Relazione sul Progetto: Finance APP

Stefania Cerboni

04/02/2024

Sommario

Questo progetto si propone di sviluppare un'applicazione di gestione finanziaria che permette agli utenti di monitorare e gestire le proprie spese attraverso diverse categorie.

Utilizzando il pattern architetturale Model-View-Controller (MVC), l'applicazione separa la logica di business dalla logica di interfaccia utente, migliorando così la manutenibilità e il testing del software.

Il modello comprende classi per utenti, categorie e spese, ciascuna gestita tramite servizi dedicati che interagiscono con un database relazionale.

Lo sviluppo del progetto è stato fatto seguendo l'approccio Test-Driven Development (TDD) e i test implementati hanno verificato la correttezza delle funzionalità attraverso Unit Test, Integration Test e E2E Tests, assicurando l'affidabilità dell'applicazione.

La copertura del codice, monitorata attraverso strumenti come JaCoCo e visualizzata su Coveralls, ha dimostrato un'alta percentuale di linee di codice testate, confermando l'efficacia dei test implementati e la loro qualità, calcolata mediante SonarCloud.

1 Introduzione

Il progetto corrente prevedeva l'implementazione di una semplice applicazione di gestione delle spese (Expense), specificando la categoria della spesa (Category) e l'utente che effettuava la spesa (User). L'implementazione ha seguito l'approccio Test-Driven Development (TDD) che non solo aiuta a prevenire difetti nel software ma promuove anche un design modulare e facilmente estensibile.

Nel corso dello sviluppo, sono stati implementati test unitari e di integrazione utilizzando strumenti come JUnit e AssertJ per GUI testing, insieme a Testcontainers per simulare l'ambiente di database in modo isolato. Per tracciare la copertura del codice è stato utilizzato Jacoco in combinazione con Coveralls per monitorare e mostrare la copertura dei test nel repository GitHub, assicurando che tutte le funzionalità siano adeguatamente testate.

L'integrazione continua attraverso GitHub Actions ha permesso di automatizzare il processo di build e test, garantendo che ogni commit sia verificato in modo efficace, mantenendo così l'integrità del codice su GitHub.

Questo documento descriverà in dettaglio l'architettura dell'applicazione, le scelte di design e le metodologie di testing adottate, fornendo un quadro chiaro dello sviluppo del progetto e delle strategie impiegate per garantire un prodotto software di alta qualità.

2 Strumenti e Framework Utilizzati

Durante lo sviluppo del progetto, si è fatto ricorso a una serie di strumenti e framework essenziali per garantire l'efficacia e l'efficienza del processo di sviluppo, oltre a rispettare i principi di qualità del software e di test driven development (TDD).

- **Maven** è stato impiegato come strumento di gestione e comprensione del progetto. Ha permesso di definire il ciclo di vita della build, gestire le dipendenze e garantire la portabilità del progetto attraverso l'uso del file POM.xml.
- Per l'implementazione di test unitari e di integrazione è stato utilizzato **JUnit 5**. Attraverso la sua avanzata integrazione con IDE e build tools come Maven, ha permesso di eseguire cicli di test ripetuti e sistematici durante tutto il processo di sviluppo.
- **Mockito** è stato scelto per creare e gestire mock objects necessari nei test unitari, per isolare le classi da testare. È stato fondamentale per simulare le interazioni con le dipendenze esterne nelle classi Service e Controller, permettendo così di verificare il comportamento atteso in condizioni controllate e indipendenti.
- **AssertJ** e **Hamcrest** sono stati utilizzati per arricchire le asserzioni nei test, migliorando la leggibilità e la potenza delle verifiche effettuate su vari stati e risultati di esecuzione del codice.
- **Testcontainers** è stato impiegato per gestire container Docker nei test di integrazione, garantendo così un ambiente isolato e riproducibile per testare le interazioni con il **database MySQL**.
- **PIT** è stato utilizzato per eseguire mutation testing sul codice di business logic, garantendo non solo che i test avessero una buona copertura del codice, ma anche che fossero efficaci. Integrato con Maven e configurato per considerare i mutators principali nelle classi più importanti del progetto, in modo da non richiedere eccessivo tempo per analizzare il codice.
- **Jacoco** è stato utilizzato per monitorare e raccogliere la copertura del codice durante l'esecuzione dei test, integrato con Maven per generare report dettagliati sulla copertura raggiunta dai test, sia unitari che di integrazione.
- **Coveralls** è stato integrato con il sistema di continuous integration per tracciare la storia e l'evoluzione della copertura dei test nel tempo, fornendo una visione chiara dell'efficacia dei test e delle aree del codice che potrebbero richiedere maggiore attenzione.
- **JPA** (Java Persistence API) è stato utilizzato per facilitare l'interazione con il database, mappando gli oggetti Java a tabelle di database in modo trasparente e efficiente.
- **Hibernate** è stato scelto come ORM per facilitare l'interazione con il database, mappando gli oggetti Java a tabelle di database in modo trasparente e efficiente, contribuendo significativamente alla riduzione del boilerplate code e all'incremento dell'efficienza dello sviluppo.
- Infine, **Swing** è stato impiegato per costruire l'interfaccia grafica utente.

3 Scelte di Design e Implementazione

- **Architettura del Sistema:** Il sistema è stato progettato seguendo il modello architetturale Model-View-Controller (MVC), che separa la logica di business dalla logica di presentazione, facilitando la manutenzione e la scalabilità dell'applicazione. Questa scelta consente una chiara separazione delle responsabilità tra i componenti del sistema, rendendo anche il codice modulare e quindi più facilmente testabile:

  1. **Modello**: Gestisce i dati e le regole di business. Nel contesto del sistema sviluppato, il modello è rappresentato dalle classi entità come `User`, `Category`, e `Expense`, che sono gestite tramite il framework di persistenza Hibernate per l'interazione con il database MySQL.
  2. **Vista**: Presenta i dati all'utente e raccoglie l'input dell'utente. Le viste sono implementate utilizzando Java Swing, permettendo una rappresentazione grafica degli elementi di UI come form, tabelle e bottoni. Nel modello sviluppato, le viste sono le interfacce `CategoryView`, `UserView` e `ExpenseView` con le relative implementazioni: `CategoryPanel`, `UserPanel` e `ExpensePanel`.
  3. **Controller**: Agisce come intermediario tra modello e vista, gestendo il flusso di controllo nel sistema. I controller, come `UserController`, `CategoryController` e `ExpenseController`, manipolano i dati del modello in risposta alle azioni dell'utente catturate dalla vista.

- **Gestione della Persistenza:** La persistenza dei dati è affidata a Hibernate ORM, configurato per lavorare con un database MySQL. Le classi entità sono mappate su tabelle del database, permettendo operazioni CRUD attraverso la sessione di Hibernate. L'uso di Hibernate riduce il carico di gestione diretta delle connessioni e delle query SQL, automatizzando la gestione delle entità e delle transazioni.

- **Implementazione dell'Interfaccia Utente:** L'interfaccia utente è stata sviluppata utilizzando Java Swing, fornendo un'interfaccia grafica reattiva e intuitiva. Le classi `CategoryPanel`, `UserPanel` e `ExpensePanel` organizzano i pannelli per differenti funzionalità (utenti, categorie, spese) in schede separate. I tre pannelli sono poi uniti in tab diverse del `MainFrame`. Per garantire l'intuitività dell'interfaccia, sono stati implementati listener specifici per gestire eventi come *click sui bottoni* e *modifiche ai campi di testo*, facilitando così una reattività immediata alle azioni dell'utente. Gli aggiornamenti di stato sono mostrati tramite una label di stato dinamica, che fornisce feedback in tempo reale.

- **Test e Qualità del Software:** Per assicurare la qualità del software, sono stati implementati test unitari e di integrazione utilizzando JUnit e Mockito per simulare le interazioni tra i componenti. I test di integrazione con Testcontainers permettono di verificare il comportamento del sistema in un ambiente che simula da vicino quello di produzione, utilizzando un container Docker per MySQL.

