software che li contengono

2.1 Introduzione

I design pattern che promuovono una maggiore estensibilità, flessibilità e leggibilità del software [7], in linea teorica dovrebbero risultare meno propensi all'essere modificati e a contenere errori.
Diversi lavori sono stati svolti per verificare empiricamente se i design pattern effettivamente rispettano queste proprietà [16][2].

Nel corso di questi studi sono stati analizzati diversi tipi di software per poter osservare in che modo le classi utilizzate in design pattern tendessero a essere modificate e/o corrette.

2.2 Metriche studiate in letteratura

Per poter valutare l'effetto dei design pattern, una soluzione comune nella letteratura è stata quella di considerare due o più versioni del software analizzato ed estrarre da esse alcune metriche per comprendere il tipo di cambiamento avvenuto fra le versioni[16].
Per poter valutare come i design pattern abbiano influito sulla modificabilità del codice

è stata spesso utilizzata la change proneness, ovvero la propensione delle classi ad essere modificate. è misurata contando per ogni classe il numero di cambiamenti effettuati fra le due versioni[2][9].
Per quanto riguarda la presenza di errori nel codice è stata utilizzata la fault proneness, in maniera simile alla change proneness rappresenta la propensione delle classi a contenere errori, anche essa è misurata in base al numero di errori riscontrati all'interno della classe.

2.2.1 Change proneness

Come detto precedentemente, uno dei vantaggi dei design pattern è quello di permettere una maggiore flessibilità e estendibilità del codice, di conseguenza ci si aspetta che le classi che facciano parte di design pattern abbiano una bassa change proneness.

Lo studio svolto da Bieman et. al. [2] sembra però suggerire il contrario, durante questo studio sono state analizzate due versioni di un software industriale scritto in C++, la prima versione corrispondeva alla prima release stabile del software, mentre la seconda versione era quella più aggiornata. Durante l'analisi gli autori hanno riscontrato che le classi appartenenti a design pattern erano quelle più propense ad essere modificate, anche normalizzando i risultati per la lunghezza delle classi la conclusione rimaneva la stessa, hanno dunque concluso dicendo che contrariamente a quello che è l'immaginario comune i design pattern sono associati ad una maggiore change proneness rispetto alle classi più semplici.
Successivamente venne svolto uno studio più approfondito [3] considerando altri 4 software di utilizzo industriale scritti in Java, venne sempre osservato che le classi coinvolte in design pattern avessero una densità di cambiamento (cambiamenti / lunghezza della classe) più elevata rispetto alle altre classi

Il lavoro svolto nel 2009 da Gatrell et. al. [9] riguardante l'analisi di un software commerciale proprietario scritto in C# ha avuto risultati simili. Infatti le classi che hanno partecipato alla formazione di design pattern sono risultate più propense ad essere modificate. Viene però anche fatto notare come questa propensione può dipendere dal fatto che alcuni design pattern vengono utilizzati in posizioni critiche all'interno del software, in particolare i design pattern Singleton e Factory Method sono risultati essere quelli più propensi al cambiamento.

Nell'articolo di Aversano L. [1] è stato studiato il fattore di co-change legato ai design pattern, ovvero come un cambiamento ad una classe porta modifiche anche alle altre, è stato osservato che i design pattern hanno un fattore di co-change particolarmente elevato.

Dall'analisi di questi studi è possibile osservare che sembra esistere una correlazione fra i design pattern e un elevata change proneness, questo sembrerebbe apparentemente essere in contrasto con quello che dovrebbero essere i vantaggi dell'utilizzo dei design pattern, tuttavia questo potrebbe essere dovuto al fatto che le classi appartenenti ai design pattern sono spesso utilizzate nelle situazioni più complesse e che quindi tendono a richiedere una quantità di revisioni e modifiche più elevata rispetto alle altre classi.
Questa ipotesi è confermata dallo studio condotto da B. Rossi et. al. [12] in cui viene osservato che il cambiamento delle classi appartenenti ai design pattern è fortemente influenzato dal ruolo del pattern nel sistema, in particolare i pattern come l'observer che caratterizzano l'architettura del sistema tendono ad essere più frequentemente modificati, in oltre con il passare del tempo tendono a maturare supportando sempre più funzionalità avanzate.

2.2.2 Fault proneness

Anche in questo caso i design pattern dovrebbero teoricamente portare un vantaggio, fornendo una soluzione standard ad una certa classe di problemi, il codice dovrebbe risultare più comprensibile e meno propenso a contenere errori.
Anche in questo caso però alcuni studi risultano essere in contrasto con questa affermazione.

