

# Relazione Progetto

Mattia Matucci

25 settembre 2023

Il progetto prevede la realizzazione di un'applicazione per gestire la discografia dei musicisti raccolti nel database. L'applicazione deve permettere di salvare e cancellare musicisti dall'archivio e di modificare i loro dati. L'utente può gestire tali operazioni attraverso un'interfaccia grafica.

Una volta selezionato un musicista, l'utente può compiere le stesse operazioni di salvataggio, cancellazione e modifica sugli album che compongono la discografia del musicista selezionato.

Come si può vedere in figura 1, sia la gestione dei musicisti che quella degli album avviene nella stessa finestra dell'interfaccia grafica.

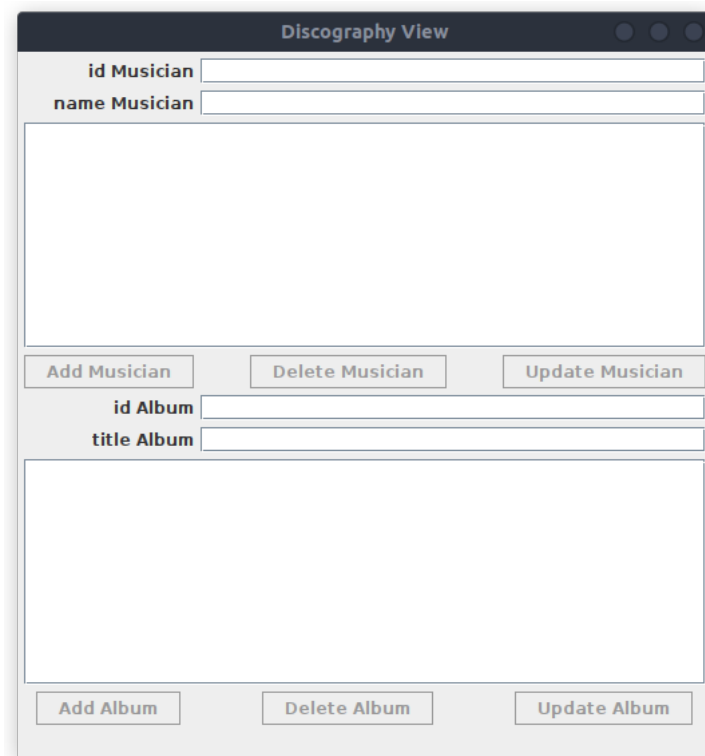


Figura 1: Interfaccia grafica

L'utilizzo di due singole entità è dovuto alla richiesta di mantenere semplice la struttura del progetto. Attraverso la figura 1 si può osservare che per i musicisti vengono memorizzati il nome e un identificativo, in maniera analoga vengono salvati il titolo e un identificativo per gli album.

Le due entità sono messe in relazione associando ai dati salvati per gli album anche un riferimento all'identificativo del musicista. Questa informazione viene recuperata automaticamente

dall'applicazione, poiché come già detto sopra la gestione degli album può avvenire soltanto dopo aver selezionato un musicista.

## 1 Tool, Frameworkg e Librerie usate

L'applicazione è sviluppata e testata con Maven, questo permette di costruire un processo automatico facilmente riproducibile per eseguire la build del codice e di non dover gestire le dipendenze transitive delle librerie usate. La storia del codice è archiviata in una repository remota su GitHub, grazie ciò è possibile avere a disposizione un Continus Integration server GitHub Actions per validare la build ogni volta che c'è un push sulla repository.

Oltre alla validazione della build il GitHub Actions può esser configurato per eseguire un'analisi più approfondita del codice: ad esempio per questo progetto in concomitanza di un push sul branch main oppure su una pull request, ho configurato che sia redatto un report di code coverage attraverso Jacoco e che siano eseguiti i mutation test con Pitest, impostando il successo della build se tutti i mutanti sono uccisi. I mutation test vengono eseguiti sulle classi che contengono la business logic, ossia il controller e le classi che gestiscono l'accesso al database, verificandoli solo sugli unit test.

Il report generato da Jacoco è inviato a Coveralls, una piattaforma che permette di archiviare tutti i risultati ottenuti e imporre condizioni per la validità della build. Per questo progetto ho imposto il raggiungimento del 100% di copertura nelle classi che contengono della logica. Nel file POM di Maven nella sezione di configurazione del plugin di Jacoco si possono selezionare le classi da escludere nel calcolo del code coverage.

Un altro tool per l'analisi della qualità codice è SonarCloud, che individua bug, problemi di sicurezza e manutenibilità del codice, attraverso la verifica di regole controllate staticamente. All'interno del file POM è possibile ignorare delle regole specificando anche le classi in cui devono essere ignorate.

In questo progetto sono ignorate le seguenti regole:

- per la classe DiscographySwingView quella della convenzione dei nomi delle variabili, visto che questi nomi sono generati dal tool con cui è stata costruita l'interfaccia grafica.
- per la classe DiscographySwingViewTest quella della scrittura di test senza un'asserzione, dal momento che la regola non riconosce AssertJ-swing

Tra le varie metriche analizzate da SonarCloud è presente il code coverager, dunque anche in questo caso è necessario inviare anche il report generato con Jacoco.

Sia Coveralls che SonarCloud sono dei tool che hanno la possibilità di fare fallire la build se non sono raggiunti determinati standard e di partecipare alla pull request lasciando dei commenti, documentando così la qualità del codice prima di fondere le nuove feature nel branch principale.

L'esecuzione di Pitest e Jacoco, insieme all'invio del report a Coveralls e SonarCloud, viene gestita automaticamente da GitHub Actions, eseguendo dei workflow configurati in dei file yml. Poiché la configurazione di questi file ha generato un pò di problemi, approfondirò la configurazione dei workflow nelle sezioni successive dell'elaborato.

Insieme ai workflow per Pitest, Coveralls e SonarCloud sono presenti quelli per verificare la build su i vari sistemi operativi. In particolare la build su Linux viene verificata usando diverse versioni di Java, a differenza di tutti gli altri workflow quelli su Linux sono eseguiti in concomitanza di ogni push su GitHub.

Il codice è sviluppato con Java 8 e testato con JUnit 4, ho scelto JUnit 4 perché attualmente per Junit Jupiter non esiste qualcosa di equivalente al runner di AssertJ Swing, guardando

online ho visto che attualmente la migliore soluzione è introdurre una classe d'utilità che svolga lo stesso funzione del runner cioè scattare un screenshot in caso di fallimento.

Per scrivere dei test facilmente leggibili ho usato AssertJ. Questa dipendenza non è espressa direttamente nel POM, poiché è già inclusa come dipendenza transitiva in AssertJ-Swing-JUnit, quest'ultima permette di scrivere test che sfruttano la fluent interface anche per l'interfaccia grafica.

Per inizializzare il SUT negli unit test ho usato Mockito, poiché gestisce automaticamente le dipendenze, definendo dei mock poi iniettati nella componente da testare. Una funzionalità fondamentale è quella che permette di usare componenti ancora da implementare, rendendo quindi più semplice scrivere test in isolation.

A partire dagli integration test la creazione del SUT è più difficile, data la necessità di usare almeno due componenti reali. Spesso la più grande problematica di questa fase è la gestione delle dipendenze, soprattutto se cicliche. In questo progetto ad esempio è presente una dipendenza ciclica tra il controller e la view, poiché ho applicato il pattern Model-View-Presenter.

Per semplificare l'inizializzazione dei SUT sia negli integration test che negli end-to-end test ho usato il framework Guice, che tramite reflection riesce a risolvere automaticamente le dipendenze, anche quelle cicliche. Il suo utilizzo è però strettamente limitato ai casi in cui non viene usato Mockito, perché in quei casi la gestione delle dipendenze avviene tramite iniezione di mock.

L'interfaccia grafica è stata definita con windowbuilder un tool che permette di creare interfacce grafiche per swing, generando anche il corrispondente codice in Java. Ho riscontrato vari problemi con le ultime versioni rilasciate di windowbuilder, ho risolto questi problemi usandone una versione più vecchia.

La repository è implementata usando il database non relazionale MongoDB. Questa scelta è ottima considerando la maggiore semplicità e velocità del database, ma ha il grosso svantaggio che la gestione dell'integrità dei dati tra le collezioni è affidata al programmatore, poiché non sono presenti strumenti per gestire concetti come chiavi esterne e cancellazioni a cascata.

I test che richiedono il database sono eseguiti in un container di Docker con un'istanza di MongoDB. Per semplificare la gestione dei container è stato usato il framework Testcontainer, che automaticamente si occupa di avviarli e arrestarli. L'utilizzo dei container e di Testcontainer, ha costretto ad affrontare scelte riguardo i workflow per MacOS e Windows.

Docker non è installato nell'ambiente virtuale di MacOS. Visto che la sua installazione richiede almeno 10 minuti, ho deciso di validare la build eseguendo solo gli unit test. La stessa scelta è stata fatta anche per il workflow di Windows, in questo caso però il problema è che nell'ambiente virtuale di Windows non è possibile eseguire i container di Linux, ma solo quelli di Windows per cui spesso non esiste il corrispondente su Testcontainer.

Nonostante la gestione dei container sia automatizzata l'avvio e il setup del database è piuttosto lento. Per questo motivo quando non è propriamente richiesto un'istanza reale di MongoDB viene usato in-memory database, che è sufficientemente affidabile per le operazioni di tipo CRUD e per piccole query. Preso atto della natura delle operazioni da implementare ho deciso di non scrivere gli integration test per verificare il comportamento delle repository con un'istanza reale di MongoDB.

Gli end-to-end test sono scritti con Cucumber, tale decisione è stata presa poiché ritengo che possa essere molto utile scrivere specifiche eseguibili in un linguaggio di alto livello. In questo modo il superamento degli end-to-end test equivale al soddisfacimento delle specifiche fissate inizialmente.

Infine nel progetto è usato Picocli per configurare facilmente l'applicazione tramite argomenti passati a linea di comando. Nel progetto la totalità dei parametri esprimibili riguarda la

configurazione del database. Per esempio si può specificare l'indirizzo del host, la porta esposta del server o il nome del database.

## 2 Cosa è stato implementato e scelte di design

Come ho già anticipato nell'introduzione il progetto prevede di realizzare un'applicazione per gestire i musicisti e la loro discografia, mettendo a disposizione dell'utente un'interfaccia grafica per eseguire le operazioni di salvataggio, cancellazione e modifica dei dati. Per costruire l'applicazione modulare ho applicato il pattern Model-View-Presenter, rappresentato in figura 2, implementando il layer d'accesso al database con il pattern Repository.

Scendendo maggiormente nel dettaglio ho implementato un'unica interfaccia grafica che interagisce con le repository, una per ciascuna entità, attraverso il presenter (controller). Poiché non ho pensato di creare più interfacce grafiche la scelta di definire un'unica interfaccia view, piuttosto che una per ogni entità, è parsa la migliore. Avendo definito solo una view ho ritenuto sufficiente creare un solo controller, che interagisca con la view e con le due repository.

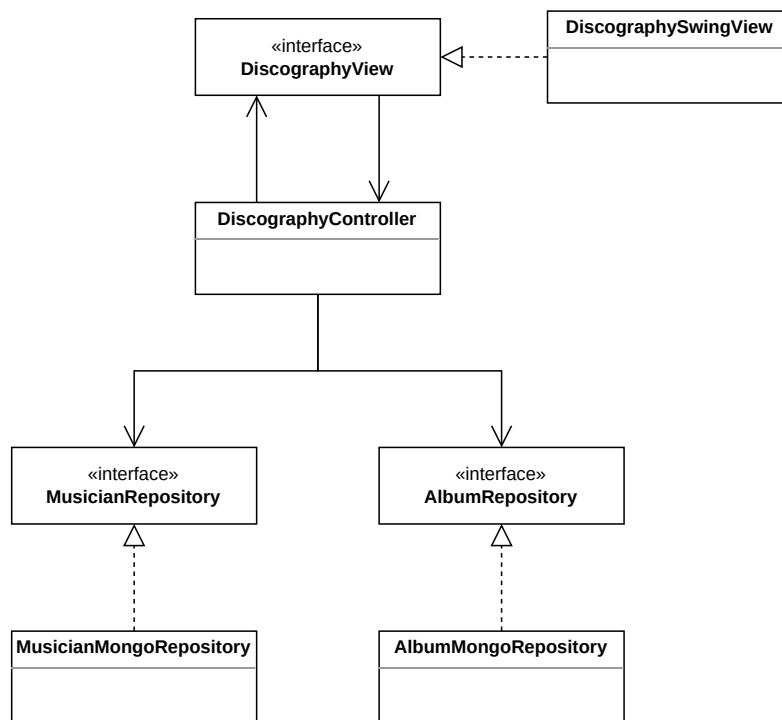


Figura 2: Schema UML del Model-View-Controller

Prima di procedere all'effettivo sviluppo dell'applicazione, ho definito le sue specifiche, ossia le funzionalità che deve possedere l'applicativo finale. Di seguito sono riportate alcune delle specifiche definite per questo progetto:

- Quando l'utente preme il bottone Add Musician, alla lista dei musicisti deve essere aggiunto un nuovo musicista con i dati specificati nei campi associati ai musicisti.
- Quando l'utente preme il bottone Delete Musician, l'elemento selezionato della lista dei musicisti deve essere rimosso, insieme a tutti i suoi album.

- Quando l'utente preme il bottone Update Musician, l'elemento selezionato della lista dei musicisti deve essere aggiornato
- Quando l'utente seleziona un musicista dalla lista dei musicisti, deve essere mostrata la sua discografia.
- ...

Ho scritto le specifiche del progetto con Cucumber, perché posso usare un linguaggio di alto livello ed eseguirle. Le specifiche vengono definite attraverso degli scenari, il cui soddisfacimento equivale al superamento delle asserzioni fatte negli step definiti dai singoli scenari. Purtroppo definire all'inizio del progetto dei test che potranno essere superati una volta che tutte le componenti sono sviluppate, ha come conseguenza il fallimento della build. Per ovviare a ciò si può specificare il flag `dryRun` a `true`, in questo modo Cucumber determina il successo di uno scenario se a ciascun step è associata un metodo, anche se non implementato.

Le singole componenti dell'applicazione sono state sviluppate applicando il TDD, tale processo è stato alternato alla scrittura degli integration test ogni qual volta fosse possibile. Analizzando il log dei commit si può osservare che l'ordine di sviluppo delle componenti:

- `DiscographyController`
- `MuscianMongoRepository`
- `AlbumMongoRepository`
- `DiscographySwingView`

Ho iniziato a implementare l'applicazione dalla business logic, partendo dal controller, perché durante il suo sviluppo si definiscono le interfacce delle repository e dalla view. Dopo di che ho implementato le interfacce associate alle repository, testandole con un in-memory database. Il passaggio successivo è stato testare l'integrazione tra il controller e le repository usando `Testcontainer` per avere un'istanza reale di MongoDB. In questo caso per avviare il SUT non è stato usato Guice, dal momento che per inizializzare il controller ho creato un mock per la view.

Successivamente ho implementato l'interfaccia grafica, osservando il log dei commit si può osservare che ho diviso il suo sviluppo in più fasi:

- Disegno dell'interfaccia grafica con `windowbuilder`.
- Implementazione attivazione dei pulsanti e verifica del loro comportamento una volta premuti.
- Implementazione dell'interfaccia `DiscographyView`.

Implementata l'interfaccia grafica, ho verificato la sua integrazione con il controller, usando un in-memory database, verificando che le azioni compiute dall'utente generino sull'interfaccia grafica l'output desiderato.

Dopo di che ho testato l'integrazione di tutte le componenti, verificando che le azioni compiute dall'utente abbiano un effetto sull'istanza reale del database. Questi test non si considerano end-to-end test dal momento che la verifica non avviene sull'interfaccia grafica.

Prima di procedere all'implementazione degli scenari, ho configurato l'avvio dell'applicazione con un modulo default di Guice configurando la possibilità di usare Picocli per passare argomenti da linea di comando. Negli integration test l'inizializzazione del SUT avviene sovrascrivendo il

modulo di Guice definito per l'applicazione. Parlerò meglio di questa configurazione nelle sezioni successive. Al termine di tutto ciò ho implementato ogni step definito con Cucumber, usando Testcontainer per inizializzare il database.

Gli errori generati dall'applicazione vengono mostrati in fondo all'interfaccia. Per ambedue le entità esistono i seguenti errori:

- Duplicazione id, ossia esiste già un altro elemento che la stessa chiave
- L'elemento selezionato nella lista non è presente nel database, in aggiunta viene rimosso dalla lista.

Ho già specificato nell'introduzione che per connettere le due entità ho inserito un riferimento al identificativo del musicista nella classe album. Purtroppo MongoDB non prevede un meccanismo di cancellazione a cascata, questo crea un problema riguardo l'integrità dei dati, che deve essere gestito manualmente dal programmatore. Si cancellano tutti gli album di un musicista, quando questo viene rimosso, perché ogni album deve essere associato a un musicista memorizzato nel database. La cancellazione a cascata si effettua quando viene cancellato un musicista oppure nel momento in cui viene lanciato l'errore per il musicista selezionato.

### 3 Esecuzione del codice

Di seguito sono riportati i comandi per eseguire la build del progetto senza errori con Maven al variare delle versioni Java

- Java 8:  
`$ ./mvnw verify`
- Java 11:  
`$ ./mvnw verify -DmyArgLine="--illegal-access=deny --add-opens java.base/java.util=ALL-UNNAMED"`
- Java 17:  
`$ ./mvnw verify -DmyArgLine="--add-opens java.base/java.util=ALL-UNNAMED"`

Per assicurare la riproducibilità della build, ho utilizzato il wrapper di Maven, in questo modo oltre ad aver fissato le versioni delle dipendenze e dei plugin ho potuto fissare anche la versione stessa di Maven.

### 4 Parti più interessanti

In questa sezione sono trattati in modo più dettagliato gli aspetti più interessanti del progetto.

#### 4.1 Test Database Layer

Il layer d'accesso al database è testato esclusivamente con degli unit test usando un in-memory database. Ho preso questa decisione dopo aver analizzato la tipologia di operazioni che devono essere svolte. Considerato che si tratta di piccole query o operazioni CRUD, posso ritenere l'in-memory database sufficientemente affidabile. Questa scelta ha inoltre il vantaggio di ridurre i tempi d'esecuzione, poiché non deve avviare un server. Durante gli integration test quando non è stato necessario usare un istanza reale di MongoDB, ho usato un in-memory database. Negli altri casi è stato usato Testcontainer con l'immagine Docker mongo:7.0.0.

## 4.2 Dependency Injection: Google Guice

Per rendere le componenti del progetto loosely-coupled vengono inizializzate mediante dependency injection. Questo è però fonte di difficoltà durante la creazione del SUT poiché richiede di gestire le varie dipendenze. Durante gli unit test è stato usato Mockito per testare le componenti in isolation, gestendo le dipendenze mediante mock. A partire dagli integration test è invece stato usato Google Guice perché permette di gestire le dipendenze, anche quelle cicliche, mediante reflection.

Per utilizzare Guice è necessario annotare con Inject i costruttori delle classi che devono essere gestiti con Guice. In questo progetto i costruttori annotati sono il controller e le repository. Per gestire la dipendenza ciclica con la view è stato annotato con Assisted il parametro view nel costruttore del controller. Questo indica a Guice che nel momento della creazione del controller verrà passata l'istanza della view da iniettare. Per passare la view si definisce una factory per il controller, che accetta come parametro del metodo create la view. Al momento della creazione del controller Guice ricerca il metodo create associato per prendere i valori da iniettare nei parametri annotati con Assisted.

Una criticità di Guice riguarda il fatto che l'iniezione è guidata dai tipi. Quando è necessario iniettare molteplici valori allo stesso tipo, si può risolvere definendo delle annotazioni personalizzate per distinguere i valori. A questo punto nei costruttori si utilizzano tali annotazioni per indicare quali devono essere i valori associati a quei parametri.

La configurazione dei vari valori da iniettare è fatta estendendo la classe AbstractModule, ridefinendo il metodo configure. In questo progetto è stata definita un'apposita classe per creare il setup di default. In configure usando il metodo bind si associano alle interfacce le loro implementazioni e alle annotazioni i loro valori. Qualora sia necessaria una maggiore libertà si possono definire dei metodi annotati con Providers, per compiere le operazioni di associazione. Il metodo annotato con Providers viene associato al suo tipo di ritorno. In questo progetto sono stati definiti due metodi: uno per creare il MongoClient, l'altro per definire la DiscographySwingView. All'interno di quest'ultimo metodo viene risolta la dipendenza ciclica con il controller, usando la factory definita precedentemente.

Nella classe definita per configurare Guice alle annotazioni sono stati associati valori di default, tuttavia durante il setup del database è necessario passare i valori effettivi dell'indirizzo e della porta esposti dal host. Per farlo ho implementato una fluent interface ispirata al pattern builder, altrimenti era necessario estendere nuovamente AbstractModule sovrascrivendo i valori desiderati della classe che configura Guice.

## 4.3 Integration Test

Per rispettare la piramide dei test, la scelta di cosa testare negli integration test è fondamentale. In questa sezione ho già spiegato i motivi per cui non ho scritto degli integration test per verificare l'interazione tra una vera istanza del database e le repository.

Detto ciò ho preferito testare:

- DiscographyControllerIT verifica l'interazione tra il controller e le repositories. La dipendenza dalla view è gestita mediante Mockito, con cui ho verificato che il controller effettivamente chiamasse i metodi attesi dall'interfaccia view.
- DiscographySwingViewIT verifica l'interazione tra il controller e la view. Il SUT è stato creato usando Guice, ridefinendo l'associazione del tipo MongoClient. La ridefinizione di AbstractModule è necessaria poiché il test usa un in-memory database. Il controllo

è effettuato osservando l'aggiornamento dell'interfaccia grafica dopo che l'utente ci ha interagito.

- `ModelViewControllerIT` verifica l'interazione tra tutte le componenti del progetto. Anche in questo caso il SUT è stato creato usando Guice, senza però il bisogno di ridefinire `AbstractModule`. Il controllo viene effettuato sul database per verificare che le azioni compiute sull'interfaccia grafica si ripercuotano sul database.

È importante osservare che gli integration test effettuati nel `ModelViewControllerIT` non sono end-to-end test, visto che la verifica non è effettuata sull'interfaccia grafica, ma sul database.

#### 4.4 Cancellazione del musicista

La scrittura dei test ha richiesto di prestare particolare attenzione all'errore relativo all'assenza del musicista nel database, perché tipicamente la verifica della presenza viene effettuata due volte: la prima volta nel momento della selezione del musicista nella lista mentre la seconda volta quando vi viene eseguita sopra o un'operazione di cancellazione o aggiornamento, oppure viene eseguita un'azione su un album del musicista.

Quando viene testata la discografia è importante che il musicista sia presente nella lista, ma non nel database al momento della selezione. Invece negli altri casi è importante che il musicista sia presente nel database e nella lista al momento della selezione, ma subito dopo che sia rimosso.

Questa situazione è stata gestita chiamando i metodi del controller nel momento in cui c'era bisogno di andare a modificare sia il database che la view. Invece sono state usate le repository o direttamente le collezioni, quando era necessario agire esclusivamente sul database. Infine è stata utilizzata la classe `DiscographySwingView` o la finestra se era necessario modificare direttamente l'interfaccia grafica.

Questa criticità è osservabile anche nel file delle feature di Cucumber, negli scenari relativi alla discografia è stato definito un step apposito per indicare che viene rimosso dal database il musicista che verrà selezionato dall'utente. Negli altri casi la cancellazione avviene in contemporanea alla selezione del musicista.

## 5 Problemi incontrati durante la scrittura del progetto

Questa sezione parla dei problemi incontrati durante lo sviluppo dell'applicazione e delle soluzioni che ho adottato.

### 5.1 Versioni dei plugin e delle dipendenze

Dal momento che le specifiche del progetto richiedono di usare le versioni più recenti dei plugin e delle dipendenze, partendo dalle versioni viste ha lezione ho cercato di seguire la migrazione per arrivare alle attuali. In alcuni casi non c'è stato bisogno di aggiornarli perché la versione non è cambiata, in altri invece la migrazione verso le nuove versioni è stata minima, poiché sono solo state aggiunte nuove funzionalità come evidenzia il cambio della minor version. Le problematiche maggiori hanno riguardato i plugin che hanno cambiato la major version, ad esempio Mockito.

La dipendenza di Mockito attualmente è alla versione 5, che non può essere usata nel progetto, in quanto richiede almeno Java 11. Dato che l'applicazione nasce per Java 8 ho usato la versione 4. Diversamente da quanto abbiamo visto in classe, con la versione inclusa nel progetto è cambiato il modo d'inizializzare i mock, permettendo più semplicemente di attivare lo `strictstub`, un check di qualità per evitare di effettuare degli stub non necessari.



Detto ciò la maggior problematica riguarda la dipendenza `mongo-java-server`, che nel passaggio dalla versione 1.43.\* alla 1.44.0 ha iniziato a compilare richiedendo almeno Java 11, con conseguenza rottura della build quando eseguita con Java 8. Per fortuna avendo già incontrato il medesimo errore con Mockito, ho risolto usando una versione precedente.

## 5.2 Passaggio argomenti a linea di comando

A seguito dell'utilizzo di `Assert-Swing-Junit`, ho avuto la necessità di passare argomenti a linea di comando per evitare warning e eccezioni durante l'esecuzione della build. Il problema sollevato usando tale dipendenza riguarda l'utilizzo della reflection per accedere ai campi che non è più permesso a partire da Java 9.

Per passare gli argomenti a Maven è stato semplice, ho definito una property nel file POM, la quale è stata usata come argomento nella configurazione dei plugin `maven-surefire` e `maven-failsafe`, così che sia possibile usarla negli unit, integration e end-to-end test.

La problematica è sorta nel momento in cui ho dovuto calcolare il code coverage con Jacoco, perché il passaggio dei parametri richiesti falsifica il risultato di Jacoco. La soluzione adottata è stata quella di calcolare il code coverage eseguendo la build con Java 8.

## 5.3 Problemi legati ai workflow

Una delle cose più frustranti nello scrivere il progetto è stato configurare i workflow per GitHub Actions, poiché possono essere valutati solo al momento della loro esecuzione sul server. Ciò crea la spiacevole conseguenza che in caso di fallimento la build viene registrata con un errore nella storia di GitHub. Nel progetto non ci sono errori di questo tipo dato che i vari workflow sono stati correttamente definiti durante PoC del progetto. Tra le maggiori problematiche incontrate c'è l'invio del report di Jacoco a SonarCloud e Coveralls.

### 5.3.1 SonarCloud

Nella sezione precedente è stato concluso che l'analisi con Jacoco per essere corretta e non generare errori nella build deve avvenire usando Java 8. Purtroppo per eseguire SonarCloud senza generare warning è richiesto almeno Java 17 nell'ambiente virtuale.

La soluzione adottata nel progetto è stata quella di eseguire la build di Maven e la generazione del report di Jacoco con Java 8 e poi passare a Java 17 per inviare i report a SonarCloud. Tale scelta è stata fatta osservando che configurare la nuova versione di Java richiede pochi secondi.

### 5.3.2 Coveralls

Diversamente da quanto abbiamo visto in classe attualmente l'invio dei report a Coveralls può essere realizzato mediante un'azione su GitHub Actions.

Sfortunatamente la prima volta che l'ho usata non venivano lasciati i commenti nella pull-request, nonostante fosse stato spuntato il flag su Coveralls. Dopo aver ricevuto il medesimo esito usando il workflow visto in classe, ho scritto al supporto di Coveralls riguardo questo problema. Nella risposta ricevuta è stato evidenziato che il motivo dei mancati commenti era legato alla scadenza di un permesso del loro bot.

Nonostante non sia propriamente un mio problema ho deciso di riportarlo in questa relazione poiché mi ha bloccato per un paio di giorni.