  Jacoco è stato utilizzato per monitorare la copertura del codice durante i test, garantendo che tutte le funzioni critiche siano adeguatamente testate. L'integrazione continua è stata configurata con GitHub Actions, eseguendo build e test automatici ad ogni push nel repository, assicurando che le regressioni vengano identificate e gestite prontamente.

4 Problemi Incontrati e Soluzioni Adottate

Durante lo sviluppo del progetto, sono state affrontate sfide e criticità alle quali sono state trovate soluzioni efficaci che hanno garantito l'integrità e la funzionalità del sistema.

- **Gestione delle Dipendenze di Expense:** Uno dei problemi principali riguardava la gestione delle dipendenze dell'entità Expense. Era fondamentale assicurarsi che fosse possibile aggiungere un'Expense solo quando gli User e le Category corrispondenti fossero già presenti nel database. Per risolvere questo problema, l'interfaccia utente è stata progettata per utilizzare DropDownBox che popolano i campi selezionabili solo con dati effettivamente presenti nel database. Questo approccio ha garantito che i dati fossero caricati al momento dell'apertura dell'interfaccia e aggiornati ogni qual volta si accedeva alla tab dell'Expense, evitando l'inserimento di riferimenti non validi.

  La stessa accortezza si è avuta in fase di cancellazione: si è reso impossibile l'eliminazione di Category o User se collegati ad Expenses presenti nel database: per eliminare tali dati e non incorrere in una violazione delle dipendenze, si deve prima eliminare l'Expense corrispondente e poi si può procedere alla cancellazione degli User o Category desiderati.

  Le classi 'Service' sono quelle che hanno la responsabilità di fare i sopracitati controlli prima di richiamare il Repository.

- **Integrazione con Database MySQL:** un'altra sfida è stata l'integrazione con un database MySQL, nonostante il corso fosse focalizzato sull'uso di MongoDB. L'implementazione è stata effettuata in alcuni Unit Tests utilizzando un database H2 in-memory MySQL mentre nei Test d'Integrazione si è preferito adottare la libreria TestContainers che mette a disposizione container Docker pre configurati, tra cui Database MySQL.

- **Copertura del Codice e Mutation Testing:** Nell'ambito del testing, si sono verificate difficoltà nel raggiungere una copertura del codice completa. In particolare, i DocumentListener aggiunti ai textBox non permettevano di coprire integralmente i metodi, dato che il metodo changedUpdate(DocumentEvent e) non veniva attivato con l'uso di campi di testo semplici. Per ovviare a questo, è stata utilizzata l'annotazione *@Generated*, che ha permesso di escludere dal calcolo della copertura i metodi non utilizzati, assicurando così una valutazione più accurata dell'effettiva copertura del codice.

- **Valutazione della Qualità del Codice:** Infine, è stato riscontrato un problema nel rilevamento automatico delle asserzioni di AssertJ per Swing da parte di SonarCloud. Per risolvere questa problematica, è stata implementata una *regola personalizzata* che ha specificato tra le asserzioni ammesse quelle contenenti il verbo "required". Questa modifica ha permesso di includere tutte le asserzioni utilizzate da AssertJ, garantendo così un'accurata valutazione della qualità del codice.

5 Sviluppo e Test

5.1 Progettazione del Modello

Il modello dell'applicazione è strutturato attorno a tre entità principali: Expense, User, e Category. Queste classi sono state progettate per gestire rispettivamente le spese, gli utenti e le categorie di spese.

1. **Expense**: Ogni istanza rappresenta una spesa, contenendo attributi per l'identificativo, l'utente associato, la categoria, l'importo e la data.

2. **User**: Gestisce le informazioni degli utenti con attributi per l'identificativo, nome utente, nome, cognome e email.

3. **Category**: Classifica le spese, dotata di attributi per l'identificativo, il nome e la descrizione.

