Shop Totem Application Report
Advanced Techniques and Tools for Software Development

Claudia Raffaelli

Matricola: 7036222
Department of Information Engineering
University of Florence
Via di Santa Marta 3, Florence, Italy
claudia.raffaelli@stud.unifi.it

Francesco Scandiffio

Matricola: 7033974
Department of Information Engineering
University of Florence
Via di Santa Marta 3, Florence, Italy
francesco.scandiffio@stud.unifi.it

***Abstract*—In questo report viene presentato Shop Totem [1], un'applicazione desktop che simula un totem interattivo di un supermercato attraverso cui poter comprare e restituire dei prodotti. Il report descrive il design e discute alcune scelte implementative dell'applicazione, con riferimenti ai patterns e tools utilizzati. Una particolarità di questa applicazione è quella di poter essere avviata, alternativamente, con un database relazionale o con uno non relazionale.**

***Keywords*—TDD, Java, Testing, MongoDB, MySQL, Maven, Docker, CI, Code Quality.**

I. Introduction

Shop Totem è un'applicazione desktop che simula un sistema di acquisto di prodotti in supermercato. L'implementazione replica alcuni vincoli del mondo reale, come ad esempio limitare la quantità di prodotti acquistabili in base allo stock disponibile in magazzino (il database). Un'altra caratteristica di questa applicazione è quella di poter utilizzare sia un database relazionale che con un database non relazionale, nello specifico MySQL e MongoDB. L'applicativo è un Maven project sviluppato in Java 8 e realizzato secondo TDD. Il sistema fa anche uso dei seguenti patterns:

- *Model-View-Controller*: definisce le interazioni tra vista e Service Layer
- *Repository*: nasconde l'implementazione concreta dei metodi utilizzati per accedere ai dati stored sul database.
- *Service Layer*: implementa la logica con cui manipolare i dati e le classi di modello
- *TransactionManager*: regola le transazioni, confinando al loro interno tutte le operazioni che richiedono accesso al database.

II. System design

In questa sezione sono descritte le singole componenti dell'applicazione.

A. Domain Model

Il Domain Model è composto da quattro classi che corrispondono a quattro tipi di entità persistibili sul database: Product, Stock, Order e OrderItem (Fig. 7).

[1] https://github.com/FrancescoScandiffio/shop-totem

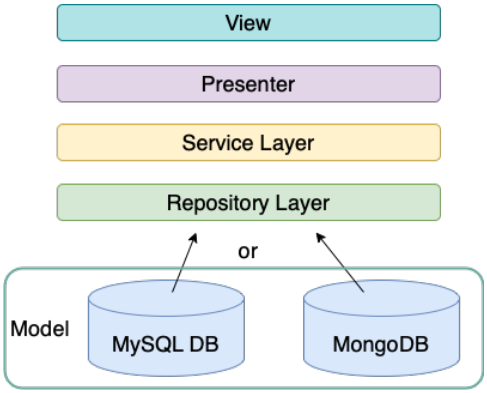

Figure 1: Overview progetto.

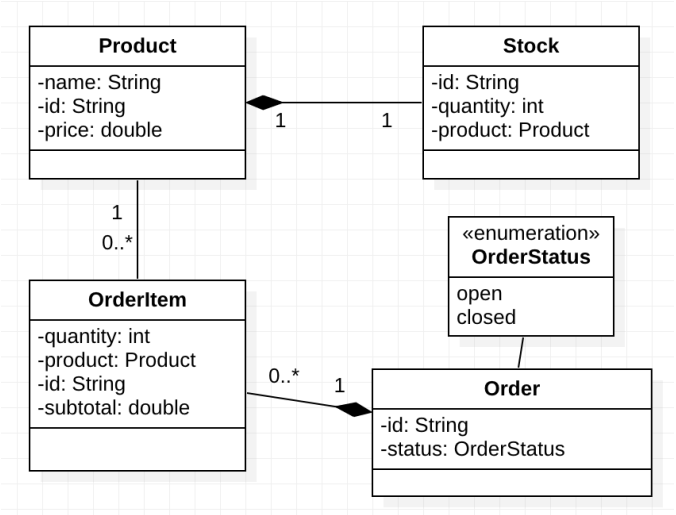

Figure 2: Class diagram.