Nello studio svolto da Gatrell et. al. [8] è stato analizzato per 24 mesi un software proprietario commerciale ed è stato osservato che classi appartenenti a design pattern erano marginalmente più propense a contenere fault, secondo gli autori questo era conseguenza del fatto che erano fra le classi più frequentemente modificate.
Nello stesso studio è stato analizzato anche un sottoinsieme di un grosso software commericiale, dopo aver ispezionato manualmente il codice per individuare i design pattern è stato osservato che le classi appartenenti ad essi erano più propense a contenere difetti, in particolare i design pattern Singleton e Adapter.

Così come per la change proneness questo risultato potrebbe dipendere dal contesto in cui i design pattern vengono utilizzati.

Infatti lo studio di M. Vock [15] conferma che il ruolo dato ad un design pattern influenza notevolmente la propensione a contenere errori. In questo studio è stato analizzato un software commerciale scritto in C++ con oltre 500000 LOC[1], è stato osservato che i design pattern Singleton e Observer fossero spesso usati in parti complesse del sistema e di conseguenza avessero un numero di difetti più elevato della media. Invece il codice scritto per il design pattern Abstract Factory è risultato meno complesso o usato in situazioni meno critiche, di conseguenza l'utilizzo del pattern era associato con una quantità minore di difetti. Il design pattern Template Method essendo stato usato sia in situazioni complesse che non è risultato privo di una particolare propensione.

Anche lo studio condotto da M. O. Elish, M Mohammed [5] ha dato risultati simili, dopo aver analizzato 5 software open source ha concluso che le classi appartenenti ai design pattern non avessero una particolare propensione ad avere fault.
E stato però osservato che il tipo di design pattern, che sono divisi in pattern strutturali, comportamentali e creazionali, avesse un effetto sulla propensione nel contenere errori, in particolare classi appartenenti a design pattern strutturali risultano meno propense a contenere errori.

2.3 Tools per Pattern Detection

Per poter verificare la presenza di design pattern la maggior parte degli studi in letteratura ha utilizzato tools specializzati per il loro riconoscimento.

Uno dei tool più utilizzati è quello sviluppato da N. Tsantalis et. al. [13], l'algoritmo che permette il riconoscimento dei design pattern si basa su similarity scoring (punteggio di pertinenza) fra vertici di grafi. Sia la struttura del codice (gerarchie, dipendenze, etc) da analizzare sia la struttura dei design pattern viene prima rappresentata sotto forma di grafi, successivamente sottoinsiemi del codice da analizzare vengono confrontati con i design pattern e se la loro similarità supera una certa soglia allora il tool riporta di aver individuato il pattern.
Il tool è stato testato su tre software di diverse dimensioni ed ha ottenuto una precision del

[1] Linee di codice

100% ed un recall superiore al 90%
Il vantaggio principale di questo approccio è la possibilità di individuare anche versioni modificate dei pattern.
Il tool però ha alcune limitazioni, infatti è utilizzabile solo su software scritti in Java, inoltre per ottenere la rappresentazione della struttura del codice si basa sull'analisi del bytecode, di conseguenza può essere utilizzato solo dopo aver compilato il codice sorgente del software da analizzare.

Un altro tool utilizzato è DeMima [10], è un tool multilayer che si basa su creare un astrazione del codice sorgente, da cui successivamente verranno identificati i design pattern.
è stato testato su 33 software industriali con una recall del 100% e precision del 33%.

Infine è spesso utilizzato FINDER [4], tool in grado di individuare 22 su 23 design pattern della Gang of Four in progetti software scritti in java.

2.4 Possibili Miglioramenti

Un problema comune fra tutte le ricerche citate è la mancanza di un grosso campione d'analisi, la maggior parte delle ricerche si è limitata ad analizzare 2-3 progetti software e per ognuno di questi circa 5-6 release, dunque un possibile miglioramento è quello di ampliare il numero di progetti da tenere in considerazione.
Per sopperire a questa mancanza in questa tesi è stato sviluppato un sistema automatizzato per la raccolta dati dalle repository presenti su GitHub.

CAPITOLO 3

Metodologia di Ricerca

L'obiettivo di questa tesi è verificare l'impatto dei design pattern sulla qualità dei progetti software open source. A tale scopo è stata effettuata un analisi evolutiva per stabilire come i design pattern, col passare del tempo, tendono ad influenzare la qualità del progetto.

3.1 Struttura del capitolo

Il capitolo è diviso in 3 parti

- **Ipotesi e domande della ricerca**:
  in questa sezione vengono descritte le motivazioni che spingono questa ricerca e le varie domande a cui questa tesi vuole rispondere;
- **Dati ricavati**:
  In questa sezione vengono spiegati i criteri con cui sono stati scelti i progetti open source analizzati e il processo con cui sono stati ricavati i dati necessari da essi;
- **Analisi dei dati**:
  in questa sezione viene spiegato come i dati ricavati sono stati utilizzati per rispondere alle domande di ricerca.

3.2 Ipotesi e Domande della ricerca

Lo studio dello stato dell'arte, nel capitolo 2, ha mostrato come le classi appartenenti ai design pattern tendono ad essere proprio quelle più soggette a cambiamenti ed errori, in contrasto con quello che suggerisce la teoria.

La presenza di numerosi cambiamenti ed errori è comunemente riscontrabile in classi che presentano una grande complessità o che presentano anti-pattern, detti anche code smells, di conseguenza l'analisi proposta in questa tesi si basa sulla seguente ipotesi:

le classi che appartengono a design pattern sono particolarmente propense a sviluppare code smells.

Per poter verificare l'ipotesi sono state formulate due domande di ricerca.

RQ1. *la presenza di design pattern influisce sulla quantità di code smells all'interno di un progetto?*

RQ2. *esistono tipologie di design pattern particolarmente propense a sviluppare code smells?*

3.3 Dati ricavati

Per rispondere alle domande della ricerca è stato necessario ricavare una gran quantità di dati relativi a classi di progetti open source.
I dati sono stati ricavati tramite un processo automatizzato basato sul repository mining di progetti presenti su GitHub.

3.3.1 Requisiti dei progetti analizzati

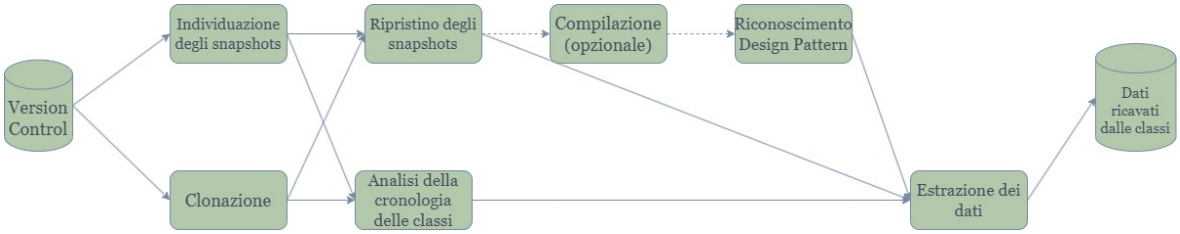
l'analisi è stata limitata a repository GitHub di progetti scritti in linguaggio Java e che utilizzano il tool di build Gradle, la limitazione del linguaggio è dovuta alla scelta del tool relativo all'individuazione dei design pattern [13], che è in grado di funzionare solo sul codice Java, mentre il sistema di build Gradle è stato ritenuto quello più affidabile in fase di compilazione

3.3.2 Criterio scelta delle repository

sono state analizzate 17 repository che rispettavano i requisiti sopracitati, la selezione delle repository è avvenuta in base al numero di stelle presenti su GitHub, che è un buon indicatore della popolarità del progetto

3.3.3 Processo di estrazione dei dati

Figura 3.1: Rappresentazione del processo



Essendo un analisi evolutiva è necessario analizzare un progetto software in vari momenti nel tempo, di conseguenza il processo si basa sull'individuazione di una serie di snapshot tramite un sistema di version control, che per questa ricerca corrisponde a GitHub, e per ognuno di questi ottenere i dati rilevanti all'analisi delle classi appartenenti a design pattern.

Il processo di raccolta dei dati si basa su un insieme di operazioni che vengono eseguite per ogni repository scelta.
dalla Figura 3.1 è possibile vedere in che modo le operazioni dipendono fra di loro

- **Clonazione:**
  In questa operazione viene effettuata la clone della repository;

- **Individuazione degli snapshot:**
  In questa operazione vengono individuati gli snapshot della repository che verranno analizzati

  In questa ricerca sono stati scelti tutti gli snapshot indicati come release dai creatori della repository su GitHub;

- **Ripristino degli snapshot:**
  In questa operazione ogni snapshot precedentemente individuato viene ripristinato a partire dalla repository clonata, il ripristino avviene consultando, tramite il sistema

di version control, le differenze fra lo stato della repository clonata e lo snapshot da ripristinare;

- **Compilazione (opzionale):**
  Operazione opzionale, spesso necessaria per il funzionamento della fase di Pattern Detection

  L'operazione di compilazione è estremamente delicata, poichè spesso può fallire per condizioni non controllabili. Infatti è molto comune trovarsi con un "broken snapshot" [14], ovvero uno snapshot impossibile da compilare in quanto presenta dipendenze non più risolvibili

  La maggior parte dei progetti open source utilizzano tool di build, che permettono di identificare e risolvere automaticamente le dipendenze richieste in fase di compilazione, è fortemente consigliato tenere in considerazione solo repository che si affidano a questi tool.

  Per le problematiche sopracitate solo una parte degli snapshot selezionati sarà poi successivamente utilizzabile per le operazioni successive;

- **Riconoscimento Design Pattern:**
  In questa operazione, per ognuno degli snapshot selezionati, vengono individuati i design pattern presenti nel progetto.
  In questa ricerca è stata rilevata la presenza dei design pattern: Adapter, Bridge, Singleton, Template Method, Proxy, State, Decorator, Factory Method, Component, Observer e Strategy;

- **Analisi della cronologia delle classi:**
  Essendo un analisi evolutiva è necessario, dove possibile, cercare di fornire un identità alla classi analizzate che va oltre il singolo snapshot, infatti spesso tramite operazioni di refractoring le classi tendono a cambiare nome ed essere spostate in moduli diversi, per poter analizzare come una classe sia cambiata nel tempo è necessario tenere traccia di questi cambiamenti.

  In questa operazione, consultando il sistema di version control, viene ricostruita la cronologia delle modifiche per ogni classe presente nel progetto e, per ogni snapshot,

viene creata un associazione fra nomi delle classi e un id che ne rappresenta l'identità fra tutti gli snapshot;

- **Estrazione dei dati:**
  In questa operazione per ogni snapshot vengono estratti i dati necessari per l'analisi.

Vengono estratti per ogni classe:

- Metriche riguardanti la complessità della classe, esempio

ID	NAME	LOC	ELOC	MCCABE	HALV	HALL	HALD	LCC	LCOM	MPC	NOA	NOO	DAC	COH
2085	io.reactivex.internal.observers.BiConsumerSingleObserver	73	36	11	190	55	288.5	0	9	11	0	0	2	0

- Il tipo di design pattern a cui appartiene (se ne esiste uno) e il ruolo in cui svolge in esso, esempio

ID	NAME	PATTERN TYPE	PATTERN ROLE
2085	io.reactivex.internal.observers.BiConsumerSingleObserver	Adapter	adapter

- i code smells che presenta, esempio

ID	NAME	DATASHOULDBEPRIVATE	COMPLEX	FUNCTDEC	GODCLASS	SPAGHETTI
2085	io.reactivex.internal.observers.BiConsumerSingleObserver	False	False	False	False	False

3.4 Analisi dei Dati

3.4.1 RQ1 - evoluzione e co-evoluzione di design pattern e code smells

Per poter rispondere a RQ1 è stata analizzata e messa a confronto l'evoluzione del numero delle classi appartenenti a design pattern con l'evoluzione del numero di classi che presentano code smells. Per ogni progetto è quindi stato calcolato, in corrispondenza di ogni versione, il numero di classi che appartengono a design pattern e il numero di classi che presentano code smells. Infine le due evoluzioni sono state normalizzate tramite il metodo min max e messe a confronto con l'evoluzione del numero di classi all'interno del progetto.

3.4.2 RQ2 - Frequenza di code smells nelle tipologie di design pattern

Per poter verificare se determinate tipologie di design pattern portano ad alcuni tipi di code smells, per ogni progetto analizzato è stato calcolato con che frequenza le varie tipologie di design pattern presentano determinati tipi di code smells

3.4.3 RQ2 - Creazione di un modello statistico

Per poter verificare la correlazione fra design pattern e code smells è stato costruito un modello statistico di regressione lineare. ogni riga del dataset corrisponde ad una classe appartenente ad una specifica versione di uno dei progetti analizzati. Le variabili del modello sono state suddivise nel seguente modo:

- **Variabili indipendenti**
  ogni variabile corrisponde ad una delle tipologie di design pattern individuate nella sezione 3.3.3, ognuna di queste è una variabile booleana che rappresenta se la classe appartiene a quel determinato design pattern.
- **Variabili dipendenti o di risposta**
  ognuna di queste è una variabile booleana che rappresenta se la classe soffre di un tipo di code smell.
  I tipi di code smell considerati sono i seguenti:
  - *Complex Class*
    una classe difficilmente comprensibile e caratterizzata da un alta complessità ciclomatica [6]
  - *Spaghetti Code*
    una classe mal strutturata che presenta metodi molto lunghi e senza parametri [6]
  - *God Class*
    una classe molto grande a cui sono assegnate molte responsabilità e che monopolizza una gran parte delle computazioni del sistema [6]
- **Variabili di controllo**
  come variabili di controllo sono state scelte diverse metriche di qualità del codice, ovvero LOC (numero di linee di codice), MCCABE (complessità ciclomatica), WMC (numero di metodi per classe), NOA (numero di attributi), e LCOM (mancanza di coesione fra i metodi di una classe).

Per ogni variabile dipendente è stato costruito un modello tramite la funzione glm di R.

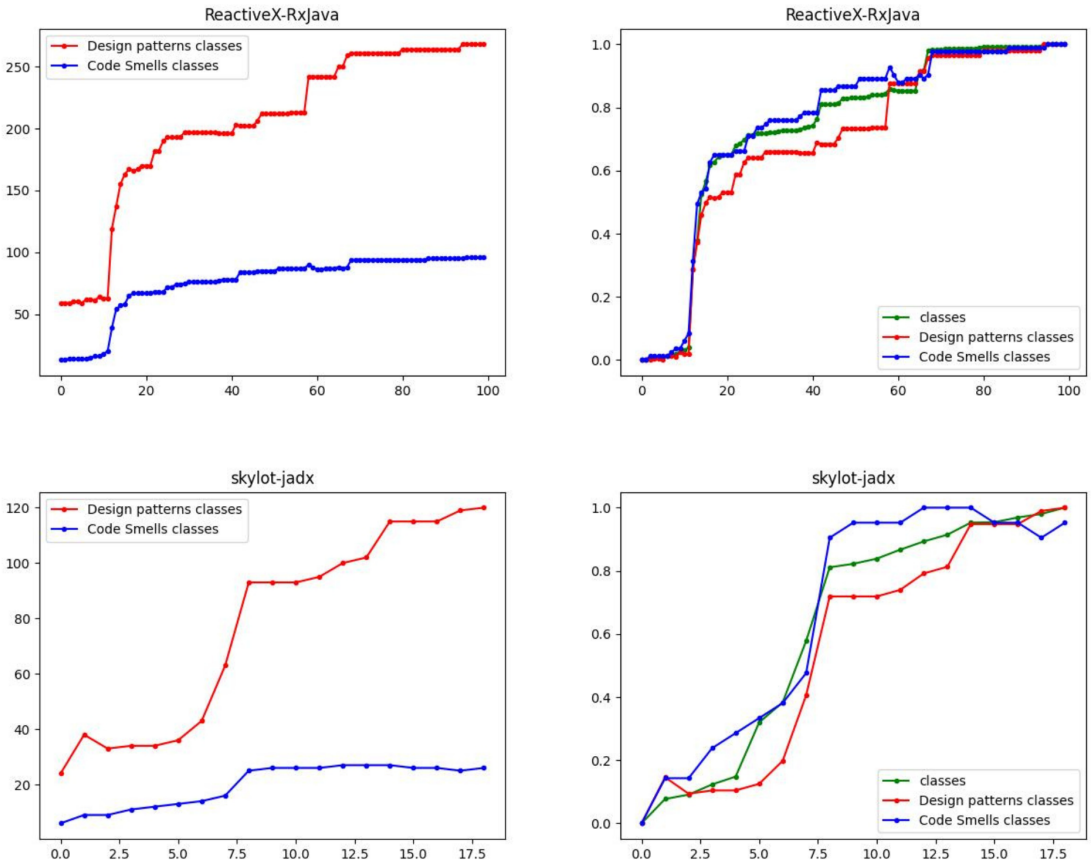
CAPITOLO 4

Risultati

4.1 RQ1 - evoluzione e co-evoluzione di design pattern e code smells

Nella Figura 4.1 è mostrata l'evoluzione del numero di classi, design pattern e code smells. Per brevità sono rappresentati solo due progetti software, è possibile consultare il resto dei grafici nella repository di replicazione del progetto.
Nella colonna di sinistra, è possibile osservare la crescita in valore assoluto del numero di classi appartenenti a design pattern e del numero di classi affette da code smells. Per ogni progetto entrambi gli andamenti tendono ad essere crescenti, inoltre il numero di classi appartenenti a design pattern tende a essere molto superiore rispetto a quello di classi affette da code smells.
L'andamento delle due curve risulta essere simile, questo quindi sembrerebbe avvalorare la tesi che il numero di design pattern influisce sul numero di code smells, è però possibile che entrambe le crescite siano influenzate da un altro fattore, ovvero la crescita del numero di classi totali del progetto.
Nella colonna di destra è possibile vedere la stessa crescita, messa in relazione anche alla crescita del numero totali di classi del progetto, normalizzata tramite min-max nel range [0 1].
Anche in questo caso le tre curve risultano essere molto simili fra di loro, questo potrebbe suggerire che il numero di design pattern e di code smells tende a rimanere, in proporzione

Figura 4.1: Crescita in valore assoluto(sinistra) e normalizzata(destra) dei design pattern e code smells



al numero totale di classi, uguale durante la vita del software.

4.2 RQ2 - Frequenza di code smells nelle tipologie di design pattern

nella Figura 4.2 è mostrata la frequenza con cui le varie tipologie di design pattern tendono ad avere determinati tipi di code smells.

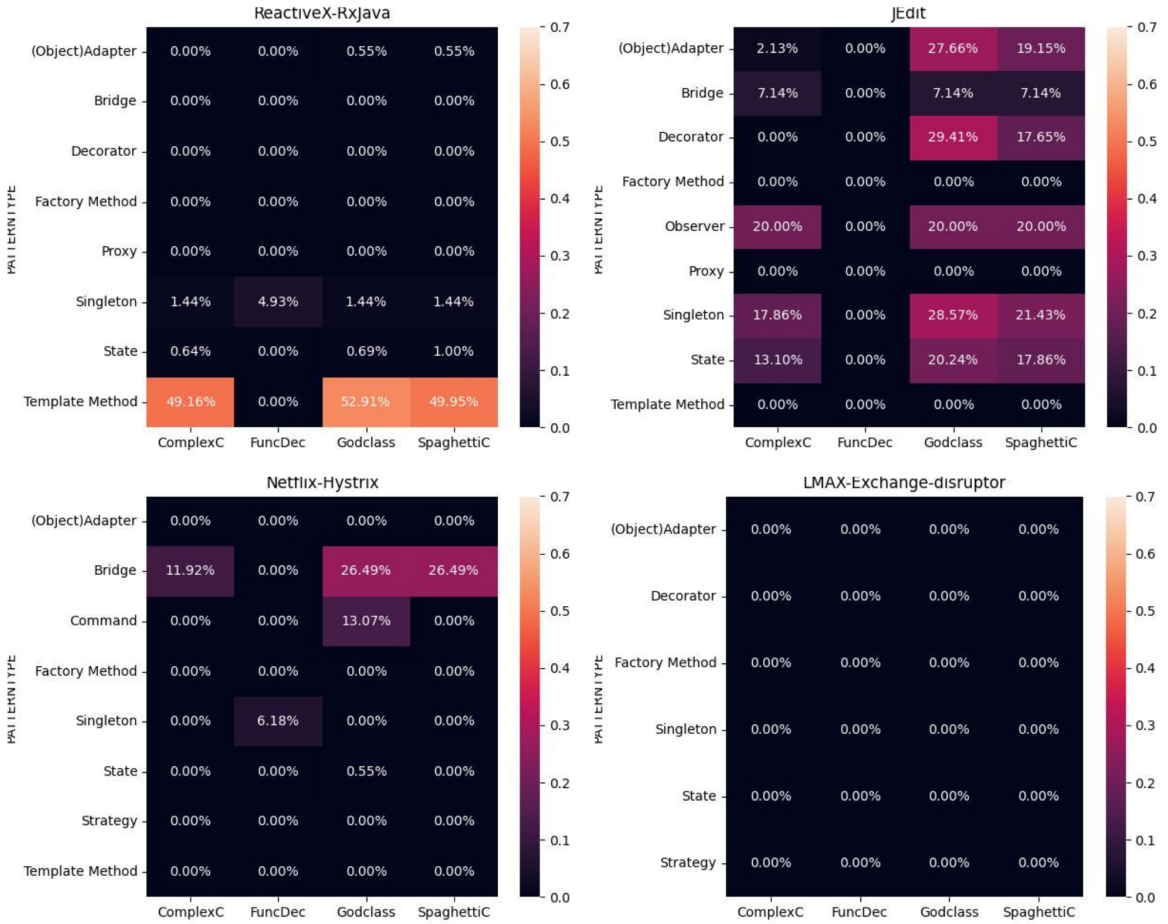
Per brevità sono visibili i grafici di solo 4 progetti software, è possibile consultare il resto dei grafici nella repository di replicazione del progetto.

Dall'analisi completa dei 17 progetti, sono emerse 3 situazioni ricorrenti.

In 10 dei 17 progetti si è verificata la stessa situazione del progetto LMAX-Exchange-distruptor, indicato nel riquadro in basso a destra della Figura 4.2, ovvero nessuna delle classi appartenenti ai design pattern presentava code smells.

In 6 dei 17 progetti si è verificata la stessa situazione del progetto ReactiveX-Rxjava e Netflix-

Figura 4.2: Frequenza di code smells per tipologie di design pattern



Hystrix, indicati rispettivamente nel riquadro in alto a sinistra e in basso a sinistra della Figura 4.2, ovvero per solo una specifica tipologia di design pattern un'alta percentuale delle sue classi erano affette da una o più tipologie di code smells.

Infine solo nel progetto JEdit, indicato nel riquadro in alto a destra della Figura 4.2, si è verificata la situazione in cui la maggior parte delle tipologie di design pattern presentavano un alta percentuale di code smells.

Secondo questo risultato non sembrerebbe esistere una correlazione generale fra tipologie di design pattern e tipologie di code smells, ma che questa possa dipendere dalla natura del progetto analizzato.

4.3 RQ2 - Modello statistico

Nella Tabella 4.1 è possibile vedere i coefficienti associati ad ogni variabile indipendente e di controllo, i coefficienti indicano come l'aumento della variabile associata influenza la

Tabella 4.1: Risultati del modello statistico per ogni smell analizzato

Variabili	Spaghetti Code	Complex Class	God Class
Adapter	-0.384*	8.341	-0.594***
Command	-12.602	166.346	1.544***
Strategy	-10.935	1,086.845	-12.355
Proxy	-10.424	1,008.371	-11.891
Bridge	0.265	17.129	-0.729***
Decorator	-0.808	125.670	-0.189
Observer	1.380	68.508	1.801
Singleton	-2.392**	93.883	1.722***
State	0.349***	-4.897	-0.036
Template Method	-0.146	-16.859	0.586***
Factory Method	1.157**	1,063.142	-11.255
LOC	0.007***	0.014	0.007***
LCOM	-0.0003***	-0.0001	-0.0003***
NOA	0.004***	-0.088	0.034***
WMC	0.033***	-0.008	0.074***
MCCABE	0.002***	7.598	-0.005***
*p < 0.1; **p < 0.05; ***p < 0.01			

variabile dipendente.In questo caso un valore positivo del coefficiente indica che quella variabile è associata ad una maggiore presenza del code smell, mentre uno negativo indica l'opposto.
Gli asterischi indicano quanto sia significativa l'influenza della variabile, di conseguenza è possibile ignorare i coefficienti che non presentano almeno un asterisco poichè essi non influiscono in modo significativo sulla presenza del code smell.
Si può facilmente notare dalla tabella che l'intera colonna relativa al code smell Complex Class non presenta asterischi, nemmeno per le variabili di controllo. Questo probabilmente è dovuto ad una scarsa quantità di istanze del code smell e di conseguenza esso verrà ignorato nell'analisi sottostante.

Partendo dalle variabili di controllo, tutte sono molto significative e in particolare notiamo che le variabili LOC, NOA, WMC e MCCABE sono tutte associate positivamente alla presenza di code smells, questo è un risultato atteso, infatti quanto più queste metriche aumentano quanto più la classe risulta complessa e di conseguenza tende a sviluppare code smells.

Per quanto riguarda le variabili indipendenti, qui sono riportati i risultati per ogni code

smell:

- Spaghetti Code:
  Tra i vari design pattern l'Adapter e il Singleton sembrano portare ad una riduzione della presenza del code smell, mentre State e Factory Method ad un aumento del code smell;

- God Class:
  In questo caso invece a portare una riduzione del code smell sono i pattern Adapter e Bridge, mentre i pattern Command, Singleton e Template Method sembrano essere associati ad una maggiore presenza del code smell.

Da questi risultati sembrerebbe che i pattern che ricadono nella categoria dei pattern strutturali, come il Bridge e l'Adapter, tendono a evitare la presenza dei code smells. Mentre i pattern comportamentali o creazionali, come il Singleton, Tempalte Method, Command e Factory Method, sembrerebbero essere associati ad una maggiore presenza di code smells.

CAPITOLO 5

Conclusioni

5.1 Considerazioni sui risultati ottenuti

Dai risultati ottenuti dall'analisi evolutiva nella sezione 4.1 possiamo concludere che per quanto la crescita di design pattern e code smells sembrerebbe a prima vista correlata, è più probabile che entrambe dipendano semplicemente dall'aumentare delle dimensioni del software e che non ci sia un rapporto di causa-effetto fra le due cose.
Dai risultati ottenuti dalle frequenze di tipologie di design pattern e code smells nella sezione 4.2 possiamo concludere che il tipo di progetto software può influenzare in che modo design pattern e code smells siano correlati.
Infine dai risultati del modello statistico nella sezione 4.3 si può concludere come i pattern strutturali, che promuovono la flessibilità e la manutenibilità del codice, siano più propensi a evitare code smells, mentre pattern comportamentali e creazionali, che sono utilizzati in per implementare funzionalità più complesse, portano l'effetto opposto.

5.2 Possibili miglioramenti e sviluppi futuri

Una grossa limitazione alla raccolta dei dati è stata causata dalla necessità di dover compilare il codice sorgente di ogni progetto prima di poter estrarre da esso i design pattern, questo infatti ha portato a dover scartare un gran numero di repository altrimenti considerate valide da analizzare.

Dunque un chiaro miglioramento sarebbe quello di creare un tool di estrazione di design pattern che non richiede la fase di compilazione, ma che si basi sull'analisi statica del codice sorgente.
Inoltre l'analisi della frequenza fra tipologie di design pattern e code smells, ha rivelato che la tipologia del progetto influisce sui tipi di design pattern e code smells che emergono all'interno del codice, sarebbe quindi interessante rieffettuare l'analisi dividendo le repository in gruppi basati sulla tipologia e area di interesse dei software.

Bibliografia

[1] L. Aversano, G. Canfora, L. Cerulo, C. Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. pages 385–394, 01 2007. (Citato alle pagine i e 7)

[2] J. M. Bieman, D. C. Jain, and H. J. Yang. Oo design patterns, design structure, and program changes: an industrial case study. *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 580–589, 2001. (Citato alle pagine i, 5 e 6)

[3] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander. Design patterns and change proneness: an examination of five evolving systems. *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, pages 40–49, 2003. (Citato a pagina 6)

[4] H. Dabain, A. Manzer, and V. Tzerpos. Design pattern detection using finder. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 1586–1593, New York, NY, USA, 2015. Association for Computing Machinery. (Citato a pagina 9)

[5] M. O. Elish and M. A. Mohammed. Quantitative analysis of fault density in design patterns. *Inf. Softw. Technol.*, 66(C):58–72, oct 2015. (Citato a pagina 8)

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999. (Citato alle pagine 2 e 15)

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. (Citato alle pagine 1, 2, 3 e 5)

[8] M. Gatrell and S. Counsell. Design patterns and fault-proneness a study of commercial c software. pages 1–8, 05 2011. (Citato a pagina 7)

[9] M. Gatrell, S. Counsell, and T. Hall. Design patterns and change proneness: A replication using proprietary c software. In *2009 16th Working Conference on Reverse Engineering*, pages 160–164, 2009. (Citato a pagina 6)

[10] Y.-G. Guéhéneuc and G. Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008. (Citato a pagina 9)

[11] M. D. Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. *2008 IEEE International Conference on Software Maintenance*, pages 217–226, 2008.

[12] B. Rossi and B. Russo. Evolution of design patterns: A replication study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, New York, NY, USA, 2014. Association for Computing Machinery. (Citato a pagina 7)

[13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32:896–909, 12 2006. (Citato alle pagine 8 e 11)

[14] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29:e1838, 01 2016. (Citato a pagina 13)

[15] M. Vokác. Defect frequency and design patterns: an empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30:904–917, 2004. (Citato a pagina 8)

[16] F. Wedyan and S. Abufakher. Impact of design patterns on software quality: A systematic literature review. *IET Software*, 14, 09 2019. (Citato alle pagine i e 5)

Ringraziamenti

Prima di tutto ringrazio il prof. Palomba e il dott. Giordano che mi hanno supportato e guidato con grande disponibilità durante il lavoro di tesi.

Ringrazio i miei genitori, che non mi hanno mai fatto mancare niente.

Ringrazio le mie migliori amiche Sara, Flavia, Alice e Arianna, che mi sono sempre state vicine.

Ringrazio il mio coinquilino Michael, e i miei colleghi Dario, Daniele, Alfonso, Mario, Nicolò e Stefano con cui ho svolto il progetto di Ingegnieria del Software e passato i momenti migliori dell' università.