Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di un'API dichiarativa per il testing di Plugin Gradle

Elaborato in:
Laboratorio di Sistemi Software

Relatore:
Dott.
Danilo Pianini

Presentata da:
Mirko Felice

Sessione I
Anno Accademico 2022-2023

A mia mamma e mio papà, a mio fratello
e a tutta la mia famiglia:
l'unica certezza in questa vita.

Abstract

Le metodologie agili sono diventate uno standard per lo sviluppo di progetti software. Tramite le tecniche e le pratiche a cui esse fanno riferimento è possibile automatizzare gran parte delle procedure legate a tali progetti. L'insieme di queste pratiche viene denominato *DevOps*, un modello con un ciclo di vita circolare che tratta di argomenti chiave come *Build Automation*, *Continuous Integration* e *Continuous Delivery*.

Questa tesi si concentra su una fase cruciale dello sviluppo, il *testing*, e su alcuni strumenti, in particolare Gradle. Ponendo uno specifico contesto, ossia quello riguardante la verifica dei plugin Gradle, si nota però come non esiste una tecnologia che permetta allo sviluppatore una dichiarazione dei test da effettuare in maniera chiara e sistematica. Questo progetto di tesi mira infatti a riempire questo vuoto, proponendo un'API dichiarativa che risolva questo problema.

Viene inoltre effettuata una validazione del software prodotto, effettuando tre rifattorizzazioni a plugin preesistenti, per mostrarne i vantaggi ottenuti.

Indice

Abstract **i**

1 Introduzione **1**

1.1 Contesto e motivazioni 1

1.2 Metodologia agile 3

1.2.1 Testing 6

1.3 DevOps 7

1.4 Automazione 11

1.4.1 Build Automation 11

1.4.2 Continuous Integration 15

1.4.3 Continuous Delivery 17

1.5 Plugin 19

1.5.1 Plugin Testing 21

2 Progetto **23**

2.1 Analisi del dominio 23

2.2 Analisi dei requisiti 24

2.3 Progettazione 25

2.3.1 Scelte progettuali 25

2.3.2 Codebase 30

2.4 Tecnologie utilizzate 39

2.4.1 Kotlin 39

2.4.2 Git 40

2.4.3 Dokka 43

2.4.4 Jackson 43

3 Validazione **45**

3.1 Testing 46

3.1.1 Kotest 46

3.1.2 JUnit 47

3.2 gradle-kotlin-qa 49

3.3 publish-on-central 49

3.4 Template-for-Gradle-Plugins 50

3.5 Risultati 50

Conclusioni **53**

Bibliografia **55**

Ringraziamenti **56**

Elenco delle figure

1.1 Diagramma del rapporto tra quantità di test e qualità ricavata 6
1.2 Correlazione tra le caratteristiche della metodologia DevOps e la qualità del software 9
1.3 Ciclo di vita del modello DevOps 10

2.1 Diagramma UML delle classi del package structure nel modulo core 31
2.2 Diagramma UML delle classi del package checkers nel modulo core 33
2.3 Diagramma UML delle classi del package api nel modulo core 34
2.4 Diagramma UML delle classi del package runners nel modulo core 35
2.5 Diagramma UML delle classi del package tasks nel modulo gradle-plugin 36
2.6 Diagramma UML delle classi del package folder nel modulo gradle-plugin 37
2.7 Diagramma UML delle classi del package test nel modulo gradle-plugin 38

3.1 Grafico a barre riportante la quantità di linee di codice interessate nell'applicazione della libreria 51

Listati di codice

3.1 Codice di esempio della StringSpec. 47
3.2 Codice di una classe di test che utilizza JUnit. 48

Capitolo 1

Introduzione

In questo capitolo vengono trattati gli argomenti su cui si basa il progetto di tesi.

1.1 Contesto e motivazioni

Nel mondo la tecnologia è padrona della vita quotidiana. È penetrata completamente nelle giornate di qualunque essere umano, sia che si tratti di un bambino che si diverte guardando il proprio cartone animato in televisione, sia che si tratti di un anziano che voglia chiamare il nipote tramite il cellulare. Ma, come tutti sanno, l'attualità racconta che la tecnologia non accenna a fermarsi.

La tecnologia è ormai presente in qualsiasi campo, scientifico e non. Si può ritenere con fermezza però che il vantaggio principale che essa offre e che la rende così utilizzata, è certamente l'automazione dei più svariati processi. Ecco dunque il contesto di cui si andrà a trattare. I processi di automazione sono sicuramente lo scopo finale di molti software, i quali permettono al cliente di essere più rapidi. Ad esempio nel momento in cui devono essere memorizzati dati sensibili di un utente, il processo di inserimento dei dati sfruttando un apposito applicativo è decisamente più veloce di una scrittura manuale delle informazioni su un foglio di carta.

Ma, come prevedibile che fosse, la tecnologia permette anche a coloro che debbano sviluppare il software finale di automatizzare i processi interni alla produzione. Gli sviluppatori infatti, cercano il più possibile di velocizzare quei processi che per natura sono ripetitivi, allo scopo di evitare l'impegno delle varie risorse, tra le quali il tempo.

Nello specifico il campo che tratta di queste tecnologie e metodologie viene definito ingegneria del software. Verrà tratta una filosofia che fa parte della disciplina, definita *DevOps*.

Strettamente legato ad essa è presente tutto ciò che riguarda la metodologia agile. Si andrà quindi a trattare di automazione dei processi, adattamento e sviluppo iterativo e incrementale del software. Ad ogni teoria analizzata viene associata una pratica che la rispecchia. Alcune di queste verranno sviscerate, come la *Continuos Integration*, la *Continuos Delivery*, e altro ancora.

Come infarinatura generale, la tesi mira ad apprendere e utilizzare il contesto appena accennato mediante vari strumenti, come *Gradle*. Esso permette, mediante una propria libreria, di effettuare il collaudo di codice sorgente atto a costituire software che garantisca specifici vantaggi per lo strumento stesso. Naturalmente, tutti i dettagli vengono spiegati nei capitoli successivi.

L'obiettivo di questo progetto di tesi è quindi quello di sviluppare una libreria che permetta di facilitare il progresso di altri progetti, dalla prospettiva dell'implementazione delle verifiche da attuare.

La necessità di tale progetto consiste nell'assenza di una modalità semplificata e dichiarativa per effettuare il *testing* dei cosiddetti *plugin Gradle* di cui si andrà a trattare successivamente. La libreria ufficiale proposta da Gradle infatti, permette di scrivere i test in maniera automatizzabile, ma mancando di una modalità puramente dichiarativa. Questo progetto tenta di colmare tale lacuna, assicurando allo sviluppatore un procedimento dichiarativo per controllare i propri plugin.

1.2 Metodologia agile

Innanzitutto è bene esplicare il ciclo di vita del software: in ingegneria del software, viene definito come il processo di sviluppo, collaudo e distribuzione di un prodotto. Esso consiste in più fasi che sono generalmente sempre le stesse e suddivise in questa maniera[1]:

- Analisi:
  - del dominio;
  - dei requisiti;
  - di fattibilità;
  - dei costi;
- Progettazione:
  - architetturale;
  - di dettaglio;
- Implementazione;
- Verifiche e collaudo;
- Installazione;
- Manutenzione.

Queste fasi possono essere gestite in maniere differenti, a seconda della metodologia di sviluppo scelta per il progetto. La distinzione più importante esiste tra il metodo *Waterfall*[2] e la metodologia *Agile*[3].

Nel primo infatti, come suggerisce la traduzione, le fasi avvengono in cascata, ossia in successione l'una all'altra, portando chiaramente sia vantaggi che svantaggi. Il vantaggio più evidente è la semplificazione dal punto di vista

[1] https://archive.is/GYSJe
[2] https://archive.is/1vCpE
[3] https://archive.is/oQLAZ

del controllo dell'andamento del progetto, poiché le fasi sono ben definite fin dall'inizio. Ma uno svantaggio diretto è la difficoltà di stima delle risorse e dei tempi di sviluppo, così come una probabile incompletezza da parte del software nel rispetto dei requisiti dell'utente, il quale potrebbe scoprirli durante lo sviluppo.

Il modello a cascata infatti non prevede in alcun modo di rivisitare una fase del processo di sviluppo, una volta che questa è stata dichiarata come completata [1]. Esso si presta perciò a sistemi in cui l'alterazione del software dopo l'implementazione è pressoché proibita.

È bene però presentare anche una terza variante, definita come modello a spirale, che combina le fasi del modello a cascata in una soluzione iterativa [2]. Sembrerebbe dunque simile al modello agile, ma a differenza di quest'ultimo, il modello a spirale si concentra sui rischi dello sviluppo. Ad ogni iterazione l'obiettivo è maturare una versione del prodotto che sia esente dai rischi riscontrati dal cliente.

Al contrario, la metodologia agile è un insieme di metodi di sviluppo del software fondati su dei principi comuni, direttamente derivati dal noto manifesto[4]. La contrapposizione principale con il modello a cascata consiste in un approccio meno strutturato e più focalizzato sull'obiettivo di consegnare al cliente software funzionante e di qualità, nei tempi più brevi e il più frequentemente possibile. Tra le pratiche più comuni vi sono la formazione di team di sviluppo piccoli ed autosufficienti, lo sviluppo iterativo ed incrementale, la pianificazione adattiva e il coinvolgimento diretto e contiguo del cliente durante l'intero processo di sviluppo.

Il documento principe della metodologia agile è il *Manifesto*, al cui interno sono contenuti i dodici principi [5]:

1. Prioritizzare la soddisfazione del cliente, distribuendo software di valore, fin da subito e in maniera continua.

[4] https://archive.ph/vKNJL
[5] http://agilemanifesto.org/

2. Accogliere i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.

3. Consegnare frequentemente software funzionante, con cadenza variabile tra un paio di settimane e un paio di mesi, preferendo i periodi brevi.

4. Collaborazione quotidiana tra committenti e sviluppatori per tutta la durata del progetto.

5. Fondare i progetti su individui motivati. Dare loro l'ambiente e il supporto di cui hanno bisogno e confidare nella loro capacità di portare a termine il lavoro.

6. Conversare faccia a faccia poiché risulta il modo più efficiente e più efficace per comunicare con il team ed all'interno del team.

7. Utilizzare il software funzionante come il principale metro di misura del progresso.

8. Promuovere tramite i processi agili uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante.

9. Porre continua attenzione all'eccellenza tecnica e alla buona progettazione per esaltare l'agilità.

10. Sia essenziale la semplicità, ovvero l'arte di massimizzare la quantità di lavoro non svolto.

11. Riuscire a far emergere le architetture, i requisiti e la progettazione migliori grazie a team auto-organizzati.

12. A intervalli regolari riflettere su come diventare più efficaci, dopodiché regolare e adattare il proprio comportamento di conseguenza.

1.2.1 Testing

Una fase essenziale del ciclo di vita del software è il *testing*. Tale procedimento consiste nell'individuazione di eventuali incorrettezze, mancanze di funzionalità o inaspettati comportamenti da parte del codice [3].

In letteratura, esistono diversi livelli di verifiche e diverse tipologie di testing. Innanzitutto bisogna distinguere tra il test di unità, il test d'integrazione, il test di sistema e il test di accettazione. Il test di unità mira a verificare il funzionamento di una specifica sezione di codice, solitamente a livello di "funzione". Il test d'integrazione cerca di verificare il funzionamento tra i vari componenti del sistema per comprendere se siano presenti problemi di comunicazione tra questi. Il test di sistema invece vuole collaudare il sistema per intero, controllando quindi che tutti i requisiti siano rispettati. Infine, il test di accettazione riguarda più la parte contrattuale del progetto o in generale gli accordi con il cliente finale.

Esistono poi diversi tipi di verifiche effettuabili. A granularità elevata, si può differenziare tra test funzionali e non-funzionali. I test funzionali si possono a loro volta suddividere tra *white-box* e *black-box*.

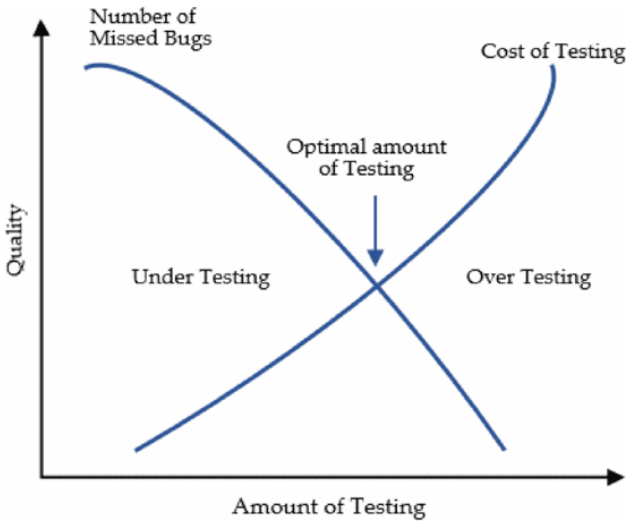

Figura 1.1: *Diagramma del rapporto tra quantità di test e qualità ricavata [4].*

Il testing può essere considerato un'attività fondata sul rischio [4]. Come mostra la figura 1.1, è presente una correlazione tra la quantità dei test e la qualità che se ne ricava. Difatti al crescere del numero dei test, chiaramente la qualità incrementa. Ma allo stesso tempo diminuisce il numero di errori trovati. Si evidenziano perciò due situazioni tipiche: quella di *under testing* e quella di *over testing*, in ciascuna delle quali non si riesce a trovare il giusto equilibrio. La soluzione migliore sta infatti nel mezzo, e bisogna trovarla empiricamente.

Per contestualizzare correttamente l'ambito, nelle metodologie agili, spesso si ricorre ad una tecnica denominata TDD, che sta per *Test Driven Development*. Questa pratica consiste nell'utilizzo dei test d'unità automatizzati allo scopo di controllare la progettazione del software e forzare il processo di dissociazione delle dipendenze. Tramite l'usuale processo di testing, lo sviluppatore spesso trova uno o più errori o difetti, ma il TDD offre un metro di successo chiaro e cristallino quando la verifica non fallisce più, migliorando il livello di confidenza che il sistema rispecchi i propri requisiti. Utilizzando l'approccio TDD un'abbondante quantità di tempo può essere risparmiata che sarebbe altrimenti sprecata sul processo di *debugging*.

1.3 DevOps

Imagine a world where product owners, Development, QA, IT Operations and Infosec work together, not only to help each other, but also to ensure that the overall organization succeeds. By working towards a common goal, they enable the fast flow of planned work into production, while achieving world-class stability, reliability, availability and security.

- Gene Kim, Jez Humble, Patrick Debois, John Willis, John Allspaw [5]

Come affermato dal coniatore stesso, Patrick Debois, non esiste un'unica ufficiale definizione del termine. Esso si è reso popolare attraverso i cosiddetti

DevOps Days, una serie di conferenze iniziate in Belgio nel 2009. Sicuramente, come ribadito da GitHub[6], il termine, combinato dalle parole *Development* e *Operations*, indica più una filosofia di approccio allo sviluppo, che una tecnologia specifica. Essa infatti riguarda un insieme di pratiche che portino ad una consegna del codice rapida e di alta qualità. La metodologia tratta quindi quelle tecniche che consentono agli sviluppatori di accelerare il processo di sviluppo. Esse pertanto, oltre a soddisfare un'esigenza dello sviluppatore, in realtà aiutano ampiamente anche il cliente finale, in quanto l'installazione di una nuova versione del prodotto viene eseguita automaticamente[7].

Uno degli scopi maggiormente legati all'utilizzo di questa metodologia è la qualità. Uno studio ha infatti riportato che è chiaramente presente una correlazione diretta tra le caratteristiche della metodologia e gli attributi che è possibile assegnare al software [6]. In figura 1.2 sono stati riportati tutti i concetti per ciascuno dei quali si è trovata un'associazione.

Si può quindi assumere che la filosofia DevOps possieda un proprio ciclo di vita [7]. Come si osserva in figura 1.3, il ciclo viene raffigurato tramite il simbolo dell'infinito, dal momento che quest'ultimo è significativo nei confronti della filosofia che rappresenta. Per di più in questa maniera è possibile suddividerlo in due parti. Dal lato sinistro sono presenti le fasi che riguardano lo sviluppo vero e proprio, mentre dall'altro lato vi sono le fasi che concernono le operazioni a contatto col cliente finale. Vi sono inoltre delle fasi intermedie giacché possono essere associate contemporaneamente a entrambi i lati.

Di seguito vengono spiegate le sotto-fasi[9].

Plan

La pianificazione è il processo che prevede la dichiarazione del problema e la definizione dell'ambito, allo scopo di identificare le risorse necessarie che sarebbero in uso per tutto il periodo. In secundis, durante

[6] https://resources.github.com/devops
[7] https://archive.is/fnv5s
[8] https://archive.is/N96nQ
[9] https://archive.is/5FzmI

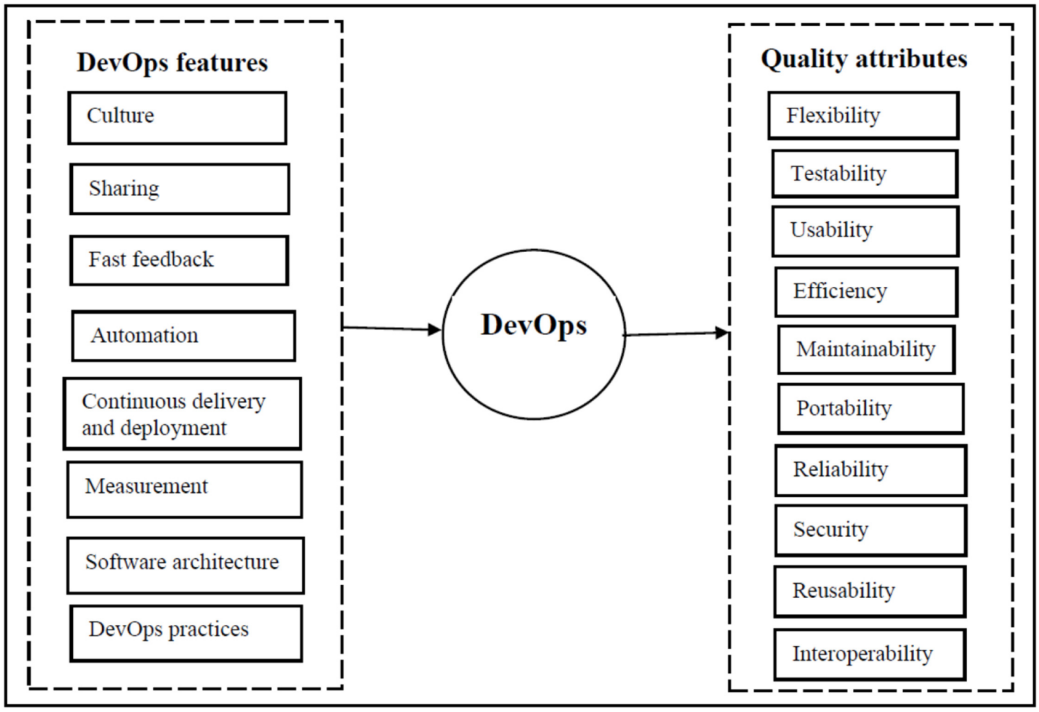

Figura 1.2: *Correlazione tra le caratteristiche della metodologia DevOps e la qualità del software [6].*

questo processo gli sviluppatori esaminano i sistemi più recenti e dichiarano i loro obiettivi. Inoltre, stabiliscono la praticabilità e la fattibilità del progetto in modo da determinare il periodo necessario per realizzarlo. Infine, il team considera le potenziali minacce, i rischi, i vincoli, la sicurezza e le integrazioni necessarie per il sistema, compilando un rapporto di fattibilità per l'intero progetto [7].

Code

Durante questa fase viene implementato il codice per sviluppare le funzionalità pianificate nella fase precedente. Vengono utilizzati strumenti appositi per mantenere lo sviluppo integro.

Build

A questo punto, il codice sviluppato deve essere trasformato negli

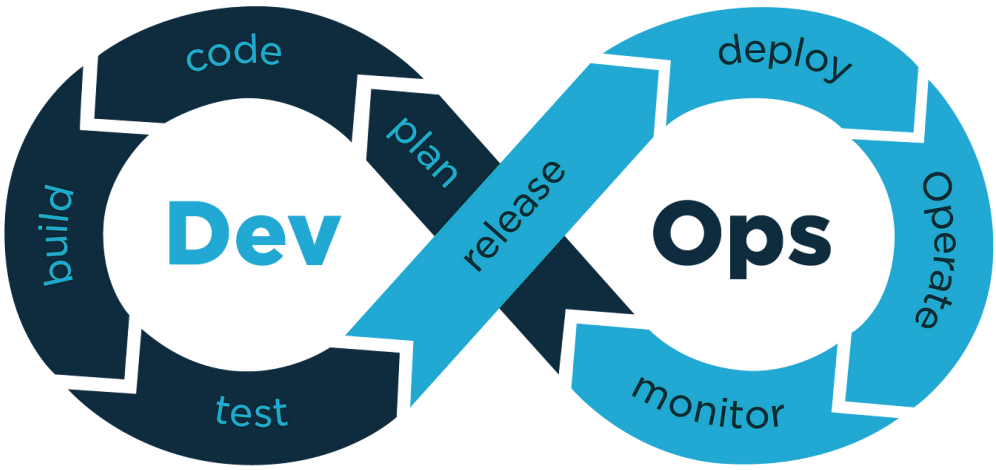

Figura 1.3: *Ciclo di vita del modello DevOps*[8].

artefatti necessari per le prossime fasi. Questa fase viene chiamata costruzione.

Test

Dopo aver elaborato il codice, è ora possibile sottoporlo a tutte le verifiche del caso per controllare che i requisiti del sistema vengano pienamente rispettati.

Release

La fase di rilascio consiste nell'assemblaggio di tutti gli artefatti necessari alla fase di installazione.

Deploy

Questa è la fase in cui il prodotto viene distribuito o rilasciato nell'ambiente di produzione finale che il cliente ha richiesto.

Operate

La fase di operatività consiste nell'esecuzione delle attività per assicurarsi che tutto funzioni perfettamente. In questa fase il team si pre-

occupa di configurare correttamente gli applicativi e di collezionare le informazioni di *feedback* da parte degli utenti finali.

Monitor

Nell'"ultima" fase avviene il monitoraggio continuo delle prestazioni del software e vengono catturate tutte le potenziali problematiche sia dalla prospettiva dell'utente, sia nei confronti del processo di sviluppo.

1.4 Automazione

Seguendo il pensiero della metodologia agile, si tenta di automatizzare tutte le sotto-fasi appena descritte utilizzando appositi strumenti.

1.4.1 Build Automation

Uno strumento fondamentale implementa la cosiddetta *build automation*, che letteralmente tradotta significa automazione della costruzione. Un sistema di automazione della costruzione è in grado di svolgere attività quotidiane di uno sviluppatore come[10]:

- compilazione del codice sorgente in codice binario;
- pacchettizzazione del codice binario;
- esecuzione di test;
- deployment di sistemi di produzione;
- creazione di documentazione e/o note di rilascio.

Una ricerca effettuata tramite sondaggi ha evidenziato come la percentuale di utilizzo degli strumenti di automazione in realtà non è alta come ci si può aspettare [8]. In particolare, nel caso di strumenti di automazione della costruzione, viene mostrato che solo il 18.5% degli utenti che hanno

[10] https://archive.is/8HBIp

avuto esperienza con gli strumenti l'ha avuta per più di 10 anni, mentre una percentuale più alta ne ha per un periodo dai 6 ai 10 anni. Perciò si può dedurre che gli strumenti vengono parzialmente scartati dopo una quantità di tempo pari a 10 anni.

Di seguito vengono elencati alcuni degli strumenti più impiegati.

Bisogna però dapprima specificare cosa sia un DSL, considerato che alcuni di questi li utilizzano. Un *domain-specific language* (DSL), come suggerisce il nome, è un linguaggio di programmazione circoscritto a uno specifico dominio[11]. Esso infatti permette una programmazione limitata rispetto ad un linguaggio tradizionale, in contrasto però ad uno specifico uso. Questo porta dei vantaggi dal punto di vista dell'utente in quanto si riesce ad esprimere in modo idiomatico esattamente ciò che si vuole andare a descrivere. Grazie infatti a degli elevati livelli di astrazione, essi permettono di configurare il dominio di cui trattano in maniera lampante e naturale, come si scoprirà in un secondo momento.

Maven

Maven è uno degli strumenti standard per la gestione di progetti basati su Java. In realtà può essere usato anche per progetti relativi ad altri linguaggi[12]. Esso utilizza un cosiddetto *POM (Project Object Model)*, un file con estensione XML, per comprendere la struttura del progetto. L'obiettivo principale è consentire allo sviluppatore di comprendere lo stato completo dell'impegno nello sviluppo nel più breve periodo di tempo. Per raggiungere questo obiettivo, Maven si occupa di diverse aree di interesse:

- Semplificazione del processo di creazione;
- Fornitura di un sistema di costruzione uniforme;
- Fornitura delle informazioni di progetto di alta qualità;
- Incoraggiamento al miglioramento delle pratiche di sviluppo.

[11] https://archive.ph/GUEHk

[12] https://maven.apache.org/what-is-maven.html

sbt

sbt è uno strumento di build automation dedicato allo sviluppo di progetti Scala[13]. Poiché Scala sfrutta comunque l'ambiente JVM è possibile implementare anche codice Java. Essenzialmente utilizza un DSL simil-Scala che permette di esprimere le dipendenze tra i task in maniera parallela. Inoltre, gli errori di battitura vengono rilevati come errori di compilazione. Un vantaggio enorme che porta lo strumento, riguarda il ciclo di modifica, compilazione e testing. Infatti questo viene notevolmente incrementato in termini di velocità. Grazie alla compilazione incrementale offerta dal compilatore integrato e la possibilità di osservare i file, lo sviluppatore può implementare il codice ed esaminare i risultati in tempo reale, senza avviare ulteriori comandi. Infine, permette anche di essere esteso fornendo ulteriore supporto per nuovi task e piattaforme.

Gradle

Gradle è uno strumento di automazione della costruzione, progettato per essere utilizzato in progetti vasti e modularizzati[14]. La particolarità di Gradle consiste nel combinare le funzionalità già presenti su altri strumenti, come Maven, con un insieme di nuove funzionalità, allo scopo principale di risultare il più incrementale possibile[15]. Questo strumento è infatti nato per gestire la costruzione di un sistema multi-progetto. Gradle è un sistema open-source per l'automazione dello sviluppo fondato sulle idee di Apache Ant e Apache Maven, che introduce un DSL basato su Groovy, al posto della modalità XML usata da Maven per dichiarare la configurazione del progetto. Gradle è stato progettato per sviluppi multi-progetto, infatti supporta gli sviluppi incrementali, in quanto determina in modo intelligente quali parti del *build tree* siano già aggiornate. In questo modo, i processi che dipendono da tali parti non devono essere ri-eseguite, riducendo significativamente il tempo di

[13] https://www.scala-sbt.org/
[14] https://archive.is/2OxVT
[15] https://archive.is/Xgfp1

costruzione del progetto. Gradle supporta anche la costruzione del progetto per processi concorrenti, il che consente di svolgere alcuni compiti (ad esempio i test automatizzati) in modo parallelo su più core o su più CPU.

Di base Gradle non offre molte funzionalità, ma tramite i potenti plugin messi a disposizione permette di essere enormemente flessibile [9]. Un plugin può fare molte cose:

- aggiungere attività al progetto;
- preconfigurare le attività aggiunte con dei valori di default;
- aggiungere configurazioni per le dipendenze del progetto;
- aggiungere nuove proprietà e metodi ai tipi preesistenti mediante *extensions*.

Di seguito vengono riportate alcune differenze con Maven[16]:

- Flessibilità: entrambi gli strumenti forniscono il paradigma noto come *convention over configuration*, il quale tenta di ridurre il numero di scelte che lo sviluppatore debba fare nell'utilizzare il sistema[17]. Deve essere possibile configurare allo stesso modo lo strumento senza però intaccare la flessibilità, basandosi quindi sul principio *DRY, Don't Repeat Yourself*. Ma, a differenza di Maven, Gradle permette una maggiore personalizzazione da questo punto di vista, poiché è stato progettato proprio per utenti responsabili e capaci.
- Prestazioni: entrambi gli strumenti permettono una risoluzione parallela delle dipendenze e in generale della costruzione del progetto. Gradle però risulta molto più veloce di Maven grazie a 3 caratteristiche:
  - Incrementalità: Gradle evita il lavoro non necessario valutando cosa effettivamente debba essere riprocessato.

[16] https://gradle.org/maven-vs-gradle/
[17] https://en.wikipedia.org/wiki/Convention_over_configuration

  - Build Cache: riuso dei file prodotti precedentemente tra le varie build.
  - Daemon: un processo di lunga durata che mantiene in memoria le informazioni più richieste.

- User Experience: Gradle lavora a stretto contatto con gli sviluppatori di ambienti integrati, come IntelliJ IDEA, per supportare il più possibile Gradle, soprattutto ora che quest'ultimo fornisce anche un DSL basato su Kotlin. Inoltre, esso mette a disposizione un'interfaccia grafica interattiva per osservare nel dettaglio tutto ciò che riguarda una build appena eseguita. Grazie infatti ad un plugin da aggiungere come sempre allo strumento, si possono esaminare le cosiddette *build scans*.

- Gestione delle dipendenze: Gradle permette una personalizzazione di selezione delle dipendenze transitive migliore di quella di Maven. Allo stesso tempo permette anche di aggiungere ambiti di dipendenze personalizzati per ad esempio separare il contesto dei test di unità da quello dei test d'integrazione. Infine, nel caso sia necessario, è concesso retrocedere la versione di una dipendenza transitiva ad una desiderata.

1.4.2 Continuous Integration

Come afferma Martin Fowler, uno degli autori del manifesto, la *Continuous Integration*, abbreviata **CI**, è una pratica di sviluppo software in cui i membri di un team integrano frequentemente il proprio lavoro; di solito ogni persona integra almeno una volta al giorno, portando quindi a più integrazioni al giorno[18]. Ogni integrazione è verificata da una build automatizzata (incluso il testing) per rilevare errori di integrazione il più rapidamente possibile. Molti team scoprono che questo approccio porta ad una riduzione significativa dei problemi di integrazione e consente a un team di sviluppare software coeso in modo più rapido.

[18] https://archive.is/xZPD1

Martin Fowler introduce determinati principi, per permettere l'integrazione continua:

- Mantenere un repository per il codice sorgente: senza questo elemento propedeutico sarebbe impossibile automatizzare la build e i test.

- Automatizzare la build: deve essere possibile eseguire il processo di trasformazione del codice sorgente in artefatti eseguibili senza alcun intervento umano.

- Rendere la build auto-testante: ogni volta che viene eseguito il processo appena citato, è importante che vengano anche eseguiti i test affinché la qualità del codice rimanga costantemente elevata.

- Tutti devono eseguire commit quotidianamente: ciascuno degli sviluppatori del team deve necessariamente salvare le modifiche effettuate durante il giorno. In questo modo si evitano grosse incongruenze da parte del codice a lungo andare.

- Ogni commit fa partire una build: ogni salvataggio potrebbe generare dei bug, per evitarli è bene compilare ed effettuare i test. In questo modo i problemi vengono risolti nell'immediato al contrario di una verifica effettuata una tantum.

- Fare in modo che la build sia veloce: il processo non deve durare troppo a lungo, in quanto uno sviluppatore potrebbe dover aspettare prima di effettuare un altro salvataggio, perdendo tempo prezioso.

- Eseguire i test in un clone dell'ambiente di produzione: se l'ambiente su cui vengono eseguiti i test non è assolutamente identico all'ambiente di produzione, potrebbe esserci il rischio che qualcosa che è stato collaudato generi dei bug inspiegabili una volta rilasciato in produzione, con conseguenti danni economici.

- Fare in modo che sia facile ottenere le ultime versioni degli artefatti: è bene che tutti, anche i clienti finali, siano in grado di visualizzare i risultati ottenuti finora.

- Tutti devono poter visualizzare i risultati della build: in questa maniera chiunque può accorgersi di un eventuale problematica da risolvere il prima possibile.

- Automatizzare il rilascio: anche il rilascio e la distribuzione dovrebbero essere automatizzati, tramite le tecniche di *Continuous Delivery* e *Continuous Deployment*.

1.4.3 Continuous Delivery

La *Continuous Delivery* è la pratica che consiste nell'automazione del processo di rilascio del software. Essa porta una serie di vantaggi concreti ai progetti che risultano di grande aiuto per le aziende produttrici.

Per osservare nella pratica i risultati, si può ad esempio analizzare il caso *Paddy Power*, il quale commenta i benefici diretti rilevati dall'azienda nell'integrazione di questa pratica [10]:

- Accelerazione dei tempi di mercato: grazie all'incremento drammatico della frequenza dei rilasci, la consegna del software contenente l'incremento del valore aziendale cliente, avviene molto più rapidamente. Questa capacità aiuta l'azienda a stare al passo con la concorrenza in un ambiente economico così competitivo.

- Fabbricazione adeguata del prodotto: ora che i rilasci sono più frequenti, i feedback degli utenti vengono raccolti più velocemente. Questo permette agli sviluppatori di impegnare le risorse solo per le funzionalità utili. Al contrario, prima avrebbero potuto lavorare su alcuni aspetti che in realtà sarebbero risultati inutili per il cliente, ma che avrebbero scoperto solo dopo il prossimo rilascio, ossia dopo mesi di sforzi vani.

- Miglioramento della produttività e dell'efficienza: ora che gli ambienti di prova vengono configurati automaticamente dalla **CD**, gli sviluppatori hanno risparmiato circa il 20% del tempo che utilizzavano per effettuare la configurazione manuale di questi. Anche gli ingegneri che si occupavano della distribuzione, dovevano prendersi qualche giorno di lavoro per completare il rilascio in produzione. Al contrario ora devono solo pigiare un pulsante. In questa maniera si guadagna del tempo che può essere utilizzato per attività di maggior valore.

- Affidabilità dei rilasci: poiché i rilasci vengono effettuati più frequentemente, il codice aggiunto o modificato è difatti minore. Questo rende la ricerca degli eventuali problemi più facile, riducendo i tempi di impatto dei bug. Inoltre, grazie al fatto che un rilascio automatico possa essere facilmente cancellato, gli sviluppatori si sentono meno stressati rispetto a prima nei confronti del cosiddetto "giorno del rilascio".

- Miglioramento della qualità del prodotto: il numero di difetti trovati è diminuito di più del 90%. Ora che il testing viene eseguito automaticamente dopo il salvataggio, lo sviluppatore, nel caso ce ne sia bisogno, impegna del tempo per correggere il bug immediatamente, prima di passare ad un altra attività. Oltretutto, non sono più in alcun modo presenti gli errori legati alle configurazioni manuali del rilascio.

- Maggior soddisfazione del cliente: dapprima era presente una grande tensione tra i dipartimenti del cliente e i team di sviluppo. Ora la relazione è migliorata enormemente ed è stata stabilita una fiducia intaccabile.

A fronte di tutti questi benefici però ci sono alcune sfide da affrontare da più punti di vista.

Organizzativo Per definizione, la Continuous Delivery coinvolge più settori o divisioni delle aziende. Siccome ognuna di esse possiede i propri obiettivi ed interessi, vengono a crearsi delle tensioni interne all'azienda

a causa della competizione tra queste divisioni. Ci è voluto quindi un costo significativo di sei mesi per giungere ad una soluzione.

Processo Alcuni processi tradizionali ostacolano la CD. Ad esempio se una funzionalità fosse pronta entro brevissimo tempo, ma fosse presente una metodologia di verifica che tardi il rilascio di quattro giorni, allora bisognerebbe trovare una soluzione che soddisfi la CD.

Tecnico Spesso non esistono soluzioni robuste e personalizzabili per la CD e implementare gli strumenti da sé costa assai all'azienda. Inoltre, la CD non si presta in maniera ottimale nei riguardi di vaste e monolitiche applicazioni, di cui in realtà l'industria è piena. Sarebbe necessario ricercare delle soluzioni per sviare a tali problematiche.

Solitamente la sotto-fase di installazione del software presso l'ambiente di produzione viene definita *Continuous Deployment*. In letteratura non è ancora chiarissima questa distinzione, spesso infatti i termini vengono intercambiati, ma la maggior parte della comunità si trova d'accordo con la definizione appena riportata. Il Continuous Deployment quindi consiste nell'automazione del processo di rilascio ufficiale del prodotto software. Infatti, normalmente per la Continuous Delivery si opta per un'installazione manuale del software. Abitualmente la scelta di applicare o meno la Continuous Deployment dipende quasi esclusivamente dall'ambiente di produzione richiesto dal cliente, poiché quest'ultimo potrebbe risultare incompatibile con le tecnologie presenti sul mercato.

1.5 Plugin

Un plugin in campo informatico è un programma non autonomo che interagisce con un altro programma per ampliarne o estenderne le funzionalità originarie[19]. La capacità di un software di supportare i plugin è generalmente

[19] https://archive.is/EkARL

un'ottima caratteristica, perché rende possibile l'ampliamento e la personalizzazione delle sue funzioni da parte di terzi, in maniera comoda e veloce. Ciò favorisce da un lato la minore obsolescenza del software e dall'altro la maggior diffusione, tanto più sono numerosi e funzionali i plugin scritti per uno specifico programma o secondo uno standard specifico.

Il punto chiave dei plugin è la riusabilità. Infatti, poiché solitamente gli sviluppatori tendono ad esprimere dei bisogni comuni, l'utilizzo dei plugin permette di evitare l'implementazione di funzionalità già accessibili dagli utenti.

Ad esempio, Gradle offre una serie di plugin ufficiali, che aiutano gli sviluppatori per le diverse configurazioni base dei progetti, come *java* per lo sviluppo in Java, *scala* per lo sviluppo in Scala e tantissimi altri ancora che apportano le proprie modifiche alla configurazione per aiutare l'utente[20]. Oltre a ciò, Gradle permette a chiunque di realizzare il proprio plugin Gradle, per renderlo disponibile pubblicamente[21]. Il software deve solo sottomettersi ad alcune minime condizioni, per standardizzare lo sviluppo dei plugin.

Un plugin Gradle può effettivamente fare cose come[22]:

- Estendere il modello (ad esempio aggiungere nuovi elementi che possono essere configurati);
- Configurare il progetto secondo le convenzioni (ad esempio aggiungere nuove attività o preconfigurare alcune impostazioni ragionevolmente);
- Applicare specifiche configurazioni (ad esempio rafforzando gli standard).

Inoltre applicando un plugin si possono conseguire una serie di benefici:

- Promozione del riuso e riduzione dell'*overhead* dovuto al mantenimento di logiche simili tra vari progetti;

[20] https://docs.gradle.org/current/userguide/plugin_reference.html
[21] https://plugins.gradle.org/docs/publish-plugin
[22] https://archive.is/VMwUr#sec:what_plugins_do

- Permesso di un elevato grado di modularizzazione, migliorando la comprensibilità e l'organizzazione;
- Incapsulamento della logica imperativa e permesso agli script di risultare il più dichiarativi possibili.

1.5.1 Plugin Testing

Come tutti i software esistenti al mondo, anche un plugin Gradle possiede il proprio ciclo di vita. Anch'esso dovrebbe quindi superare la fase di test effettuando più controlli possibili. Esiste però un problema causato dalla natura del plugin stesso. Esso infatti poiché sfrutta il build tool, genera una dipendenza circolare nel momento in cui sia necessario verificarlo. Per far ciò infatti, è necessario utilizzare il build tool stesso per eseguire il test del plugin, il quale però dovrebbe nuovamente rieseguire il build tool per effettuare concretamente i dovuti controlli.

In Gradle questo problema viene risolto sfruttando un'interfaccia denominata *Tooling API*[23]. Questa libreria permette di utilizzare Gradle programmaticamente ed è infatti la stessa che viene utilizzata dagli ambienti integrati, come IntelliJ IDEA per importare i progetti Gradle.

Analogamente, per i plugin Gradle esiste una libreria ufficiale che mira ad eseguire il cosiddetto *functional testing*, ma come ammette Gradle stesso, probabilmente si espanderà per agevolare altri tipi di test.

[23] https://archive.is/Q2cb3#embedding

Capitolo 2

Progetto

Tutto il progetto è disponibile su GitHub.

Il progetto si pone come obiettivo principale lo sviluppo di una libreria software, che per definizione è un programma in grado di aiutare determinati utenti a soddisfare un preciso bisogno. In questo contesto il bisogno consiste nello sviluppo di codice di testing e quindi di verifica di plugin Gradle, propri degli utenti.

Per far ciò si basa su una libreria ufficiale di Gradle, denominata *Teskit*[1]. Inoltre, si presuppone sin da subito uno sviluppo incrementale delle funzionalità, perciò sono stati in seguito creati due DSL aggiuntivi per facilitare l'utente finale.

2.1 Analisi del dominio

Inizialmente si deve analizzare il dominio, per comprendere soprattutto cosa è essenziale rappresentare nel software. Anzitutto bisogna ragionare in base al funzionamento della libreria di partenza. Tale libreria consente, mediante una classe principale *GradleRunner*, di poter avviare programmaticamente una build Gradle. Questo consente quindi agli sviluppatori di osservare i risultati della build stessa, tra i quali l'output, ossia le stampe

[1] https://archive.ph/LMOWf

visualizzate a schermo, e l'insieme dei task eseguiti, ciascuno dei quali con il proprio esito. Dato che lo strumento è stato progettato ad-hoc, esso impone un'implicazione: le build di verifica vengono eseguite da un nuovo processo, separato da quello in cui viene eseguita la libreria stessa. Questo significa che il cosiddetto *classpath*, ossia l'insieme dei percorsi contenenti le librerie esterne del progetto in collaudo, risulti vuoto all'interno della nuova build. Esiste però una funzionalità apposita che ne permette l'iniezione automatica o manuale. Inoltre, si possono ovviamente configurare una serie di parametri per personalizzare la build. Tra questi è possibile impostare:

- la versione di Gradle da utilizzare, normalmente impostata alla versione corrente con cui viene eseguita la build.
- la cartella in cui eseguire il processo di test, normalmente impostata alla cartella *Temp* di sistema.
- la cartella in cui eseguire la build, cioè quella contenente il progetto Gradle (almeno un *build.gradle.kts*).
- l'insieme di argomenti dati dai nomi delle attività (task) da eseguire, seguiti da eventuali opzioni, come `--info`, per impostare il livello di logging a info.
- l'inoltro dell'output, in maniera tale da visualizzare l'output dei task eseguiti a schermo.
- il *classpath* del plugin, iniettato automaticamente o manualmente.

Sfruttando questa libreria è possibile conoscere e quindi verificare tutto ciò che riguardi la propria build.

2.2 Analisi dei requisiti

Dopo aver analizzato il dominio, è ora possibile comprendere di cosa vi sia bisogno per intermediare tra l'utente finale e il Testkit. Da quanto si è compreso precedentemente su Gradle e da quanto analizzato nella sezione appena

descritta, ciò che è possibile controllare riguarda principalmente le attività eseguite. È necessario quindi offrire l'opportunità all'utente di effettuare le verifiche a tal proposito, ma non solo.

Di seguito possono quindi essere elencati i requisiti del progetto in maniera descrittiva:

- Innanzitutto, bisogna dare la possibilità di controllare che una build debba fallire o meno di proposito, ad esempio se si volesse controllare che un task non esista.

- Inoltre, si deve sviluppare una funzionalità che permetta di controllare il tipo di esito dei task eseguiti.

- Analogamente, si deve sviluppare una funzionalità che permetta di controllare l'output della build eseguita, ad esempio per verificare che il proprio plugin stampi correttamente a schermo ciò che si desidera.

- È opportuno sviluppare una funzionalità che permetta di controllare esistenza, contenuto e permessi degli eventuali file generati dal plugin.

- Infine, come di solito convengono i framework di testing, è doveroso assegnare una descrizione a ciascun test che spieghi cosa il test verifichi.

2.3 Progettazione

Di seguito viene descritta la progettazione del software, sia dal punto di vista organizzativo, sia più nello specifico dal lato implementativo.

2.3.1 Scelte progettuali

Qui di seguito vengono elencati gli elementi fondamentali organizzativi del progetto, da diversi punti di vista.

Modularizzazione

Il progetto è stato modularizzato in modo da suddividere il codice in base alle funzionalità che esso debba svolgere. Il progetto si divide in quattro moduli:

- *buildSrc*: questo è un modulo particolare, dedicato per Gradle, che consente di ridurre la duplicazione del codice di scripting necessario per configurare i vari sottoprogetti elencati di seguito.
- *core*: questo è il modulo principale della libreria contenente il software per le funzionalità nucleari.
- *gradle-plugin*: questo è il modulo contenente l'implementazione per sviluppare il plugin Gradle che sfrutti il modulo principale.
- *tests*: questo è il modulo dedicato al testing, proprio della libreria.

Quality Assurance

Un codice di alta qualità è l'essenza di un buon prodotto finale. A tal scopo, sono stati configurati dei meccanismi per mantenere un determinato livello di qualità. Innanzitutto, poiché l'unico linguaggio di programmazione utilizzato è Kotlin, è stato dichiarato un plugin Gradle che vada a controllare tutti i sorgenti Kotlin in maniera statica. Esso infatti utilizza internamente *Ktlint*[2] e *Detekt*[3], due strumenti di controllo di qualità i quali rispettivamente controllano lo stile del codice e ne individuano possibili difetti.

Inoltre, per incrementare ulteriormente la sicurezza e la qualità del codice, viene eseguita anche un'analisi da *Sonarcloud*[4]. Tramite il plugin apposito infatti, esso permette di utilizzare un servizio che consente di eseguire un'analisi statica del codice e scoprire quindi eventuali problemi.

[2] https://pinterest.github.io/ktlint/
[3] https://detekt.dev/
[4] https://www.sonarsource.com/products/sonarcloud/

Ancora, come ulteriore indice di qualità viene anche controllata la cosiddetta *coverage*, usufruendo di un altro strumento open-source, denominato *codecov*[5]. La coverage, letteralmente tradotta copertura, è una misura utilizzata nello sviluppo del software per comprendere quanto e quale codice venga eseguito durante le verifiche. Si può basare la copertura su diversi criteri. Principalmente se ne utilizzano due: *statement coverage* e *branch coverage*. La prima viene misurata calcolando quante delle istruzioni del programma vengono effettivamente eseguite durante i test, ottenendo quindi la percentuale di istruzioni "coperte". Allo stesso modo la seconda tiene conto del numero di diramazioni del codice, dovute ad espressioni condizionali. Rappresenta quindi un buon criterio di misurazione per comprendere l'effettivo collaudo del software, sebbene non sia possibile affermare che un programma con il 100% di coverage sia completamente esente da *bug*.

Versioning

Per tenere traccia delle differenze nello sviluppo progressivo del progetto, è stato utilizzato un processo di versionamento; in particolare è stato adottato lo standard *Semantic Versioning*, il quale prevede che una versione sia principalmente caratterizzata da:

- *Major*: numero identificativo da aggiornare a ogni cambiamento non-retrocompatibile.
- *Minor*: numero identificativo da aggiornare a ogni aggiunta o modifica retrocompatibile di una funzionalità del sistema.
- *Patch*: numero identificativo da aggiornare a ogni correzione dei difetti del sistema.

Adottando la specifica *Conventional Commits*[6] durante lo sviluppo tramite DVCS, è stato possibile automatizzare l'assegnamento della versione

[5] https://about.codecov.io/
[6] https://www.conventionalcommits.org/en/v1.0.0/

al sistema sulla base dei commit effettuati. Per fare ciò, è stato sfruttato *Semantic Release Bot*.

Inoltre, per forzare il sottoscritto o eventuali collaboratori del progetto ad adottare tale convenzione, è stato utilizzato un plugin Gradle che prima di effettuare il salvataggio delle modifiche al progetto, controlla che il messaggio segua lo standard adottato.

Distribuzione

Dal momento che si è scelto di utilizzare Kotlin come linguaggio per l'implementazione della libreria e del plugin Gradle, inevitabilmente la distribuzione di questi moduli deve essere effettuata tramite i cosiddetti *JAR* (Java ARchive). Questi sono pacchetti compressi contenenti del codice particolare, detto *Java Bytecode*, il quale viene generato dalle classi originarie. Questo codice contiene istruzioni che possono essere eseguite soltanto da una *Java Virtual Machine*, ma il cui diretto vantaggio è la portabilità di quest'ultima rispetto a qualsiasi sistema operativo su cui quindi si vuole eseguire il progetto.

Per la distribuzione del modulo *core*, si usufruisce della piattaforma Maven Central[7], la quale richiede alcuni requisiti per determinare uno standard uguale per tutti gli sviluppatori. Per la pubblicazione infatti oltre al JAR descritto in precedenza, è necessario anche un JAR contenente i sorgenti e uno contenente la documentazione, ciascuno dei quali deve essere firmato da una chiave GPG.

Continuous Integration

Per fare in modo che il codice rimanga integro e corretto durante lo sviluppo, è stato utilizzato un workflow che attraverso GitHub Actions permetta di eseguire i test del progetto su diverse configurazioni di sistemi operativi, a ogni aggiornamento del progetto.

[7] https://central.sonatype.com/

Continuous Delivery

Analogamente, un altro workflow permette di automatizzare la release su GitHub Releases e Maven Central. Ad ogni versione, il bot *Semantic Release* aggiorna anche un file denominato *CHANGELOG* che mantiene lo storico dei cambiamenti tra ciascuna versione e la successiva.

Documentazione

Come da manuale, il codice deve essere commentato. Per permettere quindi all'utente di poter leggere e studiare la documentazione del progetto, è stato sfruttato un servizio gratuito di hosting, *javadoc.io*[8], che, mediante la distribuzione della documentazione sotto forma di archivio Java sulla piattaforma Maven Central, offre la possibilità di pubblicare online la propria documentazione. Per generare la documentazione è stato utilizzato *Dokka*, di cui si parlerà in seguito.

Version control strategy

Data la semplice composizione del team di sviluppo, formato da un unico sviluppatore, si è scelto di utilizzare un unico *branch* principale di sviluppo, che contenga tutta lo storico del progetto. Per aggiungere nuove funzionalità, correggere difetti, o aggiornare le dipendenze, vengono creati nuovi branch temporanei i quali vengono fusi al *master* branch, tramite la tecnica di *rebase*, in modo tale da ottenere uno storico chiaro e definito. Per quanto riguarda il versionamento, si è partiti dalla versione iniziale *0.1.0*, pubblicando ogni versione successiva, ma è bene notare che debbano ritenersi stabili solo quelle maggiori o uguali alla *1.0.0*.

[8] https://javadoc.io/

Automated Evolution

Per mantenere aggiornate le dipendenze del progetto in maniera automatica è stato utilizzato un sistema open-source, *Renovate*[9] che tramite un bot osserva le dipendenze del progetto e in caso di necessità apre delle *pull request* sul repository GitHub per aggiornarle.

Oltre a ciò, per condividere le dipendenze tra i progetti e per avere un unico raggruppamento delle dipendenze, queste sono state tutte inserite all'interno di un *version catalog*, ossia un file con estensione *.toml* che, seguendo una determinata notazione, permette di indicare facilmente modulo e versione di un artefatto necessario per il progetto.

Licensing

Per standardizzare il software, è stata scelta una licenza che ne indichi le modalità di utilizzo, nello specifico è stata scelta la licenza *MIT*[10]. La motivazione dietro questa scelta è ovviamente lo scopo finale del progetto, dato che il progetto debba risultare open-source. Per verificare la compatibilità tra la licenza del progetto e quelle delle sue dipendenze è stato utilizzato *Fossa*[11] che tramite un bot, a ogni aggiornamento delle dipendenze esegue tale verifica.

Inoltre, a ciascuna classe viene applicato un *header* che rappresenti il copyright dato dalla licenza.

2.3.2 Codebase

Di seguito vengono spiegate le classi relativi ai due moduli principali.

[9] https://www.mend.io/renovate/
[10] https://opensource.org/license/mit/
[11] https://fossa.com/product/open-source-license-compliance

core

La libreria ha un unico scopo principale: permettere di eseguire test, dichiarati strutturalmente, all'interno di una cartella contenente un progetto Gradle. Per far ciò è stata quindi progettata proprio la struttura dei test. Sono state infatti pensate le apposite classi che debbano contenere le informazioni utilizzabili successivamente. Per rappresentare al meglio la struttura finale che si desidera, le classi presentano una gerarchia di componentistica tra di esse. Inoltre, per permettere all'utente di creare i test in maniera dichiarativa si è scelto di fornire una modalità di descrizione tramite file *yaml*.

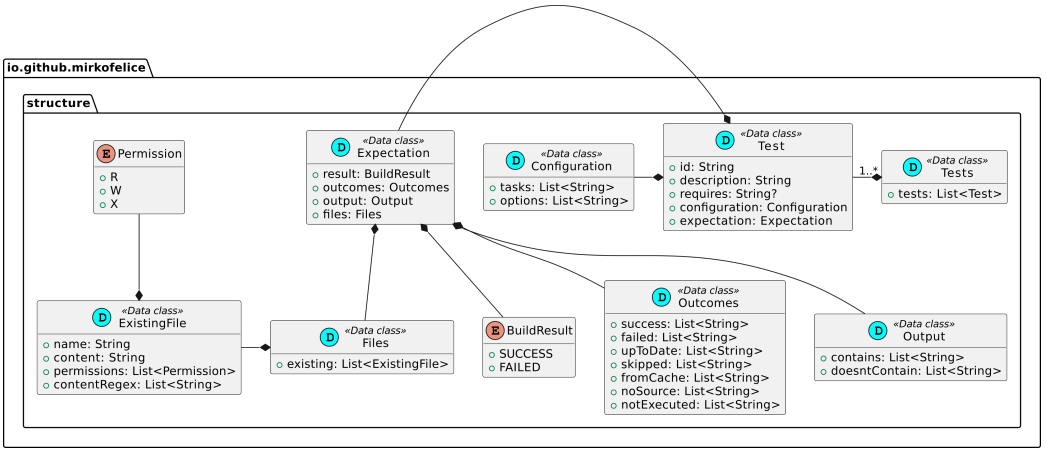

Figura 2.1: *Diagramma UML delle classi del package* `structure` *nel modulo* ***core****.*

Le classi sono state organizzate e suddivise per `package`.

Come si osserva in figura 2.1 nel package `structure` per rispecchiare una definizione sensata e semplice dei file yaml, le classi sono state collezionate in questa maniera:

- `Tests`: rappresenta l'insieme dei test;
- `Test`: rappresenta un singolo test;

- `Configuration`: rappresenta la configurazione di un test, formata dall'insieme dei task da eseguire e dalle eventuali opzioni;
- `Expectation`: rappresenta le aspettative di un test, formata dalle classi successive;
- `BuildResult`: rappresenta il risultato aspettato dalla configurazione dichiarata;
- `Outcomes`: rappresenta le aspettative sugli esiti dei vari task dichiarati nella configurazione;
- `Output`: rappresenta le aspettative in termini di frasi sull'output dei task dichiarati nella configurazione;
- `Files`: rappresenta le aspettative su eventuali file relativi ai task dichiarati nella configurazione;
- `ExistingFile`: rappresenta l'aspettativa di un singolo file che debba esistere e che debba possedere determinati permessi e un preciso contenuto;
- `Permission`: rappresenta uno specifico permesso relativo ad un ExistingFile.

Nel package `checkers` invece, in figura 2.2, è stata idealizzata un'interfaccia `TestkitChecker`, la quale permette di eseguire i vari controlli in base alla struttura citata sopra. Essa offre un'unica funzionalità `check`, che internamente sfrutta altri metodi, dichiarati astratti, per effettuare i controlli relativi agli esiti dei task, al contenuto dell'output e all'esistenza, ai permessi e al contenuto dei file. L'unica implementazione corrente di questa interfaccia è il `KotlinChecker`, che utilizza alcune funzionalità fornite di base da Kotlin, le quali permettono di lanciare un errore in caso i controlli non vadano a buon fine.

D'altro canto, nel package `api`, è presente un'enumerazione `CheckerType` contenente tutti i tipi di TestkitChecker possibili, attualmente implementati.

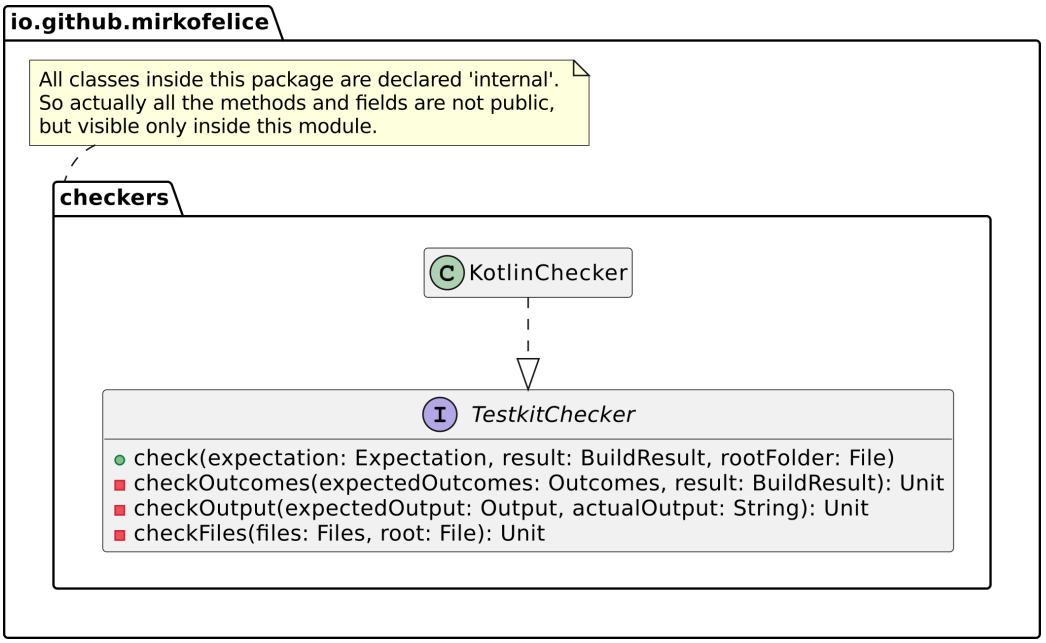

Figura 2.2: *Diagramma UML delle classi del package* `checkers` *nel modulo* ***core***.

Una parentesi da far notare consiste nella seguente scelta. Data l'apertura alla comunità open-source del progetto, è stato pensato che un potenziale collaboratore possa sviluppare una nuova personale implementazione del TestkitChecker, che dovrà quindi essere messa a disposizione inserendo il nuovo tipo nell'enumerazione CheckerType.

Come mostrato in figura 2.3, contiene poi il `Testkit`, che è l'oggetto che permette mediante funzionalità pubbliche di eseguire i test, a partire dai parametri necessari. Esso sfrutta internamente gli oggetti contenuti all'interno del package `runners`.

In questo package, sono presenti due `object`, uno dei quali utilizza l'altro, come visualizzabile in figura 2.4. Il `TestRunner` consente di eseguire i test, sfruttando direttamente la classe appropriata descritta precedentemente. Per ogni test, esso ottiene il risultato dal `BuildRunner` e ne effettua le verifiche, utilizzando il rispettivo TestkitChecker scelto dall'utente. Il BuildRunner

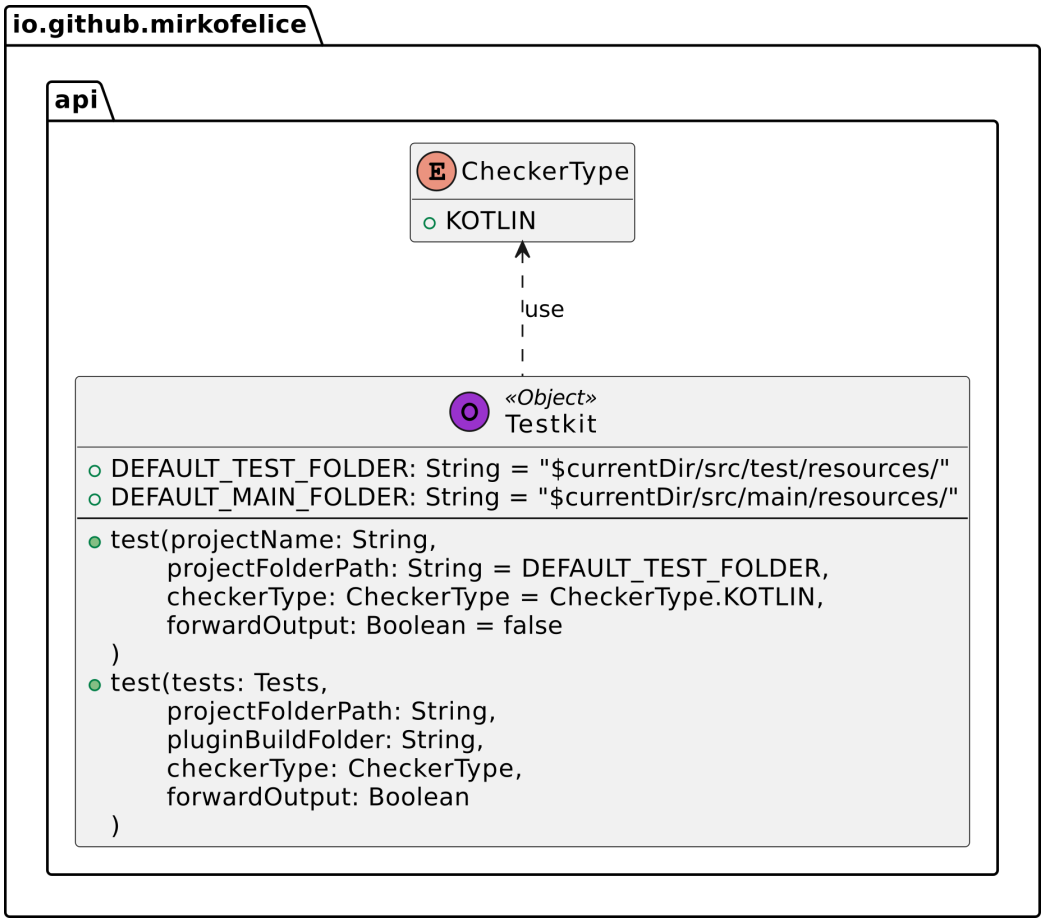

Figura 2.3: *Diagramma UML delle classi del package* ***`api`*** *nel modulo* ***core***.

quindi prevede un'unica funzionalità che consiste nell'utilizzare la libreria di Gradle per eseguire la build.

È importante sottolineare che i package runners e checkers sono invisibili all'utente finale, poiché le varie implementazioni sono dichiarate `internal`, permettendo un incapsulamento delle funzionalità eccellente per il caso d'uso in cui il progetto si ritrova. All'utente finale infatti, non serve assolutamente controllare queste classi, ma solamente le funzionalità pubbliche presenti negli altri due package.

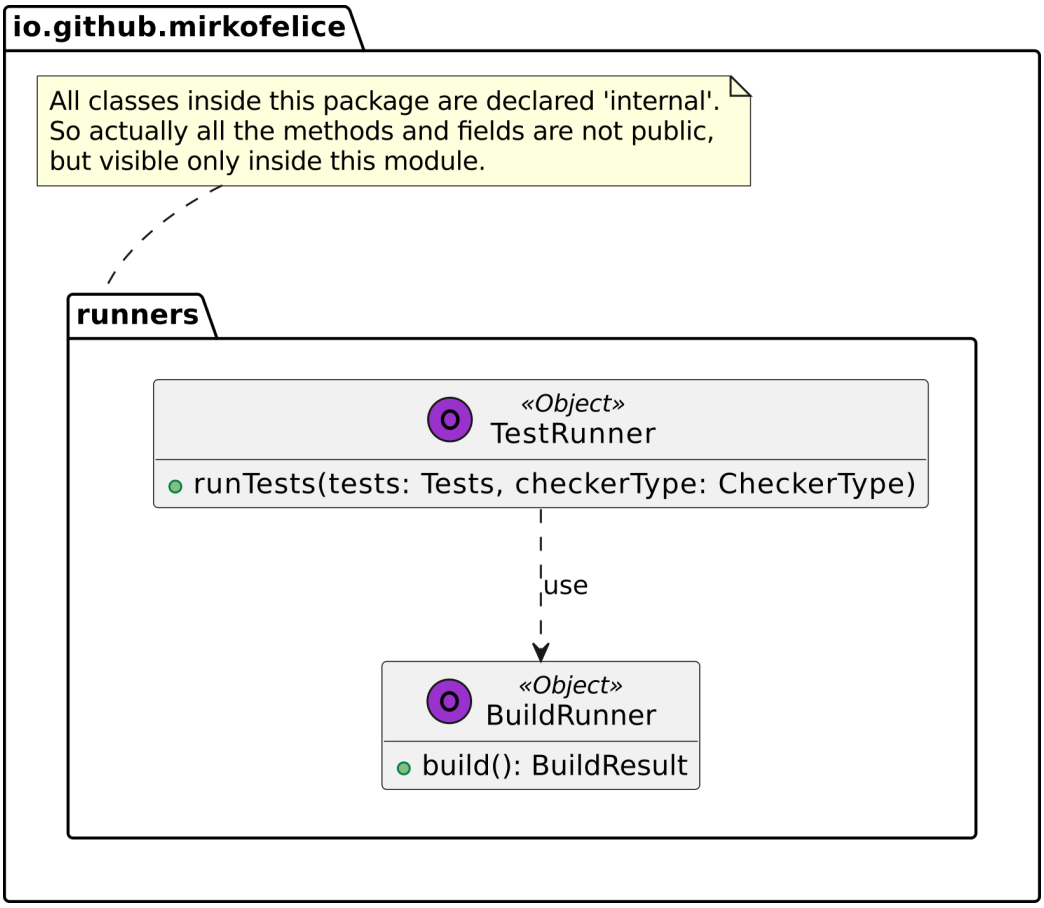

Figura 2.4: *Diagramma UML delle classi del package* ***`runners`*** *nel modulo* ***core***.

gradle-plugin

Per facilitare l'utente finale, è stato creato anche un plugin Gradle relativo alla libreria. Questo infatti permette di evitare la creazione di classi di test per eseguire i controlli desiderati.

Fondamentalmente questo plugin crea un'estensione per il build script di Gradle, nel quale è possibile configurare il plugin stesso. Questa infatti deve poter inizializzare correttamente i propri task. Difatti questo plugin crea un task principale, denominato *testkit*, che dipende da altri due task implementati direttamente dal plugin stesso. Come si osserva in figura 2.5, il

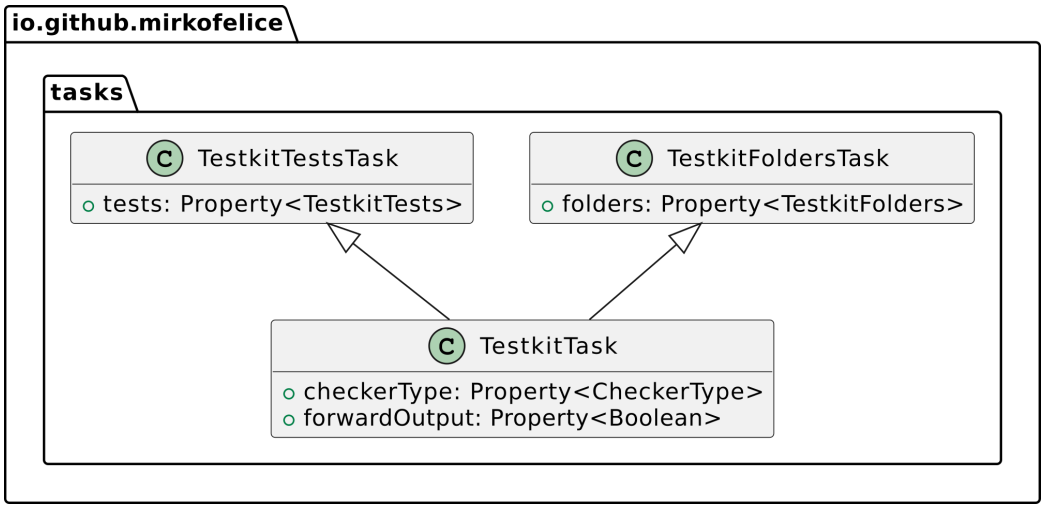

Figura 2.5: *Diagramma UML delle classi del package* ***`tasks`*** *nel modulo* ***gradle-plugin****.*

`TestkitTask` viene dotato di due `Property`: una per definire il CheckerType da utilizzare nell'esecuzione dei controlli, e un'altra per impostare la presenza o l'assenza del desiderio dell'utente a visualizzare l'output dei task del proprio plugin che devono essere eseguiti per osservarne i risultati.

I due task figli del `TestkitTask`, riguardano le due modalità di utilizzo della libreria principale.

La prima, impiegata dal `TestkitFoldersTask`, necessita di conoscere quale sia la cartella che contiene correttamente sia il file yaml, descrittore dei test, sia il progetto Gradle ove il plugin dell'utente sia applicato e eventualmente configurato. Per far ciò e grazie alle potenzialità racchiuse in Kotlin, viene creato un breve DSL che permetta di configurare il necessario, mostrato in figura 2.6. Le sue funzionalità permettono di aggiungere più cartelle, siano esse generiche all'interno del sistema su cui il plugin viene eseguito, o siano esse facenti parte del progetto in cui il plugin viene applicato. Inoltre è possibile aggiungere un insieme di sottocartelle che siano correttamente strutturate. Infine, il DSL è stato dotato di due funzionalità aggiuntive che permettono di inserire rapidamente due tipi di cartelle di default, in cui

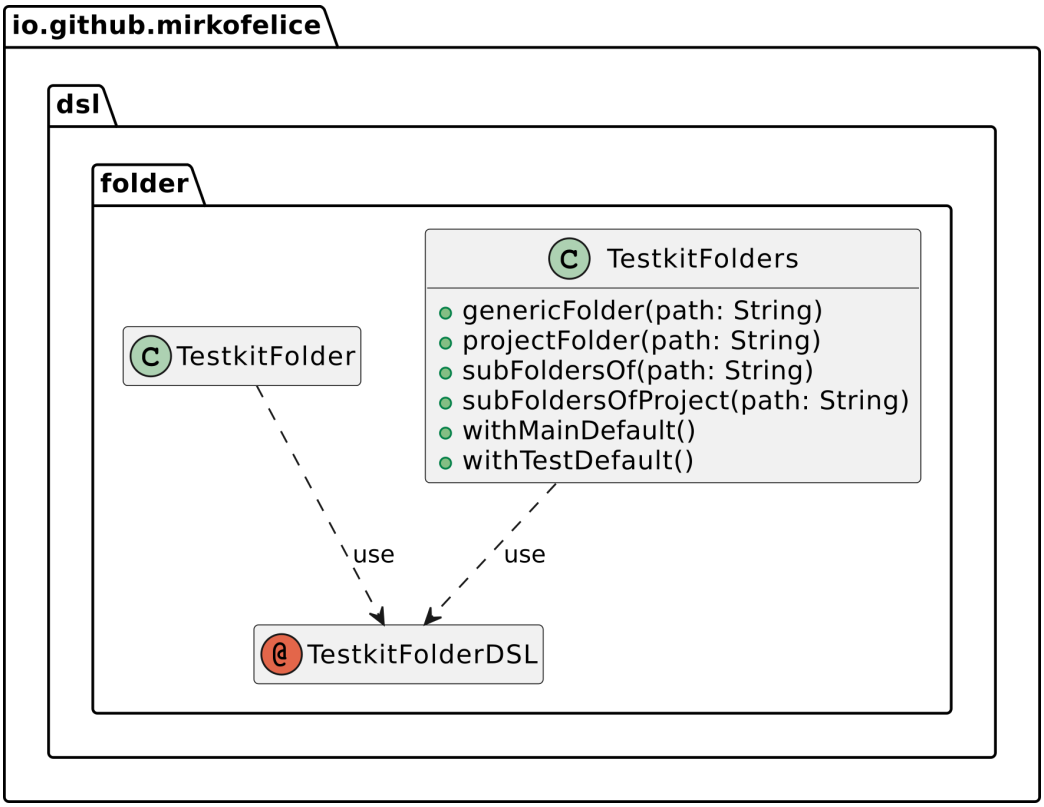

Figura 2.6: *Diagramma UML delle classi del package* `folder` *nel modulo* ***gradle-plugin***.

solitamente gli sviluppatori vanno a creare le risorse in questi casi: quella relativa al percorso *src/main/resources* e quella relativa a *src/test/resources*.

La seconda modalità di utilizzo invece, impiegata dal `TestkitTestsTask`, permette la creazione manuale dei test, proponendo quindi un'alternativa alla dichiarazione tramite file yaml. Prima di tutto però bisogna specificare comunque la cartella contenente il sotto-progetto da testare.

Difatti, come si può osservare in figura 2.7, un insieme di classi, che rispecchiano una ad una quelle relative al package structure del modulo core, costituisce un DSL chiaro e pulito per dichiarare i veri e propri test. Questi però, per poter essere utilizzati dalla libreria principale, devono dapprima essere convertiti nelle corrispondenti classi. Per questo motivo, ciascuna di

queste implementa un'interfaccia `Convertable`$\langle T \rangle$ che offre la possibilità di convertire il tipo dell'oggetto in un'altra classe.

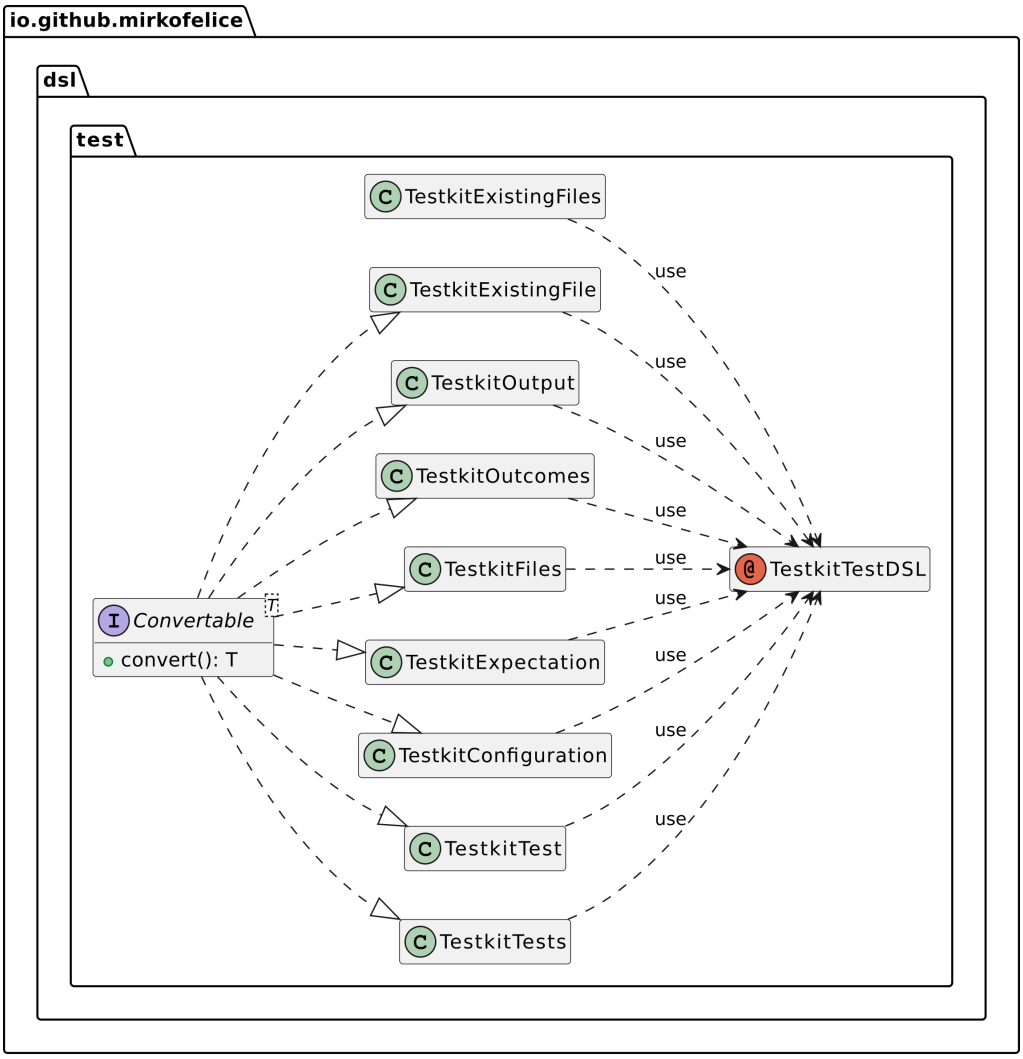

Figura 2.7: *Diagramma UML delle classi del package* `test` *nel modulo* ***gradle-plugin***.

Una nota da evidenziare è l'utilizzo di due annotazioni che permettono l'incapsulamento delle funzionalità dei due DSL, in maniera tale da prevenire incongruenza semantica: ad esempio, non dovrebbe essere possibile dichiarare due volte il DSL *folders*.

2.4 Tecnologie utilizzate

2.4.1 Kotlin

Kotlin è un linguaggio di programmazione *general purpose*, multi-paradigma e open-source, sviluppato nel 2011 dall'azienda di software JetBrains, nota soprattutto per il famoso ambiente di sviluppo IntelliJ IDEA[12]. Un linguaggio general purpose non è specializzato su un certo tipo di sviluppo, ma è predisposto a offrire funzionalità e vantaggi per tutte le possibili programmazioni[13]. Allo stesso tempo è anche un linguaggio multi-paradigma, per cui non si limita ad offrire un solo stile di programmazione, ma ne combacia molteplici[14]. Alcuni esempi di paradigmi di programmazione sono la programmazione strutturata tipica del linguaggio C, o la programmazione orientata agli oggetti tipica di Java, o la programmazione funzionale come può offrire Python. Open source sta ad indicare che il codice sorgente è ad accesso libero.

Kotlin si basa sulla Java Virtual Machine (JVM), componente della piattaforma Java che esegue i programmi tradotti in bytecode, dopo la compilazione dei codici sorgente, ed è ispirato ad altri linguaggi tra i quali Scala e lo stesso Java. È un linguaggio a tipizzazione statica e forte, per cui ciascuna variabile deve possedere un tipo che sia un intero, un carattere o un oggetto e che tale variabile debba effettivamente eseguire operazioni consone al proprio tipo. Inoltre, è particolarmente orientato verso la programmazione ad oggetti permettendo anche un approccio di tipo funzionale. Il nome del linguaggio deriva dall'omonima isola vicino a San Pietroburgo, succursale della sede principale di JetBrains a Praga. Il leader del progetto di sviluppo di Kotlin ha affermato che è stato progettato per essere interoperabile con l'ambiente Java e al contempo superare le sue criticità come errori particolari.

Kotlin è inoltre il linguaggio ufficiale di Android, come annunciato da

[12] https://archive.ph/wIArs
[13] https://archive.ph/XlOpH
[14] https://archive.is/7Ze3s#Multi-paradigm

Google nel 2019. Come Android stesso afferma, si possono riassumere quattro aspetti positivi nell'utilizzare Kotlin[15]:

- **Espressivo e conciso:** Si può fare di più scrivendo di meno, diminuendo il codice *boilerplate*, ovvero sezioni in cui il codice si ripete senza praticamente nessuna alterazione.
- **Codice più sicuro:** Si può aumentare la qualità del sistema. Kotlin possiede infatti alcune funzionalità che aiutano a evitare errori di programmazione usuali come i *Null Pointer Exceptions*.
- **Interoperabilità:** Questa caratteristica permette di utilizzare all'interno dello stesso progetto sia file in Kotlin che in Java.
- **Concorrenza Strutturata**: Le *coroutines* di Kotlin rendono il codice asincrono facile da implementare tanto quanto il codice bloccante. Le coroutines semplificano drammaticamente la gestione del lavoro in background per qualsiasi cosa, dalle chiamate di rete all'accesso ai dati locali.

2.4.2 Git

Git è un DVCS (*Distributed Version Control System*) per tracciare cambiamenti nel codice sorgente durante lo sviluppo del software. È progettato per coordinare il lavoro tra i programmatori, ma può essere usato per tracciare modifiche in un qualsiasi insieme di file. I suoi obiettivi includono velocità, integrità dei dati e supporto per flussi di lavoro distribuiti e non lineari[16]. Git è stato creato da Linus Torvalds nel 2005. Linus Torvalds è anche e soprattutto l'ingegnere informatico creatore del kernel Linux e Git è stato inventato proprio allo scopo di sviluppare in modo efficiente il kernel.

Un DVCS in informatica è una forma di controllo di versione in cui tutto il codice sorgente inclusa tutta la sua storia, è rispecchiata in ogni computer

[15] https://developer.android.com/kotlin/first
[16] https://archive.is/4TdGb

di ciascun sviluppatore. Un software per il controllo di versione è un sistema responsabile della gestione dei cambiamenti di documenti e programmi e ogni sorta di collezione di informazioni[17].

Ci sono due tipi di sistemi di controllo:

- I sistemi distribuiti, i quali utilizzano un approccio di tipo *peer-to-peer*.
- I sistemi centralizzati, i quali utilizzano un approccio di tipo *client-server*.

I vantaggi che un DVCS offre sono illustrati di seguito:

- Permette agli utenti di lavorare produttivamente anche quando non sono connessi alla rete.
- Le operazioni usali come i *commit* sono più veloci perché non c'è bisogno di comunicare con un server centrale. La comunicazione è necessaria solo quando bisogna condividere i cambiamento con gli altri utenti.
- Permette agli utenti di lavorare in proprio, cioè possono usare i propri cambiamenti che magari non vogliono pubblicare.
- Le copie di lavoro funzionano effettivamente come backup remoti, i quali evitano di contare solo su una macchina fisica che rappresenta il singolo *point of failure*.
- Permette di utilizzare diversi modelli di sviluppo, come l'utilizzo di *branch* di sviluppo.
- Autorizza il controllo centralizzato della versione rilasciata del progetto.

Alcuni svantaggi invece sono:

[17] https://archive.ph/wKlUy

- Il *checkout* iniziale di un *repository* è più lento rispetto ad un CVCS, perché tutti i branch e la storia delle revisioni devono essere copiati sulla macchina locale.

- Mancanza di meccanismi di blocco che sono una parte principale dei sistemi di controllo centralizzati poiché giocano un ruolo importante quando devono essere tracciati file binari.

- Necessario spazio di archiviazione per ciascun utente per avere una copia completa di tutta la struttura del progetto e della sua storia.

- Aumento di esposizione al rischio del progetto poiché tutti i partecipanti ne possiedono una copia vulnerabile.

Le componenti base di Git sono due strutture dati: un indice modificabile che mantiene le informazioni sul contenuto della prossima revisione, e un database di oggetti a cui si può solo aggiungere e che contiene quattro tipi di oggetti:

- Blob: un oggetto *blob* è il contenuto di un file. Gli oggetti blob non hanno metadati.

- Albero: un oggetto *albero* è l'equivalente di una *directory*. Contiene una lista di nomi di file che possono essere blob o altri alberi.

- Commit: un oggetto *commit* collega gli oggetti albero in una cronologia. Contiene un messaggio di archiviazione (*log*), dei commit genitori, data, ora e un oggetto albero radice.

- Tag: un oggetto *tag* è un contenitore che contiene riferimenti a un altro oggetto. L'uso più comune è memorizzare la firma di una particolare oggetto commit relativo a una *release* del progetto.

2.4.3 Dokka

Dokka[18] è il motore logico apposito per la generazione di documentazione Kotlin. Il progetto open-source permette, a partire dai commenti in stile KDoc o Javadoc, di generare la documentazione in più formati. Principalmente offre 3 formati: HTML, Markdown e Javadoc.

Il progetto di tesi usa il formato HTML, in quanto quello raccomandato da Dokka stesso, per generare una documentazione del codice comprensibile agli utenti.

2.4.4 Jackson

Questa è la libreria che utilizza il modulo principale del progetto per riuscire a trasformare i file yaml in vere e proprie classi Kotlin. Jackson infatti è una suite di strumenti di elaborazione dei dati per le piattaforme basate sulla JVM.

È necessario spiegare cosa sia la serializzazione di un oggetto. La serializzazione è il processo di trasformazione dello stato di un oggetto in un formato che possa essere memorizzato (come i file in memoria) o trasmesso (come i flussi di byte in rete), allo scopo di trasmettere le informazioni necessarie da una macchina ad un'altra[19]. Deve essere infatti presente anche l'operazione opposta, denominata deserializzazione, la quale permette la ricostruzione dell'oggetto a partire dai dati. I formati più gettonati sono appunto YAML, JSON e XML.

Jackson è organizzato in vari progetti e moduli; quelli utilizzati dal progetto sono fondamentalmente due:

- dataformat-yaml[20]: tale modulo contiene le estensioni necessarie per permettere la lettura e scrittura di file yaml.

[18] https://github.com/Kotlin/dokka
[19] https://archive.ph/LCxr7
[20] https://github.com/FasterXML/jackson-dataformats-text/tree/2.16/yaml

- module-kotlin[21]: questo modulo aggiunge supporto per la serializzazione e deserializzazione delle classi Kotlin.

[21] https://github.com/FasterXML/jackson-module-kotlin

Capitolo 3

Validazione

Per effettuare una corretta validazione, in primo luogo, sono stati effettuati dei test interni. Successivamente, il progetto è anche stato applicato a diversi plugin open-source preesistenti.

Per confermare la validità del progetto, deve essere scelta una specifica metrica. In letteratura, esistono diversi tipi di metriche concepite negli anni. Le più comuni sono [11]:

- Linee di codice: abbreviata LOC, è la più basilare poiché si fonda semplicemente sul numero di linee.
- Linee di codice sorgente: abbreviata $SLOC$, è la versione migliorata della precedente in quanto prevede l'eliminazione delle righe considerabili superflue, come quelle contenenti commenti o le righe vuote.
- Numero di funzioni: dato che solitamente le funzioni dovrebbero racchiudere brevi porzioni di logica, è possibile utilizzarle per comprendere la complessità del codice.
- Complessità ciclomatica: rappresenta una vera e propria funzione, trovata da McCabe, che è calcolata a partire dal grafo di controllo del flusso del programma, corrispondente al numero di cicli e condizioni rappresentate nel codice. Tecnicamente, per un grafo con v nodi, e archi e p punti di uscita, la formula è: $C = e - v + 2p$

- Metriche di Halstead: queste metriche tentano di misurare la complessità logica del programma in base alla ripetitività delle operazioni e degli operandi.

Sono poi presenti altre metriche relative alla programmazione orientata agli oggetti, che riguardano ad esempio la correlazione tra le classi [12].

Per questo progetto è stata scelta come metrica il numero di linee di codice sorgente.

A seguire vengono commentati tutti i dettagli.

3.1 Testing

Il modulo *tests* è stato sviluppato a tal proposito. Esso infatti simula un virtuale progetto utente, che contenga più plugin che si suppone egli abbia creato, ed effettua il testing sfruttando la libreria proposta dal progetto. In questa maniera si è potuto controllare il corretto funzionamento di quest'ultima e correggere gli errori trovati durante lo sviluppo. Inoltre, ciò è servito anche a misurare la coverage del progetto.

Al contrario, per verificare il plugin Gradle, si è dovuti ricorrere all'utilizzo del Testkit ufficiale. Non è stato possibile infatti ricorrere in nessuna maniera ad una ricorsione del codice per riutilizzare il plugin stesso per effettuare i dovuti controlli.

Per realizzare quest'indagine sono stati quindi necessari due framework aggiuntivi.

3.1.1 Kotest

Kotest è un framework di testing multipiattaforma flessibile ed elegante per Kotlin con asserzioni estese e *property testing* integrato[1].

Kotest offre diversi stili di testing, ispirati o meno da altri framework, che permettono agli sviluppatori di scegliere quello più appropriato al proprio

[1] https://kotest.io/

caso d'uso. In questo progetto si è ritenuto opportuno utilizzare lo stile *StringSpec*, che banalmente dichiara testualmente cosa il test deve fare e al suo interno viene utilizzata la libreria. Un esempio parziale di classe del progetto viene riportato nel listato 3.1.

Listato di codice 3.1: Codice di esempio della StringSpec.

```
import io.github.mirkofelice.api.Testkit
import io.kotest.assertions.throwables.shouldThrow

class HelloTests : StringSpec({
    "Non existing task" {
        shouldThrow<IllegalStateException> {
            Testkit.test("hello/wrongExistingTask")
        }
    }
})
```

3.1.2 JUnit

JUnit è un framework di testing molto famoso per l'ambiente Java[2]. In particolare dalla versione 5, esso si è modularizzato in diversi progetti. JUnit propone una modalità più lineare di stile dei test, a vantaggio però di una vasta gamma di parametri e funzionalità per personalizzare i controlli. Ad esempio, una funzionalità molto potente e in via sperimentale utilizzata anche nel progetto riguarda l'iniezione nei test di una cartella temporanea, generata in automatico dal framework usando l'annotazione *@TempDir*. Un'altra funzionalità fornita da JUnit consiste nella possibilità di evitare la ripetizione di codice, grazie ad un'ulteriore annotazione, *@BeforeEach*, che permette di eseguire un pezzo di codice ogni volta prima di un effettivo test. L'utilizzo delle due funzionalità viene mostrata nel listato di codice 3.2.

[2] https://junit.org/

Listato di codice 3.2: Codice di una classe di test che utilizza JUnit.

```
class PluginTest {

    @TempDir
    lateinit var tempDir: File
    private lateinit var buildFile: File
    private lateinit var buildContent: String

    @BeforeEach
    fun setup() {
        buildFile = tempDir.resolve("build.gradle.kts")
        buildContent = """
            plugins {
                id(''io.github.mirko-felice.testkit'')
            }
        """.trimIndent()
    }

    @Test
    fun withMainAndTestFolders() {
        buildContent += """
            testkit {
                folders {
                    withMainDefault()
                    withTestDefault()
                }
            }
        """.trimIndent()
        runGradleBuild(testMode = true)
    }
```

3.2 gradle-kotlin-qa

Questo è stato il primo plugin a cui è stato applicato il progetto[3]. Dopo una fase di correzione di problematiche relative alla prima effettiva validazione della libreria, quest'ultima è stata correttamente applicata. A seguito di ciò infatti, sono stati portati una serie di vantaggi al plugin di riferimento. Innanzitutto sono state eliminate le dipendenze che il plugin sfruttava per creare i propri test sfruttando ovviamente principalmente sempre la libreria Testkit. In seguito, sono state sostituite le classi preesistenti con un'unica classe di test. Al suo interno sono stati riscritti i test utilizzando la nuova libreria.

In totale sono state aggiunte 50 nuove linee di testo, al netto di 134 eliminate. Sommariamente quindi sono state rimosse 84 linee di codice. Quello che è ovviamente il beneficio maggiore, è un codice di qualità più elevata in quanto più pulito e chiaro al proprietario del plugin.

3.3 publish-on-central

Questo è stato il secondo plugin a cui è stato applicato il progetto[4]. Sono state eseguite le stesse operazioni necessarie per il primo plugin, con una ulteriore modifica relativa alla differenza di progettazione della libreria proposta. Quest'ultima infatti presupponeva che i task venissero eseguiti in successione nella stessa cartella temporanea contenente il progetto da testare. Perciò è stato necessario modificare il design, aggiungendo una funzionalità che permetta ai test di dipendere da un altro test, ossia eseguiti nella stessa cartella. In questo modo, di default, i test vengono invece eseguiti in cartelle completamente separate per permettere agli utenti di considerare ciascun test come nuovo rispetto agli altri.

[3] https://github.com/DanySK/gradle-kotlin-qa/pull/489
[4] https://github.com/DanySK/publish-on-central/pull/671

In totale sono state aggiunte 149 nuove linee di testo, al netto di 231 eliminate. Sommariamente quindi sono state rimosse 82 linee di codice. Analogamente, il beneficio maggiore è un codice di qualità più elevata.

3.4 Template-for-Gradle-Plugins

Questo è stato il terzo plugin a cui è stato applicato il progetto[5]. Si tratta di un template da cui partire per sviluppare dei propri Plugin Gradle. In questo caso non si è presentato il bisogno di modificare la libreria, in quanto funzionava già perfettamente.

In totale sono state aggiunte 31 nuove linee di testo, al netto di 171 eliminiate. Sommariamente sono quindi state rimosse 140 linee di codice.

3.5 Risultati

Per esplicitare al meglio i risultati ottenuti è stata costruito un grafico a barre che mostra i miglioramenti attuati. In figura 3.1 infatti, si possono osservare la differenza tra il prima e il dopo dell'applicazione della libreria dal punto di vista delle quantità di linee di codice dei file interessati.

[5] https://github.com/DanySK/Template-for-Gradle-Plugins/pull/708

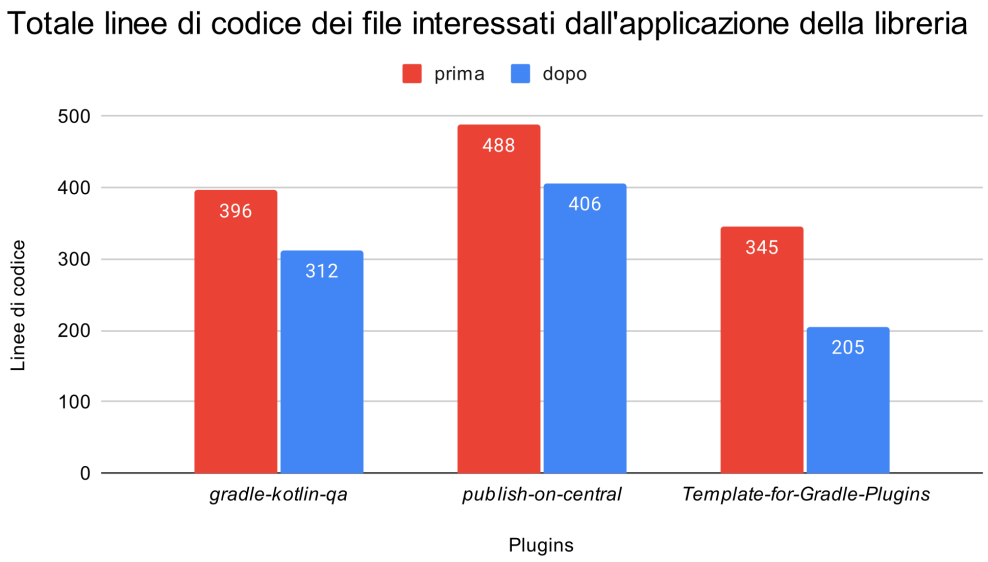

Figura 3.1: *Grafico a barre riportante la quantità di linee di codice interessate nell'applicazione della libreria.*

Conclusioni

Il progetto di tesi si è basato sull'idea di aiutare altri sviluppatori a collaudare il proprio software in maniera più semplice.

La progettazione di questa libreria mi ha permesso di provare cosa significa essere responsabile dello sviluppo di un sistema vero e proprio che sarà effettivamente utilizzato da utenti finali, per svolgere il loro lavoro quotidiano.

Infatti personalmente credo che aver intrapreso questo percorso di tesi mi abbia permesso di sviluppare un senso di responsabilità in più rispetto ad un percorso di tesi puramente teorico ed esplorativo.

Considerando gli sviluppi futuri, la libreria potrebbe essere sicuramente migliorata, in termini di qualità del codice, e aggiornata aggiungendo nuove funzionalità opzionali. Infatti, come detto in precedenza la libreria è stata pensata appositamente per essere migliorata grazie alla comunità di sviluppatori open source di GitHub. Per questo motivo è possibile aggiungere tutti i checker che si vuole.

Personalmente, nonostante le eventuali migliorie che si potrebbero apportare al progetto, mi ritengo soddisfatto di essere stato in grado di sviluppare un progetto che rimanga pubblico ed utilizzabile da chiunque voglia e sicuramente farà parte del mio bagaglio culturale.

Bibliografia

[1] Adetokunbo AA Adenowo and Basirat A Adenowo. Software engineering methodologies: a review of the waterfall model and object-oriented approach. *International Journal of Scientific & Engineering Research*, 4(7):427–434, 2013.

[2] Adel Alshamrani and Abdullah Bahattab. A comparison between three sdlc models waterfall model, spiral model, and incremental/iterative model. *International Journal of Computer Science Issues (IJCSI)*, 12 (1):106, 2015.

[3] Ahmad Jannat, Ul Hassan Abu, Naqvi Tahreem, and Mubeen Tayyaba. A Review on Software Testing and its Methodology. *i-manager's Journal on Software Engineering*, 13(3):32, 2019. doi: 10.26634/jse.13.3.15515. URL https://doi.org/10.26634/jse.13.3.15515.

[4] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182. IEEE, November 2016. doi: 10.1109/ict4m.2016.045. URL https://doi.org/10.1109/ict4m.2016.045.

[5] Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution Press, Portland, OR, December 2016. ISBN 978-1942788003.

[6] Alok Mishra and Ziadoon Otaiwi. Devops and software quality: A systematic mapping. *Computer Science Review*, 38:100308, 2020. ISSN 1574-0137. doi: 10.1016/j.cosrev.2020.100308. URL https://doi.org/10.1016/j.cosrev.2020.100308.

[7] Ravi Teja Yarlagadda. Devops and its practices. *International Journal of Creative Research Thoughts (IJCRT)*, 9(3):111–119, mar 2021. ISSN 2320-2882. URL https://ssrn.com/abstract=3798877.

[8] Akond Rahman, Asif Partho, David Meder, and Laurie A. Williams. Which factors influence practitioners' usage of build automation tools? In *3rd IEEE/ACM International Workshop on Rapid Continuous Software Engineering, RCoSE@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, pages 20–26. IEEE, 2017. doi: 10.1109/RCoSE.2017.8. URL https://doi.org/10.1109/RCoSE.2017.8.

[9] Adam L. Davis. *Gradle*, page 107. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-5058-7. doi: 10.1007/978-1-4842-5058-7_12. URL https://doi.org/10.1007/978-1-4842-5058-7_12.

[10] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015. doi: 10.1109/MS.2015.27. URL https://doi.org/10.1109/MS.2015.27.

[11] Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. The wonderful wizard of loc: paying attention to the man behind the curtain of lines-of-code metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020*, pages 146–156. ACM, 2020. doi: 10.1145/3426428.3426921. URL https://doi.org/10.1145/3426428.3426921.

[12] Gurdev Singh, Dilbag Singh, and Vikram Singh. A study of software metrics. *IJCEM International Journal of Computational Engineering & Management*, 11(2011):22–27, 2011.

Ringraziamenti

Vorrei ringraziare tutti coloro che mi hanno portato a raggiungere quest'obiettivo della mia vita, grande o piccolo che si possa ritenere.

Un grazie va ai miei genitori e alla mia famiglia che mi hanno sempre sostenuto e motivato in tutti i modi possibili e in tutte le scelte importanti del mio percorso di vita.

Un grazie va al relatore e a tutti i professori che con la loro passione mi hanno insegnato a ragionare come fa un vero ingegnere per costruire sistemi brillanti e di spettacolo.

Un grazie va al mio migliore amico, nonché coinquilino, nonché collega, per aver condiviso successi e sacrifici raccolti durante gli anni, nei progetti e nello studio.

Un grazie alla mia compagna, spero per la vita, di essermi stata il più vicino possibile nonostante la lontananza e di avermi sempre tirato su il morale quando ne avevo bisogno.

Grazie a tutti.