- ***Product***: contiene le informazioni su nome e prezzo di un prodotto.
- ***Stock***: contiene la quantità disponibile in magazzino riferita ad un prodotto. Ogni Stock ha un riferimento ad un solo Product e non possono esistere più Stock per lo stesso prodotto (1:1 unidirezionale).
- ***Order***: permette di tenere traccia dei prodotti acquistati

durante una sessione di shopping. L'ordine può avere due stati, definiti attraverso una Enumeration: aperto o chiuso.
- ***OrderItem***: rappresenta un prodotto all'interno del carrello del cliente, con informazioni circa la quantità acquistata e il costo. Ogni OrderItem fa riferimento ad un solo Product e clienti diversi possono acquistare lo stesso prodotto (se disponibile), risultando così in una relazione many to one. La classe OrderItem dispone di un attributo subtotal utilizzato per mostrare il costo di un oggetto nel carrello, in relazione alla quantità acquistata. Poiché il valore di *subtotal* è calcolabile dalla quantità nel carrello e dal prezzo del prodotto, informazioni reperibili navigando i riferimenti, sarebbe ridondante salvarlo sul database: per questo motivo il campo è transiente.

Per questo progetto è stato utilizzato un modello anemico, ovvero privo di funzioni con una qualche logica. Le classi delle Entità sono dei semplici contenitori di dati mentre la logica di business è implementata altrove.

B. Repository Layer

Come anticipato, Shop Totem può essere avviata scegliendo tra due database di tipo diverso: uno relazionale (MySQL) e uno non relazionale (MongoDB). Un sistema del genere richiede una adeguato livello di astrazione, motivo per cui è stato implementato il Repository Pattern. Abbiamo prima definito delle interfacce pubbliche per esporre i metodi di accesso ai dati sul database e poi ne abbiamo realizzato due implementazioni specifiche, una per ogni database.

C. Transaction Manager

Un transaction manager nasce dall'esigenza di permettere l'esecuzione sul database di un insieme raggruppato di operazioni, che prende il nome di *database transaction*. Dunque una transazione simbolizza una unità di lavoro eseguita su un database. Le transazioni sono necessarie per fare in modo che:

- Programmi o parti di programma che occedono al database in modo concorrente lo facciano in isolamento.
- Operazioni multiple appartenenti a diverse transazioni vengano svolte in modo consistente e indipendente.
- Se si ha un fallimento in almeno un'operazione all'interno di una transazione, tutte le operazioni precedenti vengono annullate: si ha il rollback. Pertanto si dice che le transazioni sono atomiche. Senza questa caratteristica, se l'esecuzione si ferma in modo prematuro o inaspettato con operazioni non portate a termine, il database rimane in uno stato non chiaro.

Dunque se una transazione viene eseguita con successo nella sua totalità verrà salvata, *committed*, sul database ed i cambiamenti avranno effetto. Altrimenti, viene eseguita l'azione di rollback, ripristinando lo stato pre-transaction.

D. Service layer

Il service layer ShoppingService contiene la logica di validazione dell'applicazione, controllando che gli oggetti con cui fare operazioni sul database siano validi. Controlla anche che le richieste di acquisto e rimozione di prodotti da parte dell'utente siano possibili date le attuali quantità in magazzino. Interagisce con il Transaction Manager per la gestione delle operazioni con il database. Inoltre supponiamo che i Prodotti, una volta inseriti all'interno del database non verranno più modificati. Per quanto riguarda gli Stock, la quantità viene modificata unicamente ogni qualvolta un utente inserisce o rimuove un prodotto dell'ordine.

E. Controller layer

In ShopTotem il controller prende il nome di TotemController. Esso è l'unico soggetto ad interagire con la vista, dalla quale riceve dei messaggi "triggerati" da azioni dell'utente. La vista non è dotata di una logica complessa, pertanto è il controller che, ogni qualvolta riceve una comunicazione dalla vista, risponde fornendo istruzioni su cosa fare. Per assolvere alle richieste della vista TotemController comunica con ShoppingService che fa da intermediario con le repository e gestisce le transazioni.

F. GUI

Abbiamo realizzato l'interfaccia usando la libreria Swing, e utilizzando AssertJ Swing come strumento di testing. L'interfaccia grafica è divisa nei pannelli principali:

- *Welcome page*: è il pannello iniziale. Contiene un pulsante per procedere all'avvio dello shopping, e che conduce quindi allo shopping panel.
- *Shopping page*: contiene la lista dei prodotti presente nel database, mostrandone il nome e il prezzo. Selezionando un prodotto è possibile procedere al suo inserimento all'interno del carrello. Da questa schermata è possibile navigare alla pagina di welcome, annullando l'ordine, oppure procedere alla visualizzazione del carrello.
- *Cart page*: qui sono mostrati i prodotti attualmente nel carrello, con nome, prezzo, quantità e subtotal. Questi possono essere rimossi in tutto o in parte dal carrello. Da questa pagina è possibile tornare alla schermata di shopping, così come annullare l'ordine o procedere al checkout.
- *Goodbye page*: successivamente al checkout l'utente viene condotto alla pagina di goodbye su cui è posto un pulsante che permette di aprire un nuovo ordine e che condurrà alla pagina di shopping.

Le schermate principali dell'applicazione sono illustrate in Fig. 3.

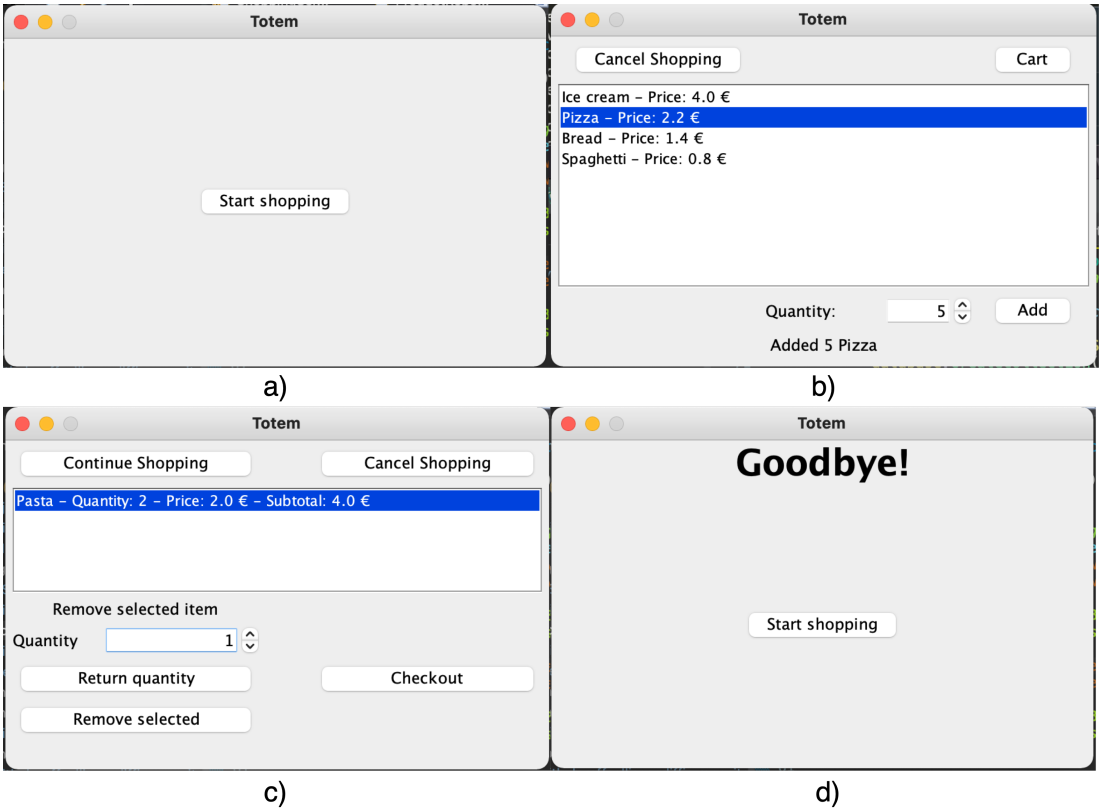

Figure 3: Le schermate: welcome a), shopping b), cart c) e goodbye d).

Mostriamo alcune delle immagini più esplicative delle schermate di shopping e cart:

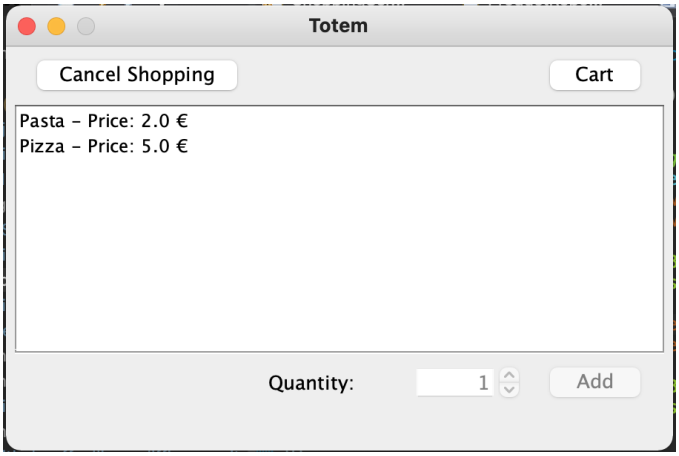

Figure 4: Nessun prodotto selezionato: pulsante di Add disattivato.

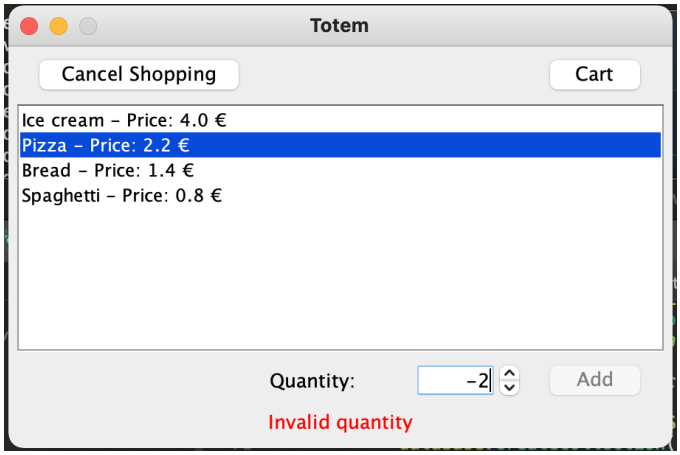

Figure 5: Inserimento di una quantità invalida per l'acquisto di un prodotto.

III. Implementation details

Vediamo qui i principali dettagli implementativi adottati nella nostra applicazione.

A. Assegnazione degli ID

Avendo due databases intercambiabili abbiamo optato per l'adozione di un campo ID in ciascuna classe di modello, di un tipo che fosse compatibile per entrambi: una String. Nel caso di MongoDB è il database stesso che al momento della creazione di un oggetto genera un id di tipo ObjectID. Viene riservato all'implementazione Mongo della repository di convertire l'ObjectID in String al momento del mapping del documento all'oggetto di classe di modello.

Per quanto riguarda il database MySQL, utilizzando Hibernate come strumento per la mappatura fra domain model e database, è esso stesso ad occuparsi della generazione

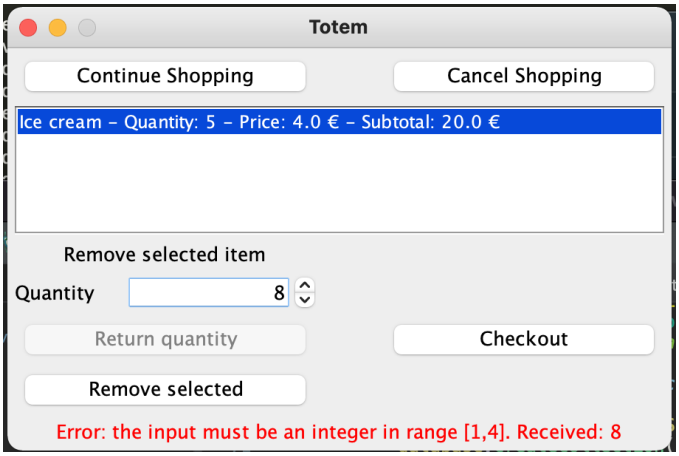

Figure 6: Inserimento di una quantità invalida per la rimozione (più di quanto inserito nel carrello)

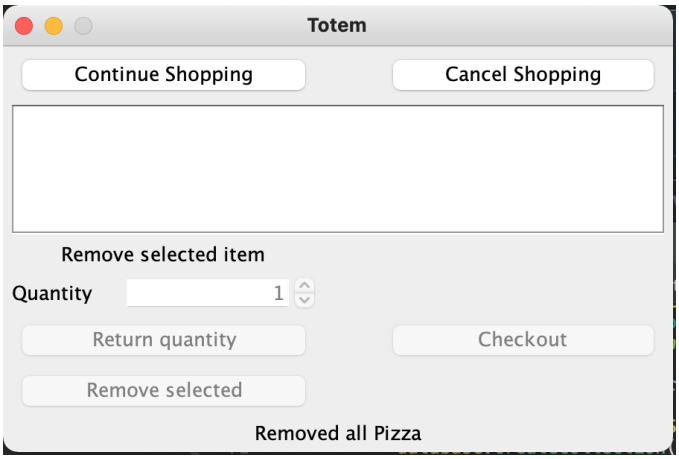

Figure 7: Rimozione di tutto il record dal carrello

degli ID. In genere quando non sono richieste operazioni fra tabelle diverse, viene fatto in modo che i valori della colonna primaria siano numerici, auto-incrementali ed unici a livello di tabella. Ciò è possibile seguendo la strategia IDENTITY per la generazione degli id. Quando invece si prevede un utilizzo incrociato delle tabelle per fare in modo che gli id siano unici a livello di database solitamente viene fatto ricorso agli UUID. Non sono, gli identificatori generati da UUID sono universalmente unici. Tuttavia in linea di massima, l'utilizzo degli UUID come chiave primaria non è efficiente, vista la loro lunghezza, data da 128 bit. Dunque idealmente potrebbero essere utilizzati due identificativi:

- Uno unico a livello di tabella, come può essere un int o un long auto-incrementale. Ciò permette di eseguire operazioni a livello di tabella in modo molto efficiente.
- Uno unico a livello di database, come uno UUID che dunque ha una garanzia di univocità assoluta a livello di database, ma pecca in efficienza.

Poichè tuttavia era nostro interesse mantenere una compatibilità fra l'implementazione adottata per MongoDB e MySQL abbiamo rinunciato a questa ottimizzazione, conservando l'utilizzo di un unico ID, come UUID. Gli UUID possono essere generati secondo diverse strategie. Quella da noi adottata è la IETF RFC 4122 compliant (variant 2).

In conclusione, utilizzando il tipo ObjectID e uno UUID2 come identificativi nel database, e convertendo gli stessi in String al momento della mappatura nelle classi di modello riusciamo a mantenere una consistenza fra le due implementazioni di una stessa repository.

B. Hibernate XML mapping

Nel caso del database MySQL, affiché Hibernate possa eseguire il lavoro di mappatura, deve essere accompagnato da delle istruzioni su come poter collegare le classi alle tabelle del database. Le annotazioni di Hibernate sono il metodo più moderno per definire la mappatura. Il vantaggio dell'uso delle annotazioni è che tutti i metadati si trovano all'interno della classe, il che aiuta a capire meglio la struttura della tabella e delle classi, in modo simultaneo.

Tuttavia poiché le classi di modello devono essere condivise anche con il database Mongo, abbiamo optato per un soluzione diversa, che non andasse ad alterare il codice comune. Abbiamo fatto ricorso ad una mappatura definita all'interno di un documento XML. Tutti i file di mappatura realizzati si trovano all'interno della cartella `/src/main/resources/META-INF` e sono caratterizzati da estensione `.hbm.xml`. Lo scopo di questa tipologia di files è lo stesso delle annotazioni, ma in questo caso il vantaggio è che le istruzioni proprie di Hibernate sono disaccoppiate dalla classe di modello.

C. Persistence xml

Nel caso di database MySQL, Hibernate prevede l'utilizzo della EntityManager API, per accedere al database. E' utilizzato per creare e rimuovere delle entità persistenti, per trovare entità con la chiave primaria e in generale per eseguire query sulle entità. Il persistence.xml fornisce la flessibilità per configurare adeguatamente l'EntityManager.

In particolare, il persistence.xml è un file di configurazione per JPA che definisce una persistence-unit con un nome univoco. La persistnece unit definisce un insieme di classi in relazione

ShopTotem dispone di due persistence files, uno dedicato ai test ed uno alla produzione. Entrambi dispongono di un tag `<mapping-file>` che va ad indicare la posizione dei file di annotazione delle classi di modello, così che possano essere recuperati per la mappatura. Inoltre viene indicato l'utilizzo del dialect MySQL8Dialect per la comunicazione con il database ed il tipo di driver con cui fare la connessione con il database.

D. Transaction Manager implementation

Per la gestione delle transazioni dunque abbiamo creato un Transaction Manager che è costituito dalle seguenti componenti:

- Una interfaccia `TransactionManager`, essenziale per permettere la sostituzione veloce dei database. Questo è inevitabile, dato che diversi database gestiscono in modo diverso le transazioni. Questa interfaccia contiene il metodo

`runInTransaction` il quale accetta un solo argomento di tipo `TransactionCode` di cui parliamo in seguito.

- Le due implementazioni di `TransactionManager` per Mongo e MySQL. In particolare:
  - Per MySQL viene fatto uso della gestione di transazioni offerta dal JPA `EntityManager` che consente l'accesso ad un persistence context su cui eseguire le operazioni in modo intermedio prima del commit sul database. L'`EntityManager` viene iniettato alle mysql repositories dal `TransactionManager`.
  - Per quanto riguarda Mongo invece viene usata la `ClientSession` la quale incapsula una sessione attiva per uso del client corrente. Come per l'`EntityManager`, anche la sessione viene iniettata alle mongo repositories che la useranno in fase di query del database.
- La `@FunctionalInterface TransactionCode`, i.e. una interfaccia con un solo metodo astratto. Questo metodo non è altro che una lambda function, la quale contiene il codice che dovrà essere eseguito all'interno della transazione.
- Una `TransactionException` la quale estende `RuntimeException` da poter essere usata in fase di rollback se qualcosa va storto con la transazione.

Dunque l'idea è che per eseguire del codice all'interno di una transazione deve essere chiamato il metodo `runInTransaction` passando come argomento il codice da eseguire. L'applicazione del codice viene effettuata prendendo in ingresso tutte le repository dell'applicazione, una cui nuova istanza è creata ad ogni transazione. Se tutte le operazioni sono eseguite con successo sarà compito della sessione o dell'entity manager di fare il commit sul database. Altrimenti, se un'eccezione viene lanciata dal codice eseguito, questa viene catturata dal transaction manager che si adopererà per eseguire il rollback.

Infine notiamo che, seppur i test di `TransactionManager` lo riguardino esclusivamente, dovendo utilizzare delle implementazioni reali dei database verranno da noi considerati non come unit tests, ma più propriamente come integration tests.

E. Vincoli e gestione degli errori

L'applicazione è stata implementata seguendo il domain model anemico, che prevede la non inclusione di business logic all'interno di domain objects. Pertanto azioni come validazioni degli argomenti sono tutte eseguite dal service layer. Inoltre i valori che potrebbero finire all'interno di classi di modello, derivanti da input dell'utente rappresentano ovviamente una criticità. Nel nostro caso l'utente può inserire una quantità richiesta di prodotto da acquistare, ed una quantità di prodotto da restituire che deve essere rimossa dal carrello. Entrambi questi input sono validati direttamente sulla vista, facendo uso dell'oggetto `JSpinner`

Infatti, entrambi gli spinner fanno riferimento ad un modello, dotato di un vincolo sui valori minimi, che non possono essere inferiori ad 1. Questo garantisce che non possano essere richiesti degli acquisti negativi o di valori zero. Allo stesso modo è garantito che vengano accettati solo numeri grazie all'implementazione di un filtro RegEx. In ogni caso non è possibile per l'utente richiedere aggiunte e rimozioni con valori non validi poiché su spinner è aggiunto un listener che attiva e disattiva i pulsanti in base al contenuto immesso. I listener sono anche posti a rilevare la selezione e deselezione del contenuto delle liste e ad agire di conseguenza.

Questo non impedisce all'utente di richiedere l'acquisto di prodotti che però non sono disponibili in magazzino. In tal caso, il controller, comunicando con il service layer metterà a conoscenza la vista dell'errore che viene così mostrato in interfaccia. Degli errori sono mostrati anche in tutti quei casi in cui si verifichi un'eccezione che ha una rilevanza per l'utente.

F. JUnit5 GUITestExtension

Utilizzando JUnit5 come unit testing framework abbiamo dovuto rinunciare all'uso del `GUITestRunner` per l'esecuzione dei test d'interfaccia. Abbiamo dunque fatto ricorso ad una `GUITestExtension` [2]. Proprio come il `GUITestRunner` di JