ScalaQuest

LSS Report

Thomas Angelini Jacopo Corina Francesco Gorini
Riccardo Maldini Filippo Nardini

Indice

Capitolo 1

Introduzione

Il progetto ScalaQuest si pone come obiettivo quello di realizzare un framework per permettere l'implementazione di giochi del genere **Interactive Fiction** (come ad esempio Zork), nei quali il giocatore può utilizzare comandi di testo per influenzare l'ambiente e proseguire nel gioco.

Esso dovrà in primis fornire una libreria, tale da permettere la **creazione di storie giocabili** da utenti terzi, tramite un API facilmente accessibile.

Dovrà inoltre fornire una **piattaforma per l'esecuzione delle storie**, basata su un'interfaccia da linea di comando. Questa permetterà ad utenti terzi di interagire con le storie create precedentemente, modificando lo stato nel gioco e avanzando man mano nella storia.

Il progetto è stato ideato per essere oggetto di esame in maniera mutuata per i corsi di PPS e LSS. A tale scopo, fin dalla definizione delle fondamenta del progetto si è posta particolare attenzione nell'adozione di una metodologia tale da integrare le peculiarità di entrambi i corsi. Il report di PPS descrive estensivamente gli aspetti relativi allo stesso corso, mentre nel report corrente vengono approfonditi nel dettaglio gli aspetti inerenti al corso di Laboratorio di Sistemi Software, tra cui:

- Il processo di sviluppo e di design, incentrato su un approccio di tipo Domain Driven;
- Le pratiche DevOps poste in atto.

In tutte le fasi il progetto è stato fortemente influenzato da questi aspetti, al quale è stato dedicato un ammontare di ore di lavoro equivalente a quello di PPS, considerando oltre alla mera implementazione anche il lavoro di approfondimento e studio delle pratiche, in parte nuove per i membri del team.

1.1 Codice sorgente e documentazione aggiuntiva

Complessivamente, i sorgenti di progetto consistono nell'insieme di repository GitHub parte dell'organizzazione ScalaQuest.

Tutti i sorgenti sono resi disponibile sotto **licenza MIT**, in quanto chiara, breve e concisa. Non vengono poste particolari limitazioni riguardo la consultazione e il riuso da parte di terzi del software fornito.

Ulteriori informazioni, documentazione e guide possono essere reperite a partire dal sito di progetto.

Capitolo 2

Aspetti di Domain Driven Design

Sin dalle prime iterazioni di progetto, particolare attenzione è stata posta nell'utilizzo dell'approccio Domain Driven Design. Nella pratica, una board collaborativa **Miro** è stata utilizzata per mettere nero su bianco idee e concetti base. È bene sottolineare che oltre ai costrutti richiesti dal DDD, la board contiene anche molteplici sketch che hanno portato il gruppo alla definizione di vari componenti.

2.1 Knowledge Crunching

Prima ancora di mettere mano all'architettura di progetto, per diverse settimane sono state effettuate sessioni di knowledge crunching che hanno visto la partecipazione di tutti i membri del team. Scopo di queste sessioni era lo studio del dominio applicativo, e la definizione di requisiti tali da guidare lo sviluppo in un'ottica DDD.

In primo luogo si sono andati a delineare in linea di massima i casi d'uso, per poi definire Ubiquitous Language e Bounded Context. Alla prima bozza degli stessi sono seguiti raffinamenti successivi, fino alla definizione di una struttura quanto più precisa e dettagliata.

Dal documento di Scrum Overview allegato in appendice è possibile individuare chiaramente come il primo Sprint tracciato (quello conseguente all'approvazione del progetto da parte del prof. Viroli) è stato interamente dedicato a questa fase. È però importante sottolineare come il lavoro di knowledge crunching sia iniziato ben prima, seppur con minore impegno, già dagli inizi di dicembre, ovvero dalla data di sottomissione dello stesso.

2.2 Ubiquitous Language

Di particolare importanza si è rilevata l'individuazione di un Ubiquitous Language originato dai concetti base del progetto. In {fig. 2.1} viene riportata la versione finale, che comprende tutti i concetti principali. Sulla board Miro è disponibile una versione dello stesso nel quale viene ampliata la descrizione di ogni termine.

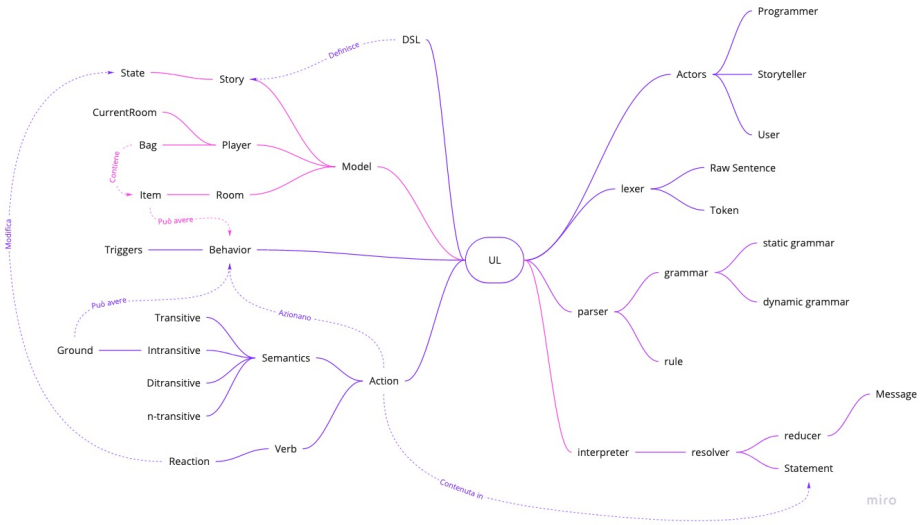

Figura 2.1: Ubiquitous Language del progetto.

2.3 Individuazione dei requisiti e dei casi d'uso

Nell'ambito di progetto sono stati individuati due principali attori, tali da interagire con lo stesso. Sulla base della loro definizione, sono stati quindi individuati vari casi d'uso:

- **Storyteller**: rappresenta l'attore in grado di creare delle storie giocabili. Questo è di fatto un programmatore che usufruisce del framework, e si assume quindi che abbia delle conoscenze di programmazione Scala. La creazione della storia consiste nella definizione delle `Room` e degli `Item` ad essa associati (includendo come parte di questa interazione la descrizione di *come* tali entità reagiscono ai comandi utente), e alla definizione dei verbi che comporranno la grammatica di una specifica storia;
- **User**: il termine indica l'attore che usufruisce della storia giocabile. Esso interagisce con il sistema immettendo comandi testuali, e consultandone l'output risultante.

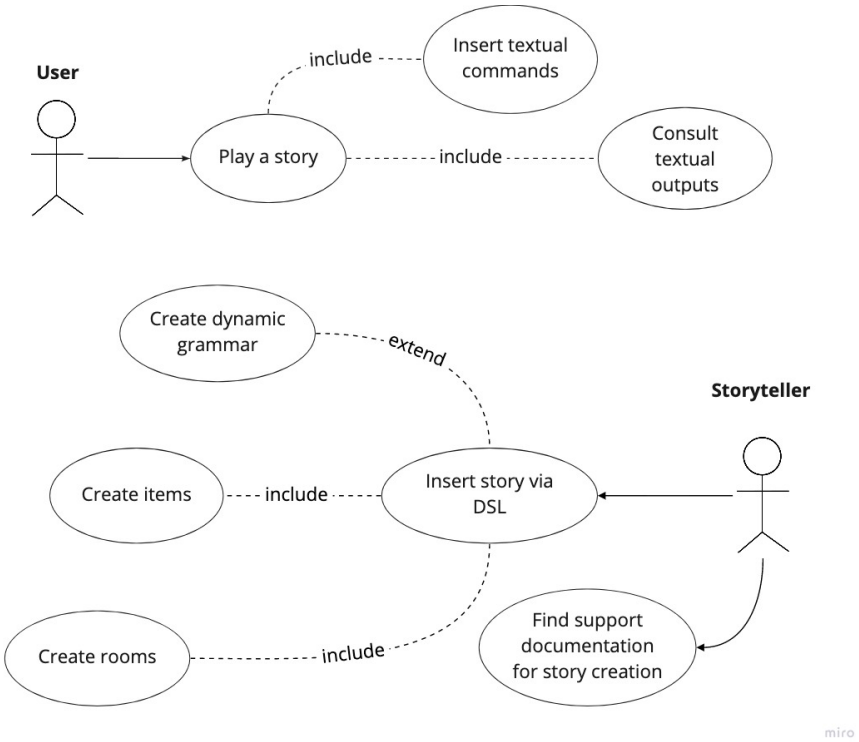

Figura 2.2: Diagramma dei casi d'uso del progetto.

2.4 Bounded context e Context map

A seguito dell'individuazione dei casi d'uso, si è andata a espandere l'analisi al fine di individuare i principali bounded context associati al progetto.

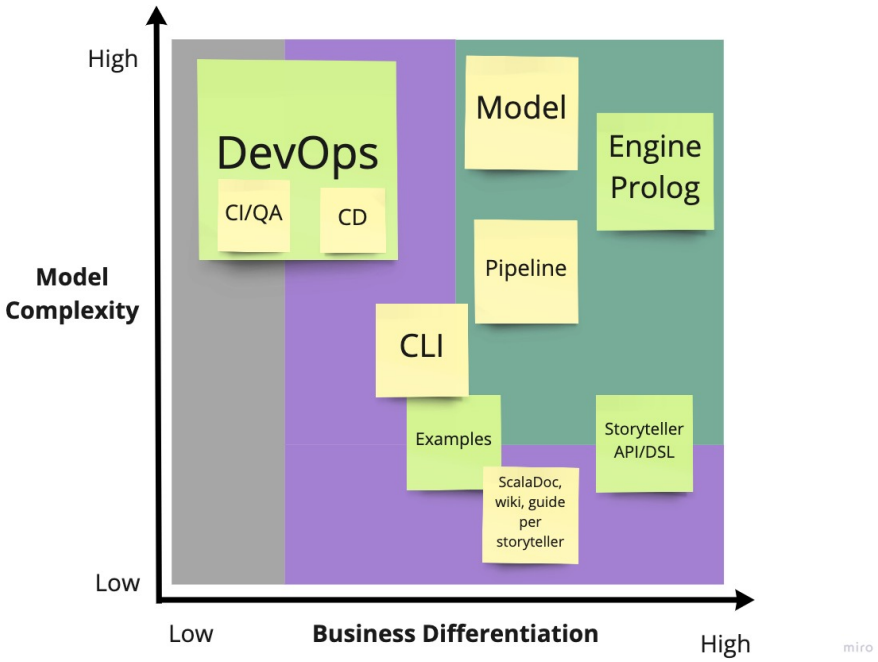

Figura 2.3: Analisi dei bounded context di progetto.

L'immagine riporta i principali bounded context, posti nel grafico in base alla complessità di modellazione degli stessi e all'importanza per il business. Si è intesa quest'ultima misura come la rilevanza di tale context, dal punto di vista di user e storyteller. Dal grafico si può evincere anche come le operazioni DevOps siano state elevate a vero e proprio bounded context: in ottica di effettuare un progetto di esame per LSS, esso rappresenta un vero e proprio requisito, ad alta complessità. L'utente finale, inteso come storyteller/user, può percepire da tali operazioni benefici indiretti (es. nella velocità delle release, nella qualità dell'API).

Sulla base di questa analisi preliminare, si è andata quindi a definire la context map, mostrata nell'immagine successiva.

Nello specifico, si è andato a accorpare quelli che erano stati individuati come bounded context di primaria importanza, in un unico **Core** bounded context. Questo in quanto, concettualmente, rappresentano moduli strettamente collegati.

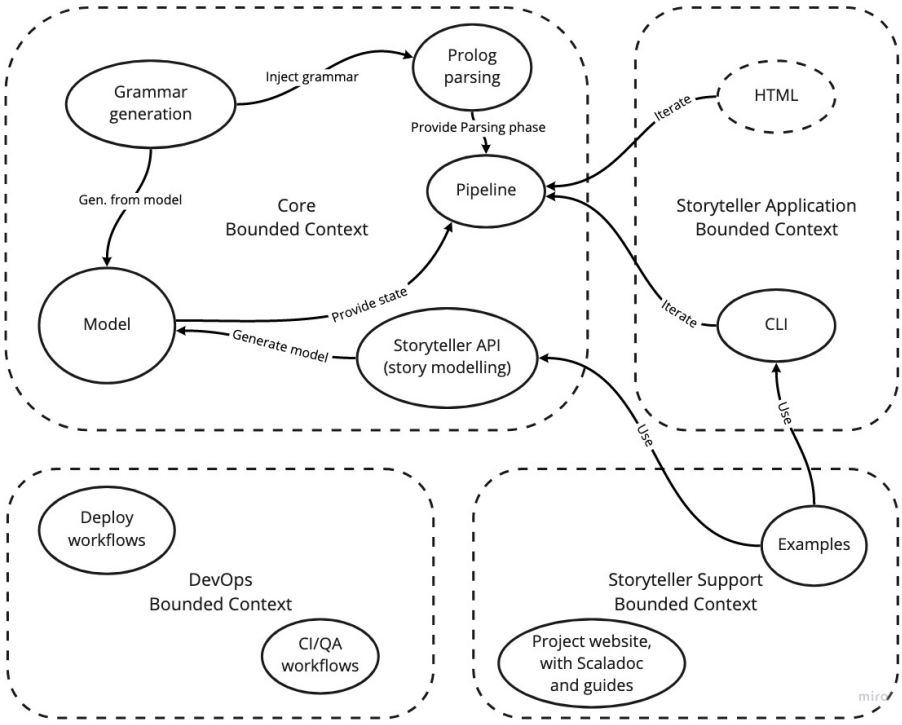

Figura 2.4: Context map di progetto.

Il bounded context **Storyteller Application** include ciò che concerne l'implementazione di vere e proprie UI per l'interazione con l'utente. HTML è stato rappresentato come tratteggiato, in quanto rappresenta un elemento da valutare in corso d'opera.

Storyteller Support include tutto ciò che concerne il supporto per lo storyteller alla costruzione della propria storia. In una libreria di queste dimensioni, fornire della documentazione di supporto diventa infatti un requisito di primaria importanza.

Infine, è stato definito un bounded context anche per ciò che concerne le pratiche **DevOps**. Graficamente, essi sono sono collegati agli altri bounded context. Ma non per il fatto di non influenzare gli altri; anzi, i collegamenti non sono rappresentati per il semplice fatto che il primo contiene degli elementi per loro natura pervasivi, che influenzano in maniera indiretta a tutti gli altri bounded context.

Capitolo 3

Gradle e struttura multi-progetto

Una volta definiti i bounded context, si è proseguito andando a definire l'architettura di progetto. È stata predisposta la repository di base di progetto, dal nome PPS-19-ScalaQuest. È qui che risiede il sorgente alla base dei principali moduli.

Si è deciso di utilizzare il tool di build automation **Gradle** per strutturare il progetto. Pur non essendo pensato primariamente per il linguaggio Scala, Gradle fornisce supporto per lo stesso.

Si è predisposta una **struttura multi-progetto** il più possibile aderente all'analisi DDD effettuata. Sono stati definiti i seguenti sotto-progetti:

- **core**: modulo che va a rappresentare di fatto i bounded context core di progetto. Esso è stato pensato infatti per definire specifiche alla base del model, della pipeline di progetto, la definizione del'API per lo storyteller, e il motore semantico del gioco per l'interpretazione dei comandi;
- **cli**: modulo che va a mappare il bounded context CLI. Rappresenta una "piattaforma" sulle quali giocare le storie, basata su un'implementazione a linea di comando. A livello pratico, CLI eredita come dipendenza il modulo core, permettendo allo storyteller di iniziare a creare storie importando il solo modulo cli.
- **examples**: sono stati definiti diversi moduli che consistono di fatto in delle storie di esempio, giocabili da un'utente finale.

3.1 Strategia basata su convention plugin

Ogni modulo necessita di differenti plugin Gradle per poter funzionare correttamente. Il dettaglio dei plugin specifici richiesti da ogni modulo viene trattato

nei capitoli successivi.

Di particolare interesse è invece una scelta architetturale che ha permesso di condividere tra vari insiemi di sub-projects plugin e configurazioni comuni. Si è infatti deciso di sfruttare una strategia fortemente raccomandata da Gradle nella sua documentazione, basata sui **convention plugins**.

Questa consiste nella definizione di vari plugin custom all'interno della directory standard `buildSrc`, ognuno comprendente configurazioni comuni a insiemi di sub-project. Una volta definiti, è quindi possibile includere tutte le configurazioni comuni semplicemente includendo all'interno dei singoli sub-project i convention plugin richiesti. In particolare:

- `scalaquest.common-scala-conventions.gradle.kts`: definisce le configurazioni comuni a tutti i sotto-progetti basati su linguaggio Scala (di fatto, tutti i sub-project). Questo comprende quindi i plugin Scala e la sua configurazione, scalatest e scoverage per la gestione dei test, un plugin per operazioni di lint-formatting del codice, varie opzioni comuni di configurazione del compilatore Scala, un plugin per il semantic versioning basato su Git;
- `scalaquest.libraries-conventions.gradle.kts`: definisce le configurazioni comuni a tutti i sotto-progetti che vanno a comporre una libreria Scala. Il plugin a sua volta importa il plugin `common-scala-conventions`, rendendo possibile configurare i moduli `core` e `cli` importando solamente `libraries-conventions`. Comprende il plugin java-library, la configurazione Scoverage specifica per le librerie, e la configurazione necessaria per la pubblicazione su Maven Central.
- `scalaquest.examples-conventions.gradle.kts`: definisce le configurazioni comuni a tutti i sotto-progetti esempio. Il plugin a sua volta importa il plugin `common-scala-conventions`, rendendo possibile configurare gli esempi importando solamente `examples-conventions`. Complende il plugin application, la configurazione di Scoversage specifica per gli esempi, e la configurazione necessaria a rendere gli esempi giocabili da linea di comando.

Capitolo 4

Modelli di sviluppo

Durante lo sviluppo del progetto, non si è adottato sempre lo stesso modello di sviluppo. Nelle prime fasi di progetto, durante le quali non si aveva del codice abbastanza stabile da essere "rilasciabile", si è seguito un approccio più flessibile e prototipale, denominato GitFlow, per poi evolvere il modello ad un più strutturato GitFlow.

4.1 GitHub Flow in fase embrionale

GitHub Flow è un modello di sviluppo ispirato a GitFlow, ma con alcune caratteristiche che lo rendono più flessibile e semplice da porre in atto.

Il modello richiede ad esempio che la versione stabile del software sia mantenuta su un branch `main` (o `master`), senza però la necessità di un branch `dev` parallelo. Allo stesso tempo, però, GitHub Flow suggerisce di organizzare il lavoro in `feature/*` branch, come in GitFlow, i quali confluiscono nel main a seguito della revisione di un secondo utente.

Alla luce di ciò, le prime iterazioni di progetto hanno presentato particolare flessibilità sulle modalità di modifica del codice. Le varie feature sono state sviluppate sui rispettivi `feature/*` branch, poi riversati nel `main` tramite pull request. Si è subordinato la chiusura di queste alla revisione da parte di un membro del team (solitamente, non appartenente allo stesso sub-team) e al passaggio di determitati workflow di CI e QA (descritti al {sec. **??**}).

4.2 GitFlow a regime

Una volta predisposta una codebase sufficientemente stabile, e una volta abilitati i workflow di Continuous Deploy, si è migrato al più strutturato modello **GitFlow**. Questo permette di avere nel branch `main` la versione ufficiale e stabile, sempre

associata a una release. A ogni push nel `main` deve corrispondere un tag, associato a sua volta a un numero di versione. La versione "di lavoro" del codice, stabile ma potenzialmente parziale, risiede nel branch `dev`.

I vari `feature/*` branch confluiscono ora tramite pull request in `dev`, con gli stessi vincoli formulati per modello precedente (controlli di CI obbligatori e revisione di un utente). In aggiunta, per una maggiore leggibilità e organizzazione del codice, si è adottata una precisa politica di merge, che prevede che queste pull request vengano chiuse tramite **squash and merge**.

Il `main` viene aggiornato tramite delle pull request sullo stesso originate da branch `release/X.Y.Z` (o `hotfix/X.Y.Z`), originati dal `dev`, dove con `X.Y.Z` si intende un numero di versione formulato secondo semantic versioning. Queste pull request presentano, oltre ai vincoli di validazione visti per le precedenti (controlli di CI e revisione di un membro del team), anche la necessità di presentare una coverage superiore al 75% nei moduli `core` e `cli`. Sono poi presenti degli accorgimenti di automazione ulteriori per la delivery automatizzata degli artefatti, e la gestione dei tag, indicati in {sec. 6}. Infine, una politica di merge ben precisa è adottata alla chiusura di queste pull request, le quali richiedono un **merge commit** che riporti, come commento del commit, un breve changelog[1].

[1] è necessario far presente che alcuni problemi sono incorsi tra la release 0.3.1 e 0.4.0, frangente nel quale, a seguito di un errore nelle politiche di commit, si è dovuto agire tramite rebase per preservare la repository. La storia tra questi due tag risulta quindi non perfettamente lineare.

Capitolo 5

Continuous Integration, Quality Assurance

Particolare sforzo è stato posto nel porre in atto workflow efficaci e automatizzati, in grado di garantire la qualità del codice, e la Contiuous Integration. Per la realizzazione di questi, si è utilizzato il tool di CI GitHub Actions, in parte per la prononda integrazione con GitHub, e in parte a causa delle recenti modifiche al piano di pricing in Travis CI. Questi a loro volta sfruttano delle funzionalità integrate all'interno del progetto grazie al tool di build automation Gradle.

5.1 Gradle e convention plugin

Uno dei primissimi accorgimenti posti in atto nel progetto ha riguardato dei controlli di qualità posti in atto sul codice Kotlin dei convention plugin stessi, e nei file `build.gradle.kts` dei vari sub-project. Lo scopo era quello di innalzare la qualità, prima ancora dell codebase Scala vera e propria, della stessa struttura Gradle a contorno del progetto.

A tal scopo, è stato abilitato il plugin **detekt**, un linter per Kotlin, posto in modalità strict: in questo modo, la build gradle fallisce nel caso in cui il codice Kotlin non rispetti determinati requisiti qualitativi, riportati in maniera dichiarativa all'interno del file `buildSrc/config/detekt.yml`.

Inoltre, per facilitare l'aggiornamento di dipendenze e plugin, è stato adottato il plugin **refreshVersions**, che consente di estrapolare le versioni di dipendenze e plugin Gradle in un file separato `versions.properties`, permettendo l'aggiornamento automatizzato delle stesse.

A tal proposito, a livello di organizzazione è stato definito un bot, denominato **dependabot** (nome ispirato al sistema di GitHub per l'aggiornamento delle dipendenze). Questa altro non è che una semplice repository con un workflow

schedulato, eseguito automaticamente ogni notte, per rilevare all'interno delle varie repository di progetto eventuali dipendenze non aggiornate, e generando automaticamente una pull request nella quale si va ad aggiornare tale dipendenza. Tale bot non fa altro che sfruttare il bot esistente UpGradle, configurandolo appositamente per agire all'interno dell'organizzazione.

5.2 Framework di test e soglie di coverage

I test sono portati avanti tramite il framework **ScalaTest**, seguendo lo stile di test **WordSpec**. Integrare ScalaTest all'interno del progetto non si è rivelato banale. A seguito di varie ricerche, si è deciso di integrarli tramite il plugin **ScalaTest di Maiflai**, un plugin che integra e configura in maniera pressoché trasparente ScalaTest, basato su jUnit 5.

In aggiunta a questo, è stato utilizzato un secondo framework di test, per rendere possibile testare il modulo `cli`. Basandosi infatti questo sulla libreria funzionale **ZIO**, il test dello stesso può essere effettuato solo con un framework apposito basato sempre su jUnit, denomitato **zio-test-junit**. L'omonima dipendenza è stata quindi aggiunta al `build.gradle.kts` dello specifico sub-project.

Infine, si è reso necessario trovare un modo per poter gestire i controlli di coverage. È infatti noto che Jacoco, uno dei tool più diffusi per i controlli di coverage su JVM, mal si adatta ai controlli su sorgente Scala. Jacoco opera infatti a livello bytecode, andando a coprire del codice autogenerato da Scala, e che può portare a stime di coverage del tutto sballate. Lo stato dell'arte per la messa in atto di controlli di coverage Scala con Gradle passa per l'utilizzo di plugin dedicati che tengono conto di queste caratteristiche, come **Scoverage di Maiflai**. Questo permette di generare, tra gli altri, report di coverage in formato html, oltre ad esporre un task `:scoverageCheck`, che permette di far fallire la build in presenza di coverage più bassa di una determinata soglia. Si è quindi installato nel progetto questo plugin, andando anche a configurare una soglia di coverage mandatoria del 75% per i moduli `core` e `cli`.

5.3 Lint e code style

Particolare attenzione è stata posta anche alla qualità del codice e allo stile dello stesso, definendo una serie di constraint atti ad innalzare la coesione stilistica del codice Scala tra le varie sezioni del progetto.

Sono presenti molteplici alternative in grado di gestire funzionalità di linting e styling di codice Scala tramite Gradle. Si è deciso a tal scopo di utilizzare il plugin **spotless**: questo aggiunge al progetto vari task per lo styling automatico del codice (`:spotlessApply`) e per il check dello stesso (`:spotlessCheck`), supportando al contempo molteplici linguaggi di programmazione tramite tecniche differenti. Per Scala, Spotless sfrutta internamente `scalafmt`, un tool per lo

styling del codice Scala. Le regole di styling applicate sono accessibili in un formato dichiarativo all'interno del file `.scalafmt.conf`.

5.4 SonarCloud

Un ulteriore strumento posto in atto per innalzare la qualità del codice e certificarla è **SonarCloud**. Questo tool permette di porre in atto controlli automatizzati sul codice, estraendo varie metriche qualitative riguardo la codebase, legate a mantenibilità, coverage, debito tecnico, duplicazione e molto altro. Lo strumento fornisce inoltre una dashboard pubblica che ne raccoglie le principali metriche, accessibile da qua.

Di particolare rilevanza per il progetto è stata la funzionalità **quality gate**: SonarCloud integra infatti un bot, che ad ogni push all'interno di una pull request in direzione di branch stabili, effettua un controllo di CI, fallendo nel caso in cui le metriche rilevate non superino delle soglie preimpostate.

Allo scopo di configurare correttamente SonarCloud, si è reso necessario aggiungere un plugin al progetto, denominato **Sonarqube**. Questo rappresenta uno strumento cosiddetto di "scanner" per SonarCloud, andando ad estrarre in maniera più mirata le metriche. Il plugin fornisce il task `:sonarqube`, che va eseguito in CI nel momento in cui si voglia eseguire un controllo di quality gate.

5.5 Il workflow CI

Allo scopo di porre in atto la maggior parte dei controlli citati in precedenza, si è reso necessario definire un apposito workflow GirHub Actions, denominato semplicemente `ci.yml`. Questo viene eseguito ad ogni push e pull request effettuata in direzionde dei branch `main` e `dev`. Il workflow è stato organizzato in differenti job, i quali agiscono in maniera completamente parallela. Ciò permette di otttenere una logica di tipo fail-fast, desiderabile nelle routine di CI. I job eseguiti sono i seguenti:

- **Build**: responsabile di verificare che il software venga buildato correttamente. Il task `:build` viene in questo caso "sezionato" nei suoi due sotto-task `:assemble` e `:check`, permettendo una più facile interpretazione del log di GH Actions in caso di fail. Il job viene eseguito su una matrice di sistemi operativi, mentre si è ritenuto non di interesse testare la build su versioni differenti di Java (avendo posto come necessario il solo supporto a Java 11 con Scala 2.13);
- **Lint**: responsabile della correttezza stilistica del codice. Al suo interno, viene semplicemente eseguito il task `:spotlessCheck`;
- **Coverage**: controlla che le soglie di coverage impostate vengano rispettate, tramite il task `:checkScoverage`.

5.6 Il workflow Opt-in CI

Ulteriore controllo di CI è stato posto tramite il workflow **Opt-in CI**. Questo permette di attivare determitate funzioni di CI, come lo style check, la build, o il controllo di coverage, a partire dai branch `feature/*`. Viene definito "opt-in" in quanto di base questi controlli sono disabilitati. Vengono di fatto utilizzati solo per dei test, e possono essere invocati includendo, nel contenuto del commit che ha generato il push nel branch, i tag `[lint]`, `[build]`, `[coverage]`.

Capitolo 6

Continuous Delivery e Deployment

Tra gli obiettivi di progetto è stato posto fin da subito quello di realizzare dei processi efficaci di Continuous Delivery e Deployment degli artefatti. Nonostante ciò, a differenza di quanto fatto per i workflow di CI, quelli di CD non sono stati realizzati immediatamente. Questo in quanto nei primi Sprint di progetto non si aveva del codice abbastanza stabile da poterne ricavare degli artefatti concretamente utilizzabili. I workflow di Continuous Delivery e Deployment sono stati predisposti durante il passaggio da GitHubFlow a GitFlow, e hanno subìto diverse modifiche lungo lo sviluppo del progetto.

6.1 Il workflow Release

Il primo e più importante workflow per il deploy degli artefatti è denominato **Release**. Questo viene lanciato ogniqualvolta viene chiusa con successo (ovvero generando un evento di merge) una pull request in direzione del `main`, a partire da un branch `release/X.Y.Z`. Tale workflow:

1. Inferisce il nome del tag da associare alla release, a partire del nome del branch di provenienza, e crea un annotated tag in corrispondenza del commit di merge;
2. Genera una release GitHub nell'apposita sezione Releases, ponendo all'interno di questa gli artefatti necessari (jar per le librerie `core` e `cli`, delle distribution in formato `zip` e `tar.gz` per gli esempi, il file `README.md`);
3. In parallelo al punto 2, effettua una pubblicazione su Maven Central;
4. Una volta completati i punti 2 e 3, viene generata una pull request dal `main`, diretta al branch `dev`. Ciò permette di integrare nel branch di sviluppo le

eventuali modifiche occorse all'interno dei branch `release/X.Y.Z`. Su di questi infatti si è delle volte agito con delle modifiche minori.

5. Una volta completati i punti 2 e 3, vengono generati ScalaDoc e report di coverage per i moduli `core` e `cli`, e resi disponibili nello spazio web del progetto.

6.2 Il workflow Prerelease

Oltre al workflow di release principale, si è predisposto un secondo workflow, che va a creare delle release GitHub ogniqualvolta viene effettuato un push nel branch `dev`. Tali release vengono marcate come non perfettamente stabili, appunto con il marcatore prerelease, da cui il nome del workflow. Particolarità di questa routine è che non necessita di definire manualmente il nome della release: questa viene generata in automatico, tramite il plugin `git-sensitive-semantic-versioning`, a partire dall'ultimo tag disponibile.

Ricapitolando, il workflow:

1. Inferisce il nome del tag da associare alla prerelease, in modo del tutto automatico, tramite il plugin Gradle `git-sensitive-semantic-versioning`. Un task personalizzato permette di estrre il numero di versione e salvarlo su un file; il contenuto viene quindi prelevato, salvato come variabile d'ambiente, e viene creato un annotated tag in corrispondenza del commit di merge;
2. Genera una release GitHub nell'apposita sezione Releases, con il flag prerelease abilitato, ponendo all'interno di questa gli artefatti necessari (jar per le librerie `core` e `cli`, delle distribution in formato `zip` e `tar.gz` per gli esempi, il file `README.md`);
3. In parallelo al punto 2, effettua una pubblicazione su Maven Central.

6.3 Maven Central

Si è deciso di dedicare una sezione a parte per approfondire il procedimento che ha portato alla pubblicazione delle librerie di progetto sul repository pubblico **Maven Central**.

La predisposizione del delivery in una repository pubblica è stata prevista sin dall'inizio nel progetto. Anche solo la scelta di porre tutte le repository di progetto sotto una stessa organizzazione è stata eseguita in quest'ottica, così da poter ottenere il dominio `scalaquest.github.io`, e utilizzare `io.github.scalaquest` quale groupId per il rilascio.

L'attuazione di questo obiettivo però è stato demandato a fasi più avanzate di progetto, individuando tale possiblità come requisito da valutare in corso d'opera.

Ci si è iniziati a muovere in tal senso durante la migrazione da GitHubFlow a GitFlow. In questo frangente si è aperto un **ticket presso Sonatype**, richiedendo la possibilità di pubblicare sotto il groupId `io.github.scalaquest`.

Tale processo è stato completato abbastanza velocemente. Lo sviluppo di questa soluzione ha però subito a questo punto una fase d'arresto, legata principalmente alla complessità nella configurazione del plugin Gradle `maven-publish` e alla firma degli artefatti, in particolare in ambiente CI. I workflow di release sono quindi stati inizialmente realizzati senza la possibilità di pubblicare su Maven.

Solo più avanti, con il raggiungimento di una relativa stabilità di progetto, si è deciso di approfondire e portare a termine questo task, concludendo la configurazione del plugin e dei workflow collegati.

Nella configurazione, una volta pubblicati gli artefatti, saranno i membri del team a concludere la release, effettuando il **close** o il **drop** della release, direttamente dall'interfaccia web **Nexus Repository Manager**, previa autenticazione.

Il team tiene a sottolineare che pubblicare su una repository pubblica popolare quale Maven Central è stata una grande fonte di soddisfazione per i propri membri.

Capitolo 7

Reports repository

Nelle prime fasi del progetto, il repository principale conteneva un ulteriore sotto-progetto denominato `reports`. All'interno di questo si erano andati a porre i report di LSS e PPS, oltre ai documenti a margine come lo Sprint Overview. La scelta di utilizzare un sotto-progetto è stata dettata dalla necessità di dare una struttura al codice, e alla possilità di utilizzare il plugin `Spotless` di Gradle anche per lo styling del Markdown, grazie al ai task `:spotlessCheck` per il controllo sul rispetto delle regole di stile, e `:spotlessApply` per l'applicazione automatica delle stesse.

Con l'aumentare con la complessità del progetto, però si è deciso di cambiare tale infrastruttura. Il sottoprogetto `reports` è stato quindi migrato in un repository del tutto slegato da quello principale, denominato **Reports**. Scelta è stata dettata anche allo scopo di costruire un'infrastruttura più complessa attorno ai report stessi.

Reports è un progetto Gradle, con abilitato il plugin `Spotless` per lo styling (con una configurazione dedicata per il Markdown), e i sorgenti della documentazione (in Markdown) contenuti all'interno della directory `/src`. Il modello di sviluppo adottato è ancora una volta GitFlow, in maniera del tutto equivalente nelle modalità a quelle del repository principale.

7.1 Continuous Integration

A guardia dei branch stabili, è stato posto un workflow di CI che va ad applicare `:spotlessCheck`, e fallisce nel caso in cui non vengano rispettate determinate regole di stile (come la lunghezza della riga posta a 80 caratteri).

7.2 Continuous Delivery e Deployment

Assolutamente peculiari sono invece i workflow di delivery e deployment. I file Markdown si prestano infatti molto bene ad essere convertiti in altri formati, utilizzando i tool adatti. Si è deciso quindi di fornire, per una consultazione più agevole, i report nei formati LaTeX PDF e in una versione web, autogenerati ad ogni nuova release. Ciò è stato possibile grazie a **Pandoc**, un tool che permette la conversione di file testuali in una moltitudine di formati differenti.

Con queste specifiche, si sono andati quindi a definire un workflow **Release** e uno **Prerelease**, lanciati con le stesse modalità descritte per il repository principale:

7.2.1 Il workflow Release

Release viene lanciato alla chiusura dei branch `release/X.Y.Z`. Effettua le seguenti operazioni:

1. Inferisce il nome del tag da associare alla release, a partire del nome del branch di provenienza, e crea un annotated tag in corrispondenza del commit di merge;
2. Genera una release GitHub nell'apposita sezione Releases. Tale release contiene i report di PPS, LSS e un file di appendice (oltre al file `README.md`), a partire dai file Markdown, tramite un'immagine Docker con integrata un'installazione Pandoc;
3. Genera una versione HTML dei report e del file di appendice, a partire dai file Markdown, tramite un'immagine Docker con integrata un'installazione Pandoc. Posiziona quindi tali file all'interno del branch `gh-pages` di progetto, e effettua automaticamente un commit su questo branch, integrando la versione aggiornata della documentazione. In questo modo, questa sarà accessibile dallo spazion web del progetto;
4. Una volta completati i punti 2 e 3, viene generata una pull request dal `main`, diretta al branch `dev`. Ciò permette di integrare nel branch di sviluppo le eventuali modifiche occorse all'interno dei branch `release/X.Y.Z`. Su di questi infatti si è delle volte agito con delle modifiche minori.

7.2.2 Il workflow Prerelease

Prerelease viene lanciato ad ogni push nel branch `dev`, e genera delle release marcate con il flag pre-release, con numero di versione generato automaticamente tramite `git-sensitive-semantic-versioning`. Effettua le seguenti operazioni:

1. Inferisce il nome del tag da associare alla prerelease, in modo del tutto automatico, tramite il plugin Gradle `git-sensitive-semantic-versioning`. Un task personalizzato permette di estrre il numero di versione e salvarlo

su un file; il contenuto viene quindi prelevato, salvato come variabile d'ambiente, e viene creato un annotated tag in corrispondenza del commit di merge;

2) Genera una release GitHub nell'apposita sezione Releases, con il flag prerelease abilitato. Tale release contiene i report di PPS, LSS e un file di appendice (oltre al file `README.md`), a partire dai file Markdown, tramite un'immagine Docker con integrata un'installazione Pandoc;

7.2.3 Configurazione di Pandoc

Per poter personalizzare l'output generato da Pandoc, sono stati utilizzati dei file configurazione Pandoc, accessibli dalla directory `/pandoc`. All'interno di questa si è andato a configurare il template LaTeX nativo di Pandoc, allo scopo di adattarlo al progetto. Per quanto rigaurda il template HTML, invece, si è creato e utilizzato un template custom, a partire da un fork del tamplate PandocBootstrap. Il template di progetto è accessibile al repository `scalaquest/PandocBootstrap`. Questo è stato posto come submodule del repository Report, così da avere a disposizione il template durante la generazione dei report in CI.

Capitolo 8

Conclusioni

In conclusione, possiamo affermare come team di ritenerci molto soddisfatti di quanto fatto in questo progetto. Rispetto a progetti trattati in precedenza dai singoli membri del gruppo, in questo si è posta un'attenzione particolare alla metodologia e all'organizzazione di progetto, cercando per quanto possibile di approcciare lo sviluppo con mentalità ingegneristica, pragmatica e strutturata.

Le sessioni di knowledge crunching iniziale si sono rivelate cruciali per la comprensione dei concetti alla base del progetto, e per l'organizzazione dello stesso. È stata una vera e propria soddisfazione poi vedere il proprio progetto pubblicato sulla repository pubblica Macen Central.

Come membri del team ci teniamo inoltre ad aggiungere che nessuno di noi aveva in precedenza lavorato ad un progetto con un'organizzazione e una metodologia strutturata. Due membri del team provengono poi da una triennale differente dal percorso degli altri (Riccardo Maldini e Francesco Gorini, UniUrb), la quale proponeva un approccio ai progetti ben più approssimativo. Una sfida che si è accettata di buon grado, coscienti delle potenzialità dei singoli.