5.2 Implementazione del Modello

L'implementazione del modello è stata guidata dall'approccio Test-Driven Development, con la creazione di classi di test specifiche per ogni entità: `CategoryTest`, `UserTest`, e `ExpenseTest`.

Ogni fase di sviluppo è stata accompagnata dalla scrittura di test che precedevano l'implementazione del codice. Questo ha garantito che ogni nuova funzionalità fosse immediatamente verificata e conforme ai requisiti funzionali. La verifica continua attraverso i test ha anche facilitato il processo di refactoring, permettendo modifiche al codice con una ridotta incidenza di errori.

5.3 Annotazione delle Classi e Configurazione della Persistenza

Dopo aver completato l'implementazione iniziale e i test delle classi Category, User e Expense, è stato essenziale stabilire la loro persistenza nel sistema di gestione del database. Questo passaggio è stato realizzato utilizzando Java Persistence API (JPA) per annotare le classi del modello di dominio.

- **Annotazioni JPA:** Ogni classe del modello di dominio è stata annotata con `@Entity`, indicando che tali classi sono entità JPA. I campi chiave di ogni entità sono stati annotati con `@Id` e `@GeneratedValue`, per gestire automaticamente la generazione degli ID (nello specifico, è stata adottata una generazione auto-incrementale degli ID). Le relazioni tra le entità, come quelle tra Expense e User o Category, sono state definite utilizzando le annotazioni `@ManyToOne` e `@JoinColumn`, stabilendo così le connessioni necessarie nel database.

- **Configurazione con persistence.xml:** Le specifiche di configurazione della persistenza sono state definite nel file `persistence.xml`. Questo include la configurazione dell'unità di persistenza sia di produzione che di testing, specificando un database in-memory MySQL.

5.4 Sviluppo dei Repository

Il passaggio successivo ha riguardato il test e l'implementazione della persistenza. Come appreso nel corso, per fare ciò è stato adottato il design pattern Repository, per ogni entità del modello di dominio è stata quindi definita un'interfaccia repository che delineava le operazioni CRUD necessarie. Queste interfacce hanno funto da contratto per le classi di implementazione, specificando i metodi essenziali per l'interazione con il database.

Sono quindi stati implementati i tre repository principali, ognuno dedicato alla relativa entità nel modello di dominio: `CategoryRepository`, `UserRepository`, `ExpenseRepository`. Anche in questo caso, i Repository sono stati implementati utilizzando l'API di JPA e Hibernate per la persistenza, prevedendo l'impiego di un Database MySQL.

Tuttavia, per isolare i test dal database di produzione, è stato utilizzato un database H2 in memoria. Questo ha permesso di eseguire test rapidi, indipendenti dallo stato del database reale, garantendo quindi l'isolamento delle classi testate da eventuali dipendenze. L'ambiente di test è stato configurato nel metodo setUp, dove vengono inizializate le risorse necessarie per testare i metodi del Repository:

1. Creazione dell'EntityMangerFactory e dell'EntityManager, sfruttando le configurazioni dell'unità di persistenza collegata con il database in-memory, specificata nel file persistence.xml.

2. Inizializzazione di eventuali istanze necessarie per la creazione di entità: questo è stato utile soprattuto nel caso dell'ExpenseRepository, visto che Expense ha un riferimento sia a User che a Category, per poter aggiungere una Expense è necessario che essi siano presenti a priori.

```
1     @BeforeEach
2     void setUp() {
3         emf = Persistence.createEntityManagerFactory("
    TestFinanceAppH2PU");
4         em = emf.createEntityManager();
5         expenseRepository = new ExpenseRepositoryImpl(em);
6         category = new Category("Travel", "Expenses for travel");
7         user = new User("john.doe", "john.doe@example.com");
8         em.getTransaction().begin();
9         em.persist(category);
10        em.persist(user);
11        em.getTransaction().commit();
12    }
13
```

3. Persistenza delle entità di supporto nel database di test.

I test implementati sono stati molteplici e controllano le funzionalità sia nel caso in cui tutto vada come previsto, sia nel caso in cui si presentino errori o eccezioni.

5.5 Sviluppo dei Servizi

Una volta stabilite le basi per la persistenza dei dati attraverso i repository, l'attenzione si è spostata verso lo sviluppo dei servizi, componenti fondamentali per la gestione della logica di business. L'implementazione dei servizi, seguendo sempre l'approccio Test-Driven Development, ha integrato l'uso del mocking per assicurare che i test fossero focalizzati sulla logica di business senza dipendere direttamente dall'implementazione

concreta dei repository. Questo ha permesso una chiara **separazione tra la logica di business** implementata nei servizi **e la logica di persistenza** nei repository, facilitando ulteriori modifiche e manutenzioni senza impatti significativi sull'intero sistema.

La creazione dei servizi è iniziata dalla definizione di test specifici che miravano a validare non solo la corretta esecuzione delle operazioni di business ma anche la gestione di scenari particolari, come la validazione dei dati di input e la manipolazione di errori.

I servizi sono stati progettati per gestire in modo efficiente le transazioni, assicurando la coerenza dei dati e l'atomicità delle operazioni complesse, ma anche per gestire efficacemente le eccezioni, garantendo che l'applicazione possa recuperare in modo controllato da situazioni impreviste e fornire risposte appropriate agli utenti.

5.6 Sviluppo del Model View Controller

Una volta completata e validata la sezione backend per l'interazione con il database e la gestione della persistenza delle entità, l'attenzione si è spostata verso lo sviluppo dell'applicazione. Questo includeva, in particolare, l'implementazione dei moduli Controller e View, dato che il Model era stato definito in precedenza.

Il processo è iniziato con la progettazione dell'interfaccia utente. Inizialmente, l'interfaccia non è stata implementata in modo concreto; si è optato per un layout funzionale basato su più schede (tab), ciascuna dedicata a una specifica entità del modello. Sono state sviluppate le interfacce preliminari per le schede: `CategoryView`, `UserView` ed `ExpenseView`, definendo i metodi essenziali per il funzionamento dell'applicazione.

In questo modo si è potuto procedere con l'implementazione del Controller, anche in questo caso uno per ciascuna entità del modello: `CategoryController`, `UserController` ed `ExpenseController`. Ciascun controller ha un riferimento all'interfaccia che gestisce e al servizio con cui comunica, quindi avremo ad esempio che `CategoryController` ha un riferimento a `CategoryView` in cui presenta i dati che ottiene attraverso `CategoryService` (e viceversa, il controller invierà i dati ottenuti dall'interfaccia CategoryView a CategoryService per effettuare le operazioni di CRUD).

Durante la fase di sviluppo, le principali difficoltà tecniche sono emerse nella gestione delle classi di Expense. Come accennato in precedenza, per poter instanziare un'Expense, è necessario che siano presenti nel database almeno uno User e una Category: non deve essere possibile poter inserire una Expense se ciò non è soddisfatto perché porterebbe a problemi sulle dipendenze. È stato quindi essenziale implementare un metodo che verificasse la presenza di un User e di una Category nel database prima di poter aggiungere una Expense, per prevenire problemi legati alle dipendenze. È stata quindi implementata una validazione che assicura la corretta compilazione dei campi e la presenza delle entità necessarie nel database.

Per quanto riguarda l'interfaccia, ogni scheda è stata standardizzata con campi di testo, tabelle per visualizzare le entità e bottoni per operazioni di aggiunta e rimozione. L'unica eccezione è presente nella tab delle Expense: essa include infatti DropDownBox per selezionare User e Category. Tali DropDownBox sono aggiornate alla creazione dell'interfaccia e ogni volta che si va nella tab di Expense, in questo modo si è sempre sicuri di avere una visione aggiornata dello stato del database. Il tasto Add è abilitato nel momento in cui tutti i campi sono stati popolati, questo significa che deve essere stata scelta una Categoria, specificato un Utente, aggiunte la somma e la data della spesa.

Le altre funzionalità sono identiche alle altre tab.

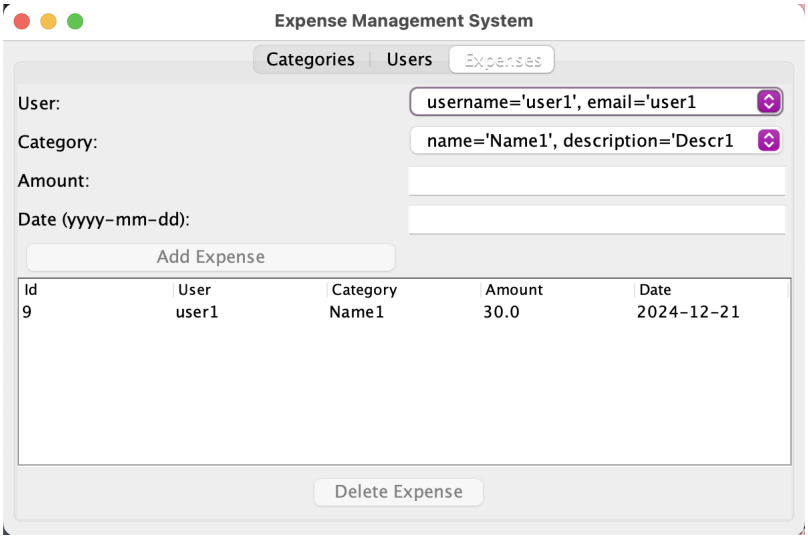

Figura 1: Screeshot della tab *Expenses* di Finance APP.

5.6.1 Testing

Per il testing d'unità delle classi controller, si è utilizzato **Mockito** per creare mock di Service e View per ogni entità, garantendo così indipendenza e testing in isolamento. Si è utilizzata l'implementazione di Mockito con Junit5 che prevede l'utilizzo di **@ExtendsWith** con la rispettiva **MockitoExtension.class**, l'annotazione dei mock con **@Mock** e l'iniezione delle dipendenze nel controlle con **@InjectMocks** per iniettare le dipendenze nei rispettivi controller. È stato predisposto un metodo di setup con annotazione **@BeforeEach** da avviare prima dei test di ciascuna classe che richiama il metodo initView del controller, che inizializza la vista e prevede inoltre il caricamento delle entità dal database in modo che il controller invii alla vista le informazioni più aggiornate rispetto alle entità presenti sul db.

Nei test d'unità si verifica che vengano richiamati i giusti metodi durante l'inizializzazione, l'aggiunta, eliminazione delle entità, testando sia i casi in cui siano presenti errori, sia che tutto vada come previsto. Si verifica inoltre che i listener vengano richiamati correttamente, sia sui bottoni di Add e Delete, sia quando si seleziona un elemento nella tabella. Per esempio quando si clicca sul tasto Add, il controller deve richiamare il rispettivo metodo di aggiunta dell'elemento, delegando però al Service l'effettiva persistenza.

Per quanto riguarda i controller sono stati sviluppati anche test di integrazione in cui, utilizzando un database simulato in un container, si verifica l'interazione del controller con le sue dipendenze: Service e View. Il database utilizzato in questo progetto è un database MySQL, nei test è stato quindi utilizzato un database con le stesse caratteristiche, sfruttando il supporto di TestContainer. Queste sono le annotazioni utilizzate per definire e avviare il container:

```
1 @Testcontainers
2 class CategoryControllerIT {
3
4     @Container
5     public static final MySQLContainer<?> mysqlContainer = new
    MySQLContainer<>("mysql:5.7")
6             .withDatabaseName("testdb")
7             .withUsername("test")
```

```
            .withPassword("test");
    private static EntityManagerFactory emf;
            ...

        @BeforeAll
    static void setUpTestClasses() {
        // Configure JDBC properties dynamically based on Testcontainers
        Map<String, String> overrides = new HashMap<>();
        overrides.put("javax.persistence.jdbc.url", mysqlContainer.
    getJdbcUrl());
        overrides.put("javax.persistence.jdbc.user", mysqlContainer.
    getUsername());
        overrides.put("javax.persistence.jdbc.password", mysqlContainer.
    getPassword());
        overrides.put("hibernate.dialect", "org.hibernate.dialect.
    MySQL5Dialect");  // MySQL dialect
        overrides.put("hibernate.hbm2ddl.auto", "create-drop");
        // Create EntityManagerFactory with these properties
        emf = Persistence.createEntityManagerFactory("TestFinanceAppPU",
    overrides);
    }

    ...
}
```

5.6.2 Testing GUI

Per il testing delle classi GUI è stato utilizzato **AssertJ Swing**. Come accennato, l'applicazione è stata impostata come un MainFrame avente 3 tab, quindi per il testing si è preferito **testare dapprima ogni tab a sé**, mockando per ogni classe il rispettivo controller, e poi il MainFrame, testando lì semplicemente solo quello che rimaneva, come ad esempio il caricamento dei dati alla creazione e allo switch delle tab.

Le 3 tab sono `CategoryPanel`, `UserPanel` ed `ExpensePanel`. Ciascuna di queste è impostata in modo simile, l'unica cosa diversa è presente nella tab Expense, in cui sono presenti due DropDownBox per la selezione dell'utente e della categoria della spesa che si vuole aggiungere. Il testing delle viste prevede il controllo delle condizioni iniziali, cioè lo stato delle textBox, Buttons e tables. Si procede andando a verificare che il contenuto dei dati sia aggiornato correttamente ed infine si va a controllare che l'intera tab funzioni come dovrebbe quindi per esempio, che i Buttons si attivino e disattivino correttamente, che le textBox vengano ripulite quando il dato è aggiunto, che la Label di stato sia aggiornata e mostri correttamente se l'azione è stata eseguita o, contrariamente, se ci sono stati errori.

Il testing quindi si focalizza prettamente sulla parte di presentazione dei dati, discernendo da tutta la Business Logic e la comunicazione con il database che infatti viene simulata tramite Mocking. Si accede al Mock solo per constatare che vengano chiamati i giusti metodi e che quindi le azioni che partono dall'inferfaccia siano **delegate correttamente al controller**, che produce una risposta che va a modificare a sua volta l'interfaccia agendo sugli elementi di testo o sulle Label.

5.7 End to End Testing con Cucumber

Nel contesto dello sviluppo del software, i test end-to-end (E2E) rivestono un ruolo cruciale per garantire che l'interazione tra le diverse componenti del sistema sia conforme alle aspettative. Per implementare tali test, è stato scelto di utilizzare il framework Cucumber, che permette di descrivere il comportamento atteso dell'applicazione in un linguaggio naturale, accessibile anche ai non sviluppatori. Di seguito è presentata una panoramica dettagliata dell'implementazione e dell'esecuzione dei test E2E utilizzando Cucumber.

5.7.1 Configurazione dei Test E2E

Il cuore della configurazione dei test E2E con Cucumber è rappresentato dalla classe `RunCucumberE2ETest`, che specifica le opzioni necessarie per eseguire i test:

```
    @RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/e2e/resources/features",
        glue = "it.unifi.financeapp.e2e.steps",
        plugin = {"pretty", "html:target/cucumber-report.html"}
)
public class RunCucumberE2ETest {
}
```

Questa configurazione dirige Cucumber a eseguire scenari di test definiti nelle feature specificate, utilizzando le definizioni di step trovate nel pacchetto specificato. Il plugin "pretty" migliora la leggibilità dell'output dei test, mentre il plugin "html" produce un report in formato HTML che documenta i risultati dei test.

5.7.2 Definizione degli Step di Test

I passaggi dei test sono definiti in classi separate per ciascuna entità del modello, come `CategorySteps`, `CommonSteps`, e `ExpenseSteps`. Questi passaggi traducono scenari scritti in linguaggio naturale in azioni concrete che interagiscono con l'applicazione. Ad esempio, nella classe CategorySteps:

```
public class CategorySteps {

    @Given("I am on the Category Management page")
    public void iAmOnTheCategoryManagementPage() {
        TestConfig.window.tabbedPane().selectTab(0);
    }

    @And("the category {string} exists")
    public void theCategoryExists(String categoryName) {
        TestConfig.window.table("entityTable").requireRowCount(1);
        assertEquals(categoryName, TestConfig.window.table("entityTable").
    target().getModel().getValueAt(0, 1));
    }
    ...
}
```

Questi passaggi definiscono le condizioni iniziali come l'accesso alla pagina di gestione delle categorie e verificano che una specifica categoria esista già nel sistema.

La configurazione e la pulizia dell'ambiente di test sono gestite nella classe CommonSteps, garantendo che ogni scenario inizi con una configurazione di base definita e che le risorse siano rilasciate al termine del test:

```
public class CommonSteps {

    @BeforeAll
    public static void setUp() {
        TestConfig.setUpClass();
    }

    @AfterAll
    public static void onTearDown() {
        TestConfig.tearDownClass();
    }
    ...
}
```

CommonSteps contiene e gestisce tutti gli step comuni, come l'inserimento di testo o la selezione di bottoni, facilitando la riutilizzabilità del codice e la manutenzione.

L'ambiente di test è configurato dinamicamente attraverso TestConfig, che utilizza un container MySQL gestito da TestContainers per simulare il database in un ambiente isolato, similmente a quanto fatto per i test d'integrazione.

```
public class TestConfig {
    @Container
    public static MySQLContainer<?> mysqlContainer = new MySQLContainer<>("
    mysql:5.7")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");
    ...
}
```

5.8 Validazione e Feedback

I test end-to-end implementati con Cucumber permettono di verificare non solo le funzionalità individuali ma anche le interazioni complesse tra le varie parti dell'applicazione, assicurando che l'intero sistema funzioni come un unico coerente organismo. Questi test, con la loro natura descrittiva e la loro stretta correlazione con i requisiti funzionali, si rivelano essenziali per validare la qualità del software in scenari d'uso reali, inoltre sono di facile comprensione anche da chi non è programmatore o un domain expert grazie all'utilizzo del BDD che permette la scrittura di test in linguaggio molto simile al linguaggio naturale.

6 Conclusioni

Il processo di sviluppo del progetto adottando il TDD ha portato alla realizzazione di un software di alta qualità, conforme alle aspettative iniziali e alle specifiche tecniche stabilite. L'approccio sistematico e disciplinato di TDD, insieme all'uso di GitHub Actions, SonarCloud e strumenti di copertura come JaCoCo e PITest, ha garantito risultati eccellenti sotto vari aspetti del processo di sviluppo e della qualità del codice.

L'implementazione di GitHub Actions ha permesso di automatizzare il ciclo di vita dello sviluppo software, compresi compilazione, testing e analisi statica del codice. Questo ha non solo semplificato il processo di integrazione continua ma ha anche assicurato che

ogni commit e pull request venissero validati in modo rigoroso, mantenendo così l'integrità del codice a un livello costantemente elevato.

Attraverso l'impiego di tecniche di testing avanzate, il progetto ha raggiunto una copertura del 100% sia in termini di code coverage che di mutation coverage. Ciò indica che ogni linea di codice e ogni potenziale mutazione del codice sono state testate, confermando l'efficacia e la completezza dei test implementati. Tale livello di dettaglio nel testing assicura che il codice sia non solo funzionale ma anche robusto di fronte a modifiche e potenziali regressioni.

In conclusione. l'adozione del TDD ha guidato lo sviluppo del software in una direzione orientata alla qualità fin dalle fasi iniziali, rendendo il codice più modulare e testabile. Inoltre, ha facilitato un ciclo di feedback rapido durante lo sviluppo, permettendo di identificare e correggere gli errori con maggiore efficienza.