Coin Collection Manager

Applicazione Java sviluppata con tecniche di programmazione avanzate, come TDD, build automation e continuous integration

Kevin Maggi
kevin.maggi@stud.unifi.it

Marzo 2023

Corso di Advanced Programming Techniques (6 CFU)
Laurea Magistrale in Ingegneria Informatica

Indice

1 Introduzione

In questo progetto ho sviluppato un'applicazione desktop dotata di GUI per la gestione di collezioni di monete, chiamata Coin Collection Manager, usando tecniche di programmazione avanzate, come per esempio *TDD*, *build automation* e *continuous integration*.

Attraverso una semplice interfaccia grafica è possibile gestire le monete raccolte in album presenti nella collezione, attraverso operazioni CRUD (sia per le monete, sia per gli album). È dunque possibile inserire, visualizzare, cercare, spostare ed eliminare degli album e delle monete dalla propria collezione. Ovviamente la collezione viene sempre tenuta in uno stato consistente, quindi, per esempio, durante l'operazione di eliminazione di un album vengono eliminate anche le monete ivi contenute e l'operazione di spostamento di una moneta è soggetta a controlli circa la disponibilità di slot liberi nel nuovo album.

Ovviamente l'obiettivo del progetto è mostrare come, attraverso suddette tecniche e strumenti, il ciclo di sviluppo del software si articola e viene supportato rendendolo più efficiente. Per questo motivo l'applicazione sviluppata non vuole essere un'applicazione particolarmente complessa, completa ed effettivamente usabile in ambito numismatico, ma anzi è stata volutamente mantenuta semplice per non distogliere il focus dal processo stesso di sviluppo (sebbene possa ancora essere utilizzata da collezionisti amatoriali).

Nella sezione 2 viene descritto design e implementazione del software, che presenta un'architettura a livelli implementata in maniera modulare.

L'applicazione è stata sviluppata in **Java 11** mediante il modello di sviluppo del software **Test Driven Development**, che punta a produrre codice quanto più possibile esente da bug e manutenibile. Il processo di sviluppo è stato agevolato da due pratiche, strettamente connesse tra loro, quali la **Build Automation**, attraverso **Maven**, e la **Continuous Integration** su **GitHub Actions**. Nella sezione 3 verrà fatto un focus su tutte le tecniche e strumenti utilizzati.

Alla luce di quanto illustrato, alcuni aspetti di maggiore interesse verranno presi in considerazione e analizzati in maggior dettaglio nella sezione 4.

Infine nella sezione 5 saranno descritti alcuni problemi riscontrati durante lo sviluppo, insieme alla soluzione trovata e proposta.

2 Struttura

2.1 Architettura

L'architettura del software (rappresentata in figura 2.1) è una classica architettura 3-tier formata da **Data Access Layer**, **Business Layer** e **Presentation Layer**.

L'accesso ai dati, quindi il Data Access Layer, avviene attraverso un pattern **Repository**, che consente di astrarre dal **Data Layer** confinando al suo interno ogni comunicazione necessaria col DB.

Il software è modellato secondo il pattern **Model View Presenter**, derivato dal MVC. Per questo motivo tutta la **business logic** è contenuta nel **Service Layer**, che costituisce dunque il solo sublayer del Business Layer; infatti il Presenter non contiene business logic, ma è un sublayer del Presentation Layer adibito solo a fare da intermediario tra View (che completa il Presentation Layer) e Model, che non comunicano direttamente.

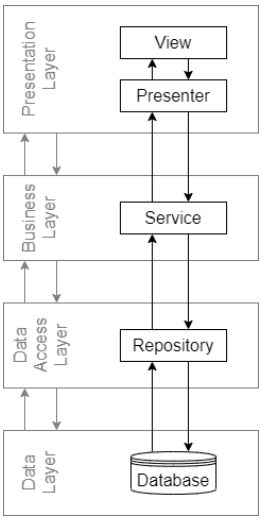


Figura 2.1: Architettura

2.1.1 Data Layer

Come Data Layer è stato scelto un database relazionale, che ben si presta a questo caso d'uso, attraverso l'utilizzo dell'API **JPA** per la persistenza dei dati. In particolare per il database la scelta è ricaduta su **PostgreSQL**, mentre per quanto riguarda l'implementazione di JPA è stato scelto **Hibernate**.

2.2 Design

Questa architettura è stata implementata dividendo il software in 3 moduli:

- il modulo **core**, che contiene il modello e il pattern Repository;
- il modulo **business**, che contiene il Service Layer e la gestione delle transazioni e
- il modulo **ui**, che contiene Presenter e View.

Nella figura a pagina 5 è riportato il class diagram, nel quale per semplicità di notazione, sono stati riportati solo i dettagli più rilevanti, omettendo attributi, metodi, classi e relazioni irrilevanti per questo livello di trattazione.

2.2.1 Model

Per definire il modello (in dettaglio in figura 2.2) partiamo dalla `BaseEntity` astratta, che raccoglie le caratteristiche base di una qualsiasi entità: è una classe astratta con un attributo `id`, che servirà da primary key (il tipo è UUID e viene autogenerato da JPA/Hibernate al momento della persistenza sul DB), e 3 metodi overrided (`toString()`, `hashCode()` e `equals()`), senza implementazione per forzare le entità concrete a implementarli.

Le entità concrete `Album` e `Coin` estendono questa classe base con i propri attributi: nome, volume, numero di slot disponibili e numero di slot occupati per `Album` (di cui nome e volume costituiscono una

chiave) e grado di conservazione, paese, anno di conio, descrizione, nota e album di appartenenza per `Coin` (di cui tutti tranne l'album di appartenenza costituiscono una chiave). Il grado di conservazione di una moneta ha valori definiti nell'enumerazione `Grade`.

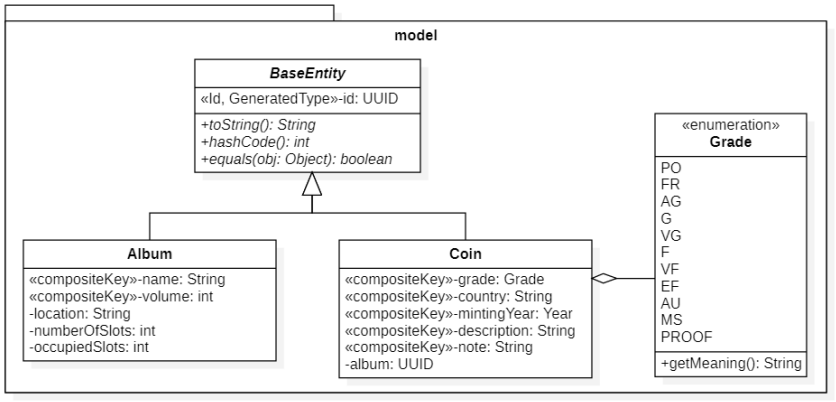


Figura 2.2: Class diagram del Model

2.2.2 Repository

Il pattern Repository è realizzato con un'interfaccia parametrizzata `BaseRepository` che espone i metodi standard di qualsiasi entità, quali `findById()`, `findAll()`, `save()` e `delete()`, estesa da due interfacce `AlbumRepository` e `CoinRepository` che aggiungono metodi specifici di ciascuna entità.

Queste interfacce vengono poi implementate in due classi (`AlbumRepositoryPostgres` e `CoinRepositoryPostgres`, entrambe sottoclassi di `PostgresRepository`) per il caso specifico di database relazionale gestito attraverso l'API JPA: i metodi introdotti poc'anzi verranno implementati con le chiamate JPA, in particolare tramite le chiamate all'`EntityManager`, che è un attributo della classe base astratta.

2.2.3 Service

Il livello di servizio viene realizzato da due interfacce, `AlbumManager` e `CoinManager`, che definiscono le operazioni di alto livello possibili sulle due entità: aggiunta, spostamento, rimozione e ricerca secondo vari criteri (tutte le entità, entità per id, album per nome e volume, moneta per descrizione, monete per album).

Questi metodi dovranno implementare la business logic mediante le operazioni di basso livello offerte dal livello Repository. Per il caso specifico, dato che viene usato un database relazionale, è necessario l'uso delle **transazioni** per garantire che il database sia sempre in uno stato consistente, ovvero che le operazioni offerte da questo livello soddisfino le proprietà ACID.

Le due classi che implementano le interfacce quindi, `AlbumTransactionalManager` e `CoinTransactionalManager` (entrambe sottoclassi della classe astratta `TransactionalManager`), implementano la business logic con le transazioni attraverso l'interfaccia `TransactionManager` descritta più avanti.

Vengono definite delle eccezioni specifiche del dominio, che potranno essere lanciate dalla business logic, ossia dall'implementazione di questo livello. Una prima fondamentale eccezione serve in caso di errore durante l'esecuzione di un'operazione a livello di database (`DatabaseException`), altre servono in caso di disallineamento tra oggetti Java e entità persistite (`Coin/AlbumNotFoundException`), mentre altre ancora rappresentano motivi per i quali una certa operazione non può essere completata (ad esempio una moneta non può essere spostata in un album già pieno).

2.2.3.1 Transaction Manager

`TransactionManager` è un'interfaccia che espone tre metodi `doInTransaction()` overloaded. Tutti e tre i metodi prendono come argomento delle funzioni e le eseguono "inoltrando" il valore restituito e assicurando che sia eseguito all'interno di una transazione; equivale a dire che se il codice va a buon fine, vengono persistite le modifiche tutte insieme, altrimenti nessuna modifica viene persistita: se la repository lancia un'eccezione, il transaction manager la cattura effettuando il rollback, altrimenti esegue il commit.

La differenza tra le tre versioni consiste sta nel tipo di funzione accettata come argomento: può essere un `CoinTransactionCode`, quindi una funzione che prende come argomento una `CoinRepository`, può essere un `AlbumTransactionCode`, quindi una funzione che prende come argomento una `AlbumRepository`, oppure può essere una `CoinaAlbumTransactionCode`, ovvero una bifunzione che prende come argomento entrambe le repository.

Questa interfaccia è implementata da una classe `TransactionManagerPostgres`, che implementa questi tre metodi utilizzando le transazioni offerte da JPA. Questa classe viene messa in vita dalla classe `PostgresTransactionManagerFactory`, seguendo il paradigma del pattern *Abstract Factory*.

2.2.4 Presenter

La classe astratta `Presenter`, con un riferimento a `View`, viene ereditata da due classi concrete `CoinPresenter` e `AlbumPresenter`; queste espongono i metodi corrispondenti alle operazioni di livello utente sulle entità, ovvero le azioni che un utente può compiere sulla collezione, le quali saranno realizzate invocando il livello service e presentando nella `View` i risultati/feedback.

Ricade tra i loro compiti catturare le eccezioni di dominio lanciate dal livello service, effettuando l'operazione più adeguata per informare l'utente e riportare l'applicazione in uno stato adeguato (ad esempio facendo comparire un messaggio e/o aggiornando la lista delle entità).

I metodi dei Presenter che modificano lo stato della collezione devono anche gestire le *race conditions*, sia a livello di applicazione che a livello di database: due di questi metodi non possono essere eseguiti contemporaneamente (o meglio, interleaved a causa di context switch) né da due thread della stessa istanza di applicazione, né da due thread di istanze diverse. Il primo caso lo si risolve rendendo `synchronized` i metodi, mentre il secondo con meccanismi del database, quali chiavi e transazioni, di cui abbiamo già parlato.

Dipendono dalla View e dal Service Layer per astrazione e quindi è possibile implementare altre versioni della UI o del DAL.

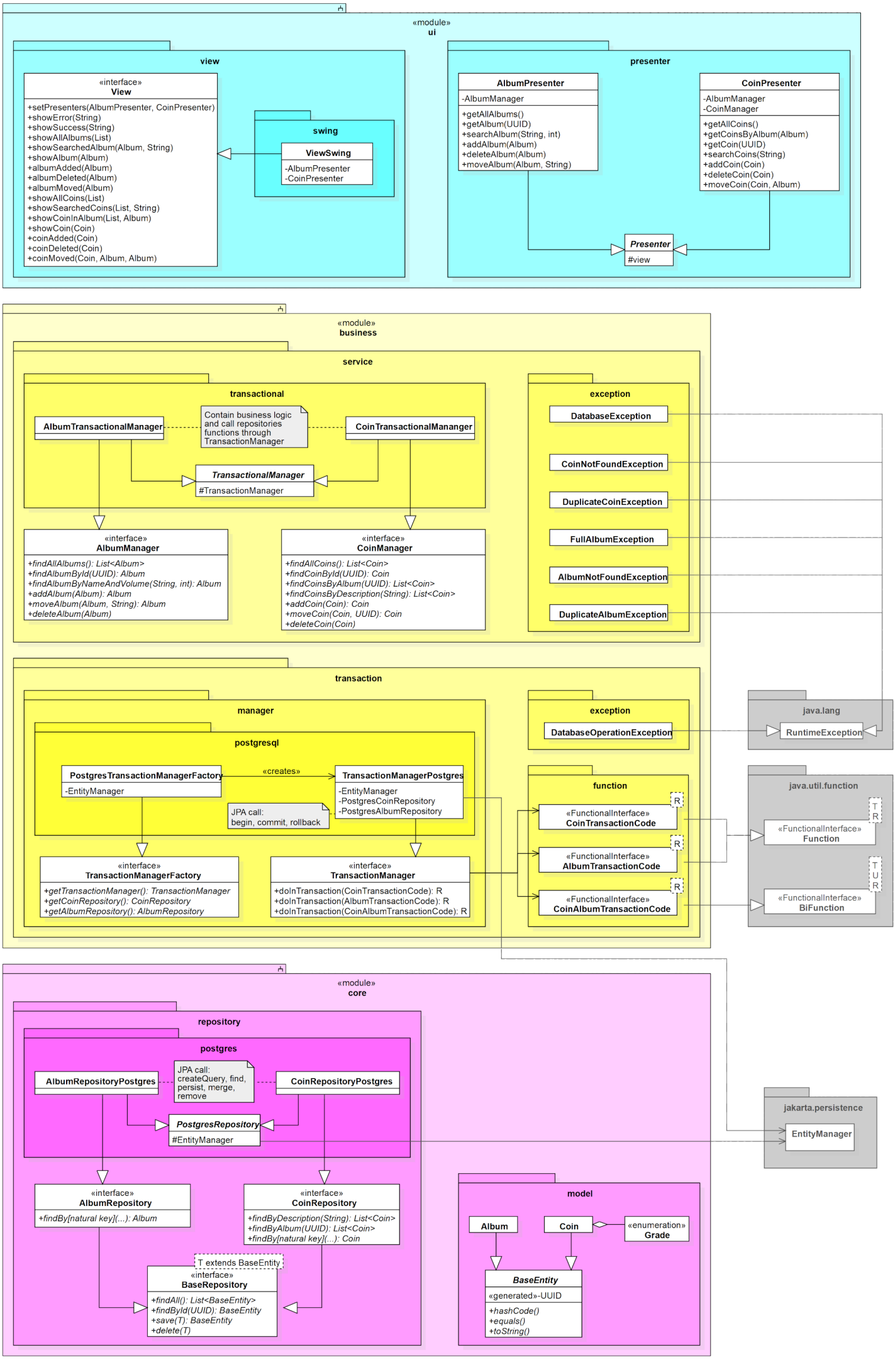
2.2.5 View

L'interfaccia `View` espone tutti i metodi utili al livello Presenter per aggiornare la UI, come `showError()` e `showSuccess()` per fornire un feedback all'utente, `showAllAlbums()`, `showAllCoins()`, etc.

Il metodo `setPresenters()` è necessario per installare la dipendenza delle implementazioni di `View` dai presenter; infatti a differenza di quanto succede nel MVC, nel MVP il Presenter dipende dalla View, ma anche viceversa.

La classe `ViewSwing` implementa quest'interfaccia realizzando una GUI in Java Swing.

La GUI dovrà essere responsive anche durante l'esecuzione delle operazioni (eventualmente long-running), ovvero le chiamate ai Presenter dovranno essere non-blocking, e questo può essere ottenuto eseguendole su altri thread (invece che nell'EDT). Tuttavia, siccome i Presenter poi invocheranno metodi della View, questi ultimi dovranno essere esplicitamente eseguiti nell'EDT.


«module»
ui
view
«interface»
View
+setPresenters(AlbumPresenter, CoinPresenter)
+showError(String)
+showSuccess(String)
+showAllAlbums(List)
+showSearchedAlbum(Album, String)
+showAlbum(Album)
+albumAdded(Album)
+albumDeleted(Album)
+albumMoved(Album)
+showAllCoins(List)
+showSearchedCoins(List, String)
+showCoinInAlbum(List, Album)
+showCoin(Coin)
+coinAdded(Coin)
+coinDeleted(Coin)
+coinMoved(Coin, Album, Album)
swing
ViewSwing
-AlbumPresenter
-CoinPresenter
presenter
AlbumPresenter
-AlbumManager
+getAllAlbums()
+getAlbum(UUID)
+searchAlbum(String, int)
+addAlbum(Album)
+deleteAlbum(Album)
+moveAlbum(Album, String)
CoinPresenter
-AlbumManager
-CoinManager
+getAllCoins()
+getCoinsByAlbum(Album)
+getCoin(UUID)
+searchCoins(String)
+addCoin(Coin)
+deleteCoin(Coin)
+moveCoin(Coin, Album)
Presenter
#view
«module»
business
service
transactional
AlbumTransactionalManager
Contain business logic and call repositories functions through TransactionManager
CoinTransactionalMananger
TransactionalManager
#TransactionManager
«interface»
AlbumManager
+findAllAlbums(): List<Album>
+findAlbumById(UUID): Album
+findAlbumByNameAndVolume(String, int): Album
+addAlbum(Album): Album
+moveAlbum(Album, String): Album
+deleteAlbum(Album)
«interface»
CoinManager
+findAllCoins(): List<Coin>
+findCoinById(UUID): Coin
+findCoinsByAlbum(UUID): List<Coin>
+findCoinsByDescription(String): List<Coin>
+addCoin(Coin): Coin
+moveCoin(Coin, UUID): Coin
+deleteCoin(Coin)
exception
DatabaseException
CoinNotFoundException
DuplicateCoinException
FullAlbumException
AlbumNotFoundException
DuplicateAlbumException
transaction
manager
postgresql
PostgresTransactionManagerFactory
-EntityManager
«creates»
TransactionManagerPostgres
-EntityManager
-PostgresCoinRepository
-PostgresAlbumRepository
JPA call: begin, commit, rollback
«interface»
TransactionManagerFactory
+getTransactionManager(): TransactionManager
+getCoinRepository(): CoinRepository
+getAlbumRepository(): AlbumRepository
«interface»
TransactionManager
+doInTransaction(CoinTransactionCode): R
+doInTransaction(AlbumTransactionCode): R
+doInTransaction(CoinAlbumTransactionCode): R
exception
DatabaseOperationException
function
«FunctionalInterface»
CoinTransactionCode
«FunctionalInterface»
AlbumTransactionCode
«FunctionalInterface»
CoinAlbumTransactionCode
R
java.lang
RuntimeException
java.util.function
«FunctionalInterface»
Function
T
R
«FunctionalInterface»
BiFunction
T
U
R
«module»
core
repository
postgres
AlbumRepositoryPostgres
JPA call: createQuery, find, persist, merge, remove
CoinRepositoryPostgres
PostgresRepository
#EntityManager
«interface»
AlbumRepository
+findBy[natural key](...): Album
«interface»
CoinRepository
+findByDescription(String): List<Coin>
+findByAlbum(UUID): List<Coin>
+findBy[natural key](...): Coin
T extends BaseEntity
«interface»
BaseRepository
+findAll(): List<BaseEntity>
+findById(UUID): BaseEntity
+save(T): BaseEntity
+delete(T)
model
Album
Coin
«enumeration»
Grade
BaseEntity
«generated»-UUID
+hashCode()
+equals()
+toString()
jakarta.persistence
EntityManager

3 Tecniche e strumenti di sviluppo

3.1 IDE [Eclipse]

Come IDE è stato usato **Eclipse** per la sua versatilità, infatti tutti gli strumenti usati, elencati nel seguito, si integrano perfettamente con questo IDE. Supporta nativamente JUnit e Git; fornisce plugin come *Docker Tooling* per gestire i container Docker, *m2e* per l'integrazione con Maven e *WindowBuilder* per un editor WYSIWYG per Java Swing; è possibile l'integrazione con Pitest col plugin *Pitclipse*; infine è disponibile il plugin ufficiale di Sonar, *SonarLint*.

3.2 VCS [Git - GitHub]

Come VCS è stato usato **Git** e la repository è stata ospitata su **GitHub** [1].

Ogni step dello sviluppo (implementazione o modifica successiva di un componente o di una classe) è stato effettuato su un branch apposito e solamente in un secondo momento fuso nel branch parent utilizzando il meccanismo delle Pull Request di GitHub, che consente, tra le altre cose, di effettuare dei controlli prima di confermare il merge (per esempio verificare il successo della build, il Code Coverage tramite Coveralls e la Code Quality tramite SonarCloud).

3.3 Build Automation [Maven]

La Build Automation rende l'intero processo di validazione, compilazione, testing e packaging eseguibile con un solo comando e lo strumento usato è stato **Maven**, che ha anche la funzione di *Dependency Management*, semplificando ancora di più lo sviluppo del software; infatti non è più necessario installare manualmente le dipendenze ma se ne occuperà Maven scaricandole da Maven Central.

Alla luce di ciò si può intuire che, se Maven attraverso il suo file `pom.xml` setta tutte le dipendenze e le configurazioni necessarie alla build, è sufficiente fissare la versione di Maven per ottenere la perfetta riproducibilità della build su macchine diverse. Per questo viene usato un *Maven Wrapper* che si occupa di scaricare e installare la versione fissata di Maven prima di eseguire il comando di build.

3.4 Continuous Integration [GitHub Actions]

La Build Automation apre le porte alla pratica della Continuous Integration, che permette di assicurarsi che ogni modifica effettuata (tipicamente testata in isolamento in locale) si integri correttamente, verificando il successo della build. Per questa pratica è stato adottato come CI server **GitHub Actions** per verificare il successo della build a ogni commit (o ogni PR a seconda delle piattaforme) e per il deployment della documentazione e del JAR per ogni tag.

3.5 Containerization [Docker - TestContainers]

I container sono l'alternativa lightweight alla virtualizzazione. Eseguendo un'istanza del server in un container **Docker** è possibile eseguire i test e, dopo aver creato l'immagine Docker dell'applicazione, con **Docker Compose** è possibile avviare con un solo comando server e applicazione ben orchestrati.

Durante lo sviluppo è fondamentale poter eseguire gli unit test in maniera veloce, anche quelli dei componenti che comunicano col database. Si potrebbe optare per un DB in-memory, ma **TestContainers** permette di avviare i container Docker in maniera programmatica e veloce.

Sebbene in letteratura talvolta siano indicati come integration test (perché di fatto si usa un server vero), qua sono stati trattati come unit test. Per gli integration ed end-to-end test, invece, l'avvio e lo stop del container Docker è stato responsabilità di un plugin Maven.

3.6 Test Automation [JUnit - AssertJ - Mockito - Awaitility]

Automatizzare i test rende il processo di sviluppo incredibilmente più produttivo e oggigiorno è indispensabile. Lo strumento utilizzato è **JUnit**, un framework di testing per Java molto potente; tuttavia la forma delle sue asserzioni non è molto intuitiva (soprattutto per il paradigma TDD, dove i test vogliono essere una sorta di documentazione eseguibile del codice) e per questo è stato usato **AssertJ** (e **AssertJ Swing**), così da rendere le asserzioni più leggibili nella forma soggetto-predicato-oggetto. A discapito del nome, JUnit è stato utile anche per gli integration ed end-to-end test.

Per testare il codice asincrono è stato usato **Awaitility**, che ha permesso di attendere fino a che il SUT non era pronto per la fase di verify, dopo l'exercise.

3.6.1 Mocking [Mockito]

Per gli unit test (e anche alcuni integration test) è stato necessario testare i componenti in isolamento dagli altri, comprese le dipendenze dirette. Per questo si è rivelato fondamentale **Mockito**, che consente di sostituire le dipendenze con dei *mock*, implementazioni fake che possono essere istruite senza difficoltà sul comportamento che devono avere in quella determinata circostanza.

3.7 Code Coverage [JaCoCo - Coveralls]

Per misurare la *Test Coverage* (o, impropriamente, *Code Coverage*) è stato sfruttato **JaCoCo**, in combinazione con **Coveralls** [2], un servizio online che ospita i risultati dell'analisi.

L'obiettivo era avere il 100% di code coverage (prendendo in considerazione solo il codice non auto-generato) ma, adottando il processo di sviluppo TDD, questo è sostanzialmente implicito.

3.8 Mutation Testing [Pitest]

Altrettanto importante è la qualità dei test, per capire se i test scritti testano effettivamente i comportamenti del codice. **Pitest** è uno strumento per l'automazione del processo di mutazione del codice e susseguente testing.

Per avere sufficiente confidenza nei test, e quindi nel codice, è necessario non avere mutanti sopravvissuti, ovvero mutazioni del codice che non fanno fallire alcun test (almeno tra quelli del set di default, ma in codice relativamente semplice come questo non ha troppo senso abilitare set più forti, che sebbene diano più fiducia aumentano il tempo necessario al processo di mutation testing).

3.9 Code Quality [SonarQube/SonarCloud/SonarLint]

Vogliamo anche che il codice sia di buona qualità, dove per qualità si intende l'assenza di *code smell* o problemi di *reliability*, *security* e *mantainability*.

SonarQube, insieme alla sua versione online **SonarCloud** [3] e alla sua versione ridotta real-time per IDE **SonarLint**, è lo strumento che è servito proprio all'analisi della qualità del codice. L'obiettivo è stato un *technical debt* pari a 0 (sebbene siano state ammesse eccezioni assolutamente ben giustificate dal contesto).

4 Sviluppo e implementazione

4.1 Sviluppo

Lo sviluppo di questo software, come già anticipato, ha seguito il paradigma TDD col suo ciclo red-green-refactor: per ogni comportamento atteso del codice è stato scritto un test e solo successivamente il codice corrispondente per fare passare il test; infine in alcune occasioni grazie alla presenza di test già passati è stato rifattorizzato il codice per migliorarne leggibilità e manutenzione (per esempio estraendo metodi privati di utilità). In rare occasioni sono stati aggiunti dei test col solo scopo di documentazione.

I test hanno seguito la metafora della piramide (fig. 4.1):

- **200 unit test**: i componenti sono stati testati in isolamento, usando dei mock delle dipendenze interne con metodi *stubbed*; questi test hanno coperto ogni branch decisionale del codice;
- **86 integration test**: i componenti sono stati testati in integrazione, eventualmente usando mock di dipendenze interne non facenti parte del SUT; questi test hanno coperto principalmente i casi positivi dei casi d'uso utente;
- **23 end-to-end test**: l'applicazione è stata testata nella sua interezza, trattandola in prospettiva black-box, ovvero senza usare direttamente le classi, ma solo la GUI; questi test hanno coperto solamente i casi positivi dei casi d'uso utente.

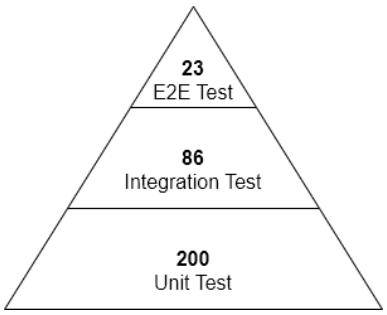


Figura 4.1: Piramide dei test

4.2 Implementazione

Il progetto è stato realizzato in 8 moduli Maven, che ora verranno descritti più in dettaglio, soffermandosi su alcuni aspetti implementativi di interesse.

4.2.1 BOM

Il modulo BOM serve solamente a tenere in sincronizzazione le versioni dei vari moduli che compongono il progetto; è una buona pratica nei progetti multi modulo per essere sicuri di usare la stessa versione di ogni submodule.

POM

Nella sezione `pluginManagement` sono elencati i plugin i cui goal sono associati di default a qualche fase dei 3 lifecycle di default di Maven (anche nei moduli con packaging *pom*, quindi anche questo): fissandone la versione si rende la build riproducibile.

4.2.2 Parent

Il modulo Parent (che eredita dal BOM) serve a raccogliere a fattor comune dipendenze e configurazioni che si applicano a tutti i submodule.

POM

Nella sezione `DependencyManagement` troviamo le dipendenze già citate, come Hibernate, PostgreSQL, TestContainers, JUnit, AssertJ, Mockito e Awaitility, ma anche **HickariCP** (un *connection pooler* che funge da *connection provider* per Hibernate), **commons-lang3** (un package di classi di utilità per i tipi di `java.lang`), **Log4j** (un *logger*) e **picocli** (un *command line argument parser*).

Nella sezione `PluginManagement` troviamo, oltre ai plugin classici:

- **Build Helper**: configurato per aggiungere le source folders `src/it/java` e `src/e2e/java` (non di default per Maven) per gli integration ed end-to-end test;
- **Failsafe**: configurato per eseguire in due esecuzioni distinte gli integration e gli end-to-end test riconoscendoli dal suffisso IT/E2E nel nome del file e per settare la proprietà di sistema `postgres.port` al valore della porta del server PostgreSQL da usare per i test;
- **JaCoCo**: configurato per misurare la Code Coverage, salvarla in un report e successivamente verificare che sia superiore a una soglia (la soglia non è del 100%, perché altrimenti la build fallirebbe immediatamente e i risultati non sarebbero inviati a Coveralls per la consultazione);
- **Pitest**: configurato per eseguire il Mutation Testing con la classe di mutatori di default e fallire in caso di qualsiasi sopravvivenza; viene configurato per evitare le chiamate al logger, così da risparmiare tempo e non intasare la console con gli output di questo;
- **Sonar**: non riceve alcuna configurazione perché i parametri di configurazione gli vengono passati da command line in quanto vogliamo eseguirlo solo in CI;
- **Coveralls**: non riceve alcuna configurazione per i motivi di cui sopra;
- **Docker**: configurato per avviare il container di PostgreSQL (con la corretta configurazione) all'avvio degli integration/end-to-end test (pre-integration-test) e stopparlo al termine (post-integration-test). Si occupa di salvare la porta sulla quale viene mappata la porta standard di Postgres in una proprietà Maven, che viene usata da Failsafe per settare la proprietà di sistema;
- **Exec**: non riceve alcuna configurazione perché viene configurato solo nel submodule App.

Infine definisce 4 profili: **jacoco** per abilitare la Code Coverage; **pitest** per abilitare il Mutation Testing; **coveralls** per abilitare la Code Coverage e condividerla con Coveralls (quindi è indipendente dal profilo *jacoco*) e **GUItest** che configura *Surefire* per eseguire solamente gli UI test (nella sezione 4.3 sarà chiaro il motivo).

4.2.3 Aggregator

Il modulo Parent qua non svolge la funzione di aggregatore, per il quale c'è il modulo Aggregator. Sostanzialmente questo modulo (indipendente da BOM e Parent) si occupa solo di raccogliere tutti gli altri moduli insieme per invocare il processo di build di tutto il progetto (tutti tranne Report, che viene incluso solamente quando attivo uno dei profili *jacoco* e *coveralls*).

POM

Nella sezione `PluginManagement` viene configurato il plugin **Site**, che serve a realizzare il sito, e vengono elencati alcuni plugin che servono a produrre la documentazione [4] (come **Javadoc**, **Surefire Report** e **Pitest** [16]) e che possono essere invocati anche da command line. Nella sezione `Reporting` vengono configurati questi plugin (e altri come **Project Info Reports** e **JXR**, che non hanno senso fuori dal lifecycle *site*) per produrre i report da includere nel sito: *Javadoc* e *Surefire Report* devono produrre i report aggregati. Se attivo il profilo **pitest**, anche *Pitest* dovrà produrre un report aggregato.

Infine nella sezione `properties` vengono elencate le regole di Sonar da escludere, ma queste verranno commentate più avanti.

4.2.4 Report

Questo modulo viene incluso nella build solamente quando attivi i profili *jacoco* e *coveralls* e serve ad aggregare i report *JaCoCo* dei vari moduli.

POM

L'unico contenuto è la configurazione dei plugin *JaCoCo* e *Coveralls* per escludere dalla Code Coverage le classi che hanno solo codice auto-generato (come il modello e le eccezioni) o che non sono testate con unit test (come la classe `App`). *JaCoCo* dovrà produrre un report aggregato e *Coveralls* condividerlo con Coveralls (alcuni parameri di configurazione sono passati tramite command line nel CI).

4.2.5 Core

Questo modulo (che eredita dal Parent) implementa il modulo Core del class diagram a pagina 5.

POM

Oltre a importare le dipendenze, il POM configura *Sonar* e *JaCoCo* per escludere il Model dalla Code Coverage e configura *Pitest* per il Mutation Testing: vengono mutate solo le classi che contengono della logica, in particolare quelle dei package `utility` e `repository.postgresql`.

Codice

Le entità del modello fanno override dei metodi `hashCode()` e `equals()`. Questi chiaramente non possono basarsi sull'id dato dalla chiave primaria autogenerato al momento della persistenza almeno per il seguente motivo: un oggetto non ancora persistito non ha ancora l'id ma può essere uguale a un altro oggetto. In realtà è buona norma lasciare che la chiave primaria sia usata solamente da JPA e implementare questi metodi attraverso una *business key* [17] [18]. Per questo motivo `Album` e `Coin` usano le chiavi descritte precedentemente per implementare questi metodi.

L'attributo `mintingYear` di `Coin` è di tipo `java.time.Year`, un tipo non supportato da Hibernate e JPA in generale. L'ostacolo viene superato implementando un `AttributeConverter` tra i tipi `Year` e `Integer` [19], così che JPA sia in grado di mappare automaticamente tra entità sul DB e oggetti.

Nelle classi Repository le query sul DB sono state effettuate tramite `TypedQuery`, un tipo di query di JPA espresse in *JPQL* che non sfruttano tutta la potenza espressiva di SQL, ma permettono un maggior controllo sui tipi.

Unit Test

Per i test delle classi Repository è stato sfruttato TestContainers: programmaticamente viene creato un container Docker dall'immagine Postgres all'inizio della classe di test e viene settata la proprietà di sistema `db.port` al valore della porta sulla quale viene mappata la porta standard di PostgreSQL. Questa proprietà viene usata nel file `persistence.xml`, che è il file dove viene definita la *persistence unit*, ovvero una serie di proprietà JPA/Hibernate che permettono di connettersi al DB.

Prima di ogni test il database viene svuotato per assicurare il massimo isolamento del test.

4.2.6 Business

Questo modulo (che eredita dal Parent) implementa il modulo Business del class diagram a pagina 5.

POM

Oltre a importare le dipendenze, il POM configura *Sonar* e *JaCoCo* per escludere dalla Code Coverage le classi dei package `transaction/exception`, `transaction/function` e `service/exception` e configura *Pitest* per il Mutation Testing: vengono mutati solo i Manager, che sono le classi che contengono

la business logic. Inoltre aggiunge il Transaction Manager alla lista di file per cui *Sonar* non valuta la *Code Duplication*, questo perché contiene i tre metodi overloaded `doInTransaction()`, che hanno esattamente la stessa implementazione, ma per 3 tipi diversi che non hanno superclassi in comune (o meglio solamente due di essi), quindi non può essere tolta la duplicazione.

Infine vengono attivati i plugin necessari per gli integration test.

Codice

`PostgresTransactionManager` esegue il codice che riceve come argomento dentro a una transazione ed eventualmente cattura le eccezioni facendo *rollback*. In questo file è stata disattivata la regola di Sonar che suggerisce di definire e lanciare un'eccezione dedicata anziché una generica: la segnalazione si riferiva al blocco catch che cattura una qualsiasi `RuntimeException` e la rilancia dopo il rollback, ma a questo livello non siamo interessati al tipo dell'eccezione e non la vogliamo gestire (sebbene sia necessario catturarla per fare rollback).

`PostgresTransactionManagerFactory` implementa la factory per `PostgresTransactionManager` e Repository secondo un principio di *Singleton*, infatti alla prima invocazione dei metodi vengono creati gli oggetti, ma alle successive vengono restituiti gli stessi oggetti.

Le classi `AlbumTransactionalManager` e `CoinTransactionalManager` implementano la business logic dell'applicazione avendo la responsabilità di eseguire le operazioni di alto livello in maniera transazionale. È a questo livello che vengono generate delle eccezioni specifiche del dominio, che verranno poi catturate dai Presenter.

Unit Test

Gli unit test non presentano alcun aspetto di particolare interesse da trattare; anche qua, per alcuni test, è stato necessario il server ed è stato avviato programmaticamente con TestContainers.

Integration Test

Sono presenti degli integration test che testano il livello servizio con delle Repository reali e, chiaramente, il vero server. Sono stati testati i casi d'uso di livello utente.

4.2.7 UI

Questo modulo (che eredita dal Parent) implementa il modulo UI del class diagram a pagina 5.

POM

Oltre a importare le dipendenze, il POM configura *Pitest* per il Mutation Testing: vengono mutate solo le classi del package `presenter` e le classi di utilità del package `view` (non viene mutata la classe `View` perché non ha senso mutare la GUI).

Infine vengono attivati i plugin necessari per gli integration test.

Codice

Come già detto i Presenter si occupano di invocare il Service Layer per ogni operazione di livello utente e di presentare all'utente il risultato/feedback attraverso la View. Per fare questo catturano tutte le eccezioni lanciate dal Service Layer e aggiornano la View per allinearla al database e/o informano l'utente. Si occupano infine di loggare su console l'esito delle operazioni.

La View cattura gli eventi invocando i Presenter (come già detto, in un nuovo thread). È stata creata con WindowBuilder e per questo motivo, essendo il codice auto-generato, sono state disattivate due regole Sonar: nomi delle variabili in formato non convenzionale e valori costanti duplicati. I suoi metodi sono invece esplicitamente eseguiti nell'EDT.

Una classe di utilità definisce un metodo per "convertire" un `ComboBox` in `Array`, utile per creare il `Dialog` di spostamento di una moneta con un dropdown con gli album attualmente mostrati in grafica.

Unit Test

Tra gli unit test della View, oltre a quelli per ogni suo metodo, ce n'è uno che verifica lo stato iniziale della GUI: verifica la presenza di ogni elemento e la sua corretta inizializzazione.

I test della View sfruttano il metodo `GUIActionRunner.execute()` per forzare le istruzioni che riguardano la GUI ad essere eseguite nell'EDT.

In questi file è stata disattivata la regola Sonar che segnala dei test senza asserzioni, questo perché Sonar non riconosce le asserzioni di AssertJ Swing (si tratta quindi di falsi positivi).

Integration Test

Anzitutto vengono testati i Presenter con un vero Service Layer (quindi veri Repository e server) per ogni caso d'uso di livello utente.

I Presenter vengono poi testati anche in alcuni casi di *race condition*: la situazione viene replicata mettendo in vita per ogni test diversi thread dentro ai quali istanziare i Presenter ed effettuare l'operazione. La verifica consiste nell'assicurarsi che, dopo che tutti i thread terminano l'esecuzione, lo stato del DB sia compatibile con la situazione in cui un solo thread ha avuto successo.

Infine viene aggiunta anche la View. In questi integration test sostanzialmente tutti i componenti sono presenti, rendendoli molto vicini a degli end-to-end test, ma i test vengono ancora eseguiti in prospettiva white-box, ovvero utilizzando le classi e non la GUI.

4.2.8 App

Questo modulo (che eredita dal Parent) contiene solo la classe `App`, che è quella col metodo main, e di conseguenza gli end-to-end test.

POM

Oltre a importare le dipendenze, il POM configura *JaCoCo* per escludere la classe dalla Code Coverage e *Pitest* per disabilitare il Mutation Testing. Vengono configurati il plugin **Assembly** per creare il FatJAR completo di MANIFEST e il plugin **Exec** per testarne la corretta esecuzione; e vengono attivati i plugin necessari per gli end-to-end test.

Viene infine definito il profilo **docker-build** nel quale il plugin *Docker* viene configurato per due esecuzioni: per creare l'immagine Docker dell'applicazione a partire dal Dockerfile (package) e per testarne il corretto funzionamento (verify).

Codice

Il metodo `main` si occupa solo di installare manualmente tutte le dipendenze, connettersi al database (e disconnettersi in chiusura) e avviare la GUI. Per la connessione al database il file `persistence.xml` non definisce tutte le proprietà necessarie (o le definisce con valori di default), lasciando che le altre (ossia: indirizzo del server, username, password e database) vengano definite programmaticamente con le informazioni passate tramite gli argomenti command line.

End-to-end Test

Gli end-to-end test coprono tutti i casi d'uso di livello utente in prospettiva blackbox, quindi interagendo solo con la GUI. Nel caso in cui un test debba partire con un database non vuoto, questo viene popolato direttamente a livello di DB, quindi usando JPA e bypassando i livelli Service e Repository.

È stata disabilitata la regola Sonar che segnala delle classi di test senza il suffisso standard (E2E non lo è).

Dockerfile

L'immagine base da cui partire è stata **Eclipse Temurin**, un'implementazione di OpenJDK (e non OpenJDK perché deprecata). Per creare l'immagine dell'applicazione poi viene copiato il FatJAR dal context al filesystem del container; questo, siccome contiene nel nome la versione, viene specificato come argomento (che verrà specificato dal plugin Docker di Maven quando invoca la build dell'immagine).

Essendo un'app dotata di GUI è necessario installare dei pacchetti che consentano di sfruttare l'ambiente grafico (in particolare **libxtst6**, **libxrender1** e **libxi6**).

Infine quest'immagine necessita evidentemente di un'istanza Postgres che sia già in grado di accettare connessioni. Non c'è modo di assicurare ciò, nemmeno sfruttando Docker Compose, perché (allo stato attuale di Docker Compose e plugin Docker di Maven) non è possibile specificare una condizione di attesa. Per aggirare il problema si usa lo script **wait-for-it** per attendere la disponibilità della porta di Postgres. Questo rende l'immagine funzionante senza errori anche quando lanciata senza Docker Compose: se è presente il server Postgres si connetterà, altrimenti uscirà dopo un timeout.

Il comando `CMD` quindi invoca lo script e, quando la condizione è verificata, avvia l'applicazione passando come argomenti per la connessione al database delle variabili d'ambiente (motivo per cui è stato specificato in *Exec form* e non in *Shell form*). Queste variabili dovranno essere specificate all'avvio del container (che avvenga con Docker o con Docker Compose).

4.3 Continuous Integration workflows

Sono stati messi a punto 5 workflow. Tutti come prime operazioni scaricano il contenuto della repository, mettono a punto il JDK e gestiscono la cache di Maven e Sonar. Vediamoli in dettaglio:

- **maven-linux**: è il workflow principale, esegue ad ogni commit e PR. Come prima cosa installa un *VNC server* e un *window manager*, necessari per l'UI testing (cfr. capitolo 5), dopodiché esegue la build attivando Code Coverage con Coveralls, Code Quality con Sonar, Mutation Testing e build dell'immagine Docker. Se la build dovesse fallire, allora si occuperà di generare i report JUnit e Pitest e di archiviarli, insieme anche a eventuali screenshot di UI test falliti, per poterli consultare. Gli altri workflow non ha senso che ripetano alcuni controlli come Code Coverage, Code Quality e Mutation Testing;
- **maven-macos**: esegue solo per le PR e per i commit del branch main, in quanto richiede molto tempo per installare Docker (non fornito nell'ambiente macOS di GH Actions). La build esegue i test e la build dell'immagine Docker per verificarla;
- **maven-windows**: esegue solo per le PR e per i commit del branch main. Non può effettuare i test perché l'ambiente Windows di GH Actions non supporta i container Linux e l'immagine Postgres è rilasciata solo per Linux. Ha tuttavia senso eseguire gli UI test, in quanto la GUI potrebbe comportarsi in maniera diversa tra Linux e Windows, quindi vengono eseguiti;
- **GH-Pages**: esegue solo per i commit sul branch main. Esegue la build (compresi test e Mutation Testing perché ne servono i report), produce i report, li integra nel sito e fa il deploy di quest'ultimo su GH Pages;
- **GH-Releases**: esegue solo per i tag. Esegue la build (che genera il FatJAR), genera il source JAR e il Javadoc JAR e fa il deploy di tutti e tre in una release della repository su GitHub.

5 Problemi affrontati

Durante lo sviluppo si sono presentati alcuni problemi. In molti casi la documentazione delle tecnologie/librerie/framework utilizzati è stata, ovviamente, di aiuto; in altri casi la soluzione ha richiesto della ricerca in più, mentre in altri ancora di cambiare strumento.

Riporto qua gli ostacoli principali incontrati durante lo sviluppo e la soluzione trovata; tutti hanno riguardato lo sviluppo o l'esecuzione dell'interfaccia grafica.

AssertJ Swing e JUnit 5

Tutti i test sono stati scritti con JUnit 5, che offre strumenti incredibilmente più potenti della versione precedente, come i test annidati e il nome custom (`@DisplayName`), soprattutto per la leggibilità dei test e la semplicità di utilizzo.

Tuttavia il `GUITestRunner` di AssertJ Swing non supporta JUnit 5 [25], dunque l'unica soluzione per i test che hanno riguardato la GUI (unit, integration ed end-to-end) è stata usare la versione precedente, che fortunatamente è inclusa in JUnit 5 con l'engine *JUnit Vintage*.

Lingua locale Java Swing

Durante gli UI unit test, i test riguardanti i due dialog della GUI fallivano perché non trovavano i button "cancel". Analizzando l'albero delle componenti fornito da AssertJ Swing in occasione dei fallimenti è stato possibile notare che al loro posto erano presenti dei button "annulla"; tuttavia lanciando la GUI da Eclipse per verificarne il motivo non si presentava questo problema.

Il motivo si è rivelato essere la proprietà di sistema `user.language`, da cui Swing attinge per la lingua della GUI: lanciando da Maven la proprietà viene settata di default alla lingua del sistema locale, mentre lanciando da Eclipse viene usata la lingua di Eclipse. È stato risolto forzando la proprietà all'inglese tramite la proprietà `<argLine>-Duser.language=en</argLine>` nel Parent POM.

Ambiente grafico in CI

L'aspetto che ha creato più problemi è stata la configurazione dell'ambiente grafico nel CI server. Infatti l'ambiente Linux di GH Actions non fornisce un ambiente grafico, tuttavia fornisce Xvfb e quindi inizialmente, come consigliato in [5], tramite il programma **xvfb-run** veniva avviato un *virtual X server* (configurando una risoluzione sufficientemente elevata da contenere tutta la GUI). Questo ha consentito di eseguire gli UI test, ma alcuni fallivano in maniera apparentemente non deterministica: non veniva trovato l'elemento della UI o non veniva eseguita correttamente l'azione (questo tipo di problemi è in realtà frequente, cfr. [20]).

Dopo una lunga ricerca la soluzione è stata usare un VNC server (anziché un X server) come consigliato dalla documentazione ufficiale di AssertJ Swing [21] attraverso lo script `execute-on-vnc.sh`. Ma non solo: i test hanno continuato a fallire fino a che non è stato usato anche un *window manager*. Con questa configurazione (**TightVNC** e **blackbox**) i test sono andati a buon fine.

Applicazione GUI con Docker

Un container Docker (e quindi l'applicazione dockerizzata) ha bisogno di alcuni accorgimenti per poter accedere, e quindi usare, l'ambiente grafico. In particolare ha bisogno di connettersi a un X server.

Su Linux, che dispone già un X server, questo è relativamente semplice: è sufficiente settare la variabile d'ambiente `DISPLAY` del container al valore dalla variabile d'ambiente `$DISPLAY` del sistema e impostare il volume `/tmp/.X11-unix:/tmp/.X11-unix` (a patto di aver già verificato che ogni client possa connettersi all'X server). Quindi è bastato impostare queste due cose nel `docker-compose`.

Windows non dispone di un X server, quindi questa non può essere una soluzione immediata. Può tuttavia diventarlo se installiamo un X server, lo configuriamo correttamente e settiamo la variabile d'ambiente `$DISPLAY` del sistema al valore `<YOUR_IP>:0.0` [23]. In questa situazione è possibile usare lo stesso `docker-compose`.

Tuttavia Windows, dalle versioni più recenti, offre WSLg [22], che abilita l'esecuzione delle applicazioni GUI Linux che sfruttano X11. Per poter sfruttare questo potente strumento, a patto di averlo installato, basta settare la variabile d'ambiente di sistema `$DISPLAY` al valore `:0`. Tuttavia cambia la locazione della cartella da passare al volume del container, che dunque diventa `/run/desktop/mnt/host/wslg/.X11-unix:/tmp/.X11-unix`. Per offrire anche questa opportunità, senza eliminare la soluzione di default per Linux e Windows, è stato sfruttato il meccanismo di overriding dei `docker-compose` [24]: un file secondario specifica solamente questa configurazione e quando usato in combinazione col principale sovrascrive il volume "di default".

A Replicazione della build

A.1 Requisiti

Per eseguire le istruzioni sottostanti, sono necessari i seguenti strumenti:

- Java 11
- Docker
- Docker Compose (già incluso in Docker Desktop)

Maven non è richiesto dato che viene usato un Maven Wrapper: per un'esperienza perfettamente riproducibile verrà scaricata la versione di Maven usata per lo sviluppo.

A.2 Ottenere il codice

A.2.1 Clonare la repository

Per clonare la repo:

```
git clone https://github.com/KevinMaggi/CoinCollectionManager.git
```

A.2.2 Progetto Eclipse

Se si vuole importare il progetto in Eclipse si può farlo importando da Git (puoi aver già clonato la repository o no).
Se si usa Eclipse Smart Imports, bisogna ricordare di non importare la cartella principale come progetto, ma solo le sottocartelle, per un'esperienza migliore.
A volte capitano degli errori e/o warning appena importato in Eclipse, ma è solo questione di fare il refresh dei progetti Eclipse e tutto sparisce.

A.3 Eseguire la build e i test

Dopo aver clonato la repo è necessario spostarsi nella cartella root del progetto:

```
cd CoinCollectionManager
```

Prima di lanciare la build è necessario avere l'immagine Docker **postgres:15.1**, altrimenti sarà scaricata durante la prima build, causando (probabilmente) il fallimento di questa a causa dello scadere di un timeout.

```
docker pull postgres:15.1
```

Dopodiché, per lanciare la build del progetto e tutti i test con Maven (a patto di avere già Docker in esecuzione), basta eseguire:

```
./mvnw -f coin-collection-manager-aggregator/pom.xml clean verify
```

Abilitando il profilo `-Pdocker-build`, sarà anche eseguita la build dell'immagine Docker dell'applicazione (verrà approfondito più avanti).
Se si è su Windows, bisogna sostituire `./mvnw` con `mvnw.cmd`. Inoltre su Windows, oltre ad abilitare l'opzione "*Expose daemon on tcp://localhost:2375 without TLS*", si suggerisce di eseguire il comando precedente con il parametro `-Ddocker.host=tcp://localhost:2375` al fine di rendere l'esecuzione più affidabile (nel senso che senza a volte fallisce perché Docker non è accessibile).

Se si è su Linux o macOS potrebbe essere necessario far precedere `sudo` al comando, in base alle configurazioni del sistema e di Docker, altrimenti la build fallirà quando richiederà Docker.
Se non si vuole usare il Wrapper (a proprio "rischio" di fallimento della build) basta sostituire tutte le occorrenze di `./mvnw` con `mvn`.

A.3.1 Test

Sono disponibili i seguenti profili Maven per abilitare funzioni extra durante i test:

- `-Pjacoco` per eseguire l'analisi della Code Coverage (verrà prodotto un report aggregato disponibile nella cartella `coin-collection-manager-report/target/site/jacoco-aggregate`);
- `-Ppitest` per eseguire anche il Mutation Testing (verrà prodotto un report aggregato disponibile nella cartella `coin-collection-manager-aggregator/target/pit-reports`).

Se invece si vuole saltare tutti i test, si può usare i parametri `-DskipTests` (per saltare unit, integration ed end-to-end test) e `-DskipITs` (per saltare solo gli integration ed end-to-end test).
Infine, se si vuole per qualche ragione eseguire solo gli UI test, si può fare usando il profilo `-PGUItest`.

A.3.1.1 Eseguire i test da Eclipse

Per eseguire gli unit test da Eclipse non è necessaria nessuna configurazione aggiuntiva, dato che sono eseguiti con TestContainers; per eseguire gli integration ed end-to-end test invece serve un'istanza in esecuzione e configurata correttamente di PostgreSQL. Può essere avviata in un container Docker con:

```
docker run --rm -p 5432:5432 -e POSTGRES_USER=postgres-user -e POSTGRES_PASSWORD=
  postgres-password -e POSTGRES_DB=collection postgres:15.1 postgres -c
  max_connections=300
```

A.4 Eseguire l'applicazione

Una volta fatta la build dell'applicazione è possibile eseguirla eseguendo il JAR o usando la sua immagine Docker. In caso si decida di eseguire tramite JAR è possibile evitare la build dell'applicazione e scaricare il FatJAR dalla release su GitHub.

A.4.1 Eseguire tramite JAR

Prima di tutto serve avviare un'istanza di PostgreSQL. Si può avviarla in un container Docker con:

```
docker run --name ccm_db -p 5432:5432 -e POSTGRES_USER=<YOUR_USER> -e
  POSTGRES_PASSWORD=<YOUR_PW> -e POSTGRES_DB=<YOUR_DB> postgres:15.1
```

o avviare un container creato precedentemente con:

```
docker start ccm_db
```

In Linux o macOS potrebbe essere necessario far precedente ai comandi Docker `sudo`, in base alle configurazioni del proprio sistema/Docker.
L'applicazione può essere avviata con:

```
java -jar ./coin-collection-manager-app/target/coin-collection-manager-app-1.0.0-
  jar-with-dependencies.jar --postgres-url=localhost --postgres-port=5432 --
  postgres-db=<YOUR_DB> --postgres-user=<YOUR_USER> --postgres-password=<YOUR_PW>
```

Ovviamente si può anche usare un'altra istanza di PostgreSQL, basta modificare i parametri `--postgres-url` e `--postgres-port`.
Lanciando l'applicazione con `--help` si può ottenere informazioni sui parametri accettati.

A.4.2 Eseguire tramite Docker

Se la build dell'applicazione viene lanciata con il profilo `-Pdocker-build`, verrà fatta la build di un'immagine Docker dell'applicazione.
Si può avviare l'applicazione con un solo comando grazie a Docker Compose, che avvierà anche un'istanza di PostgreSQL. Il comando è il seguente e deve essere eseguito nella cartella principale, ma prima di questo è necessario impostare l'ambiente grafico: le istruzioni sono riportate più avanti in base al proprio OS.

```
docker compose up
```

A.4.2.1 Linux

In OS Linux l'unica cosa da fare è disabilitare il controllo degli accessi all'X server (se non è già disabilitato) con

```
xhost +
```

e poi è tutto pronto per `sudo docker compose up`.

A.4.2.2 Windows

In Windows per eseguire un'app dotata di GUI con Docker (che eseguirà container Linux dato che PostgreSQL non esiste per Windows) ci sono due opzioni: installare un X server (come **VcXsrv**) o sfruttare il nuovo **WSLg** (per Windows 11 e Windows 10 Buil 19041 o successive).

- X server Per usare un X server basta:

- eseguirlo;
- disabilitare il controllo degli accessi;
- settare una variabile `DISPLAY` che "punti" al display (con l'IP della macchina, che si può trovare con `ipconfig`:

  ```
  set DISPLAY=<YOUR_IP>:0.0
  ```

- eseguire `docker compose up`

- - VcXsrv un possibile X server è VcXsrv, che si può scaricare da qua.
Una volta scaricato e installato è possibile avviarlo scegliendo le opzioni "*multiple windows*" e "*start no client*" e spuntando l'opzione "*disable access control*".
Dopodiché si può settare la variabile `DISPLAY` e infine avviare l'applicazione con Docker Compose, come già visto.

- WSLg Per usare WSLg [22] la prima cosa da fare è installare l'ultima versione di WSL dal Microsoft Store o aggiornarlo se è già installata una versione precedente:

```
wsl --update
```

Se Docker era in esecuzione potrebbe essere necessario riavviarlo dopo l'update di WSL.
Adesso basta settare la variabile `DISPLAY`:

```
set DISPLAY=:0
```

e si può avviare l'applicazione con Docker Compose. In questo caso è necessario sovrascrivere una configurazione per il container dell'app, quindi il comando diventa:

```
docker compose -f docker-compose.yaml -f docker-compose.wslg.yaml up
```

Riferimenti

Repository

[1] CCM GitHub. `github.com/KevinMaggi/CoinCollectionManager` .

[2] CCM Coveralls. `coveralls.io/github/KevinMaggi/CoinCollectionManager` .

[3] CCM SonarCloud. `sonarcloud.io/project/overview?id=KevinMaggi_CoinCollectionManager` .

[4] CCM documentation. `kevinmaggi.github.io/CoinCollectionManager/` .

Libro

[5] Lorenzo Bettini. *Test-Driven Development, Build Automation, Continuous Integration. With Java, Eclipse and friends*. Leanpub.

Documentazioni

[6] AssertJ Doc. `assertj.github.io/doc/` .

[7] AssertJ Swing Doc. `javadoc.io/doc/org.assertj/assertj-swing/latest/index.html` .

[8] Docker Maven Plugin. `dmp.fabric8.io/#introduction` .

[9] Hibernate Doc. `hibernate.org/orm/documentation/6.1/` .

[10] Hibernate Doc - Setup and configuration. `docs.jboss.org/hibernate/stable/entitymanager/reference/en/html/configuration.html` .

[11] Javadoc Doc. `oracle.com/java/technologies/javase/javadoc-tool.html#javadocdocuments` .

[12] Mockito Doc. `javadoc.io/doc/org.mockito/mockito-core/4.11.0/org/mockito/Mockito.html` .

[13] Pitest Quickstart. `pitest.org/quickstart/maven/` .

[14] TestContainers JDBC Support. `testcontainers.org/modules/databases/jdbc/` .

[15] TestContainers Postgresql Javadoc. `javadoc.io/doc/org.testcontainers/postgresql/latest/index.html` .

Guide

[16] PIT - Test aggregation across modules. `https://pitest.org/aggregating_tests_across_modules/`.

[17] JBoss Community Archive. Equals and HashCode. `https://developer.jboss.org/docs/DOC-13933`.

[18] Thorben Janssen. Ultimate Guide to Implementing `equals()` and `hashcode()` with Hibernate. `https://thorben-janssen.com/ultimate-guide-to-implementing-equals-and-hashcode-with-hibernate/`.

[19] Vlad Mihalcea. How to map `java.time` Year and Month with JPA and Hibernate. `https://vladmihalcea.com/java-time-year-month-jpa-hibernate/`.

Risoluzione problemi

[20] StackOverflow question. https://stackoverflow.com/questions/57338739/why-do-we-get-occasional-failures-in-assertj-swing-tests.

[21] AssertJ Swing Documentation - Running Tests. https://joel-costigliola.github.io/assertj/assertj-swing-running.html.

[22] Enabling WSL to include support for Wayland and X server related scenarios. https://github.com/microsoft/wslg.

[23] Running WSL GUI Apps on Windows 10. https://techcommunity.microsoft.com/t5/modern-work-app-consult-blog/running-wsl-gui-apps-on-windows-10/ba-p/1493242.

[24] Share Compose configurations between files and projects. https://docs.docker.com/compose/extends/.

[25] Lorenzo Bettini. Support for JUnit 5. GitHub issue. https://github.com/assertj/assertj-swing/issues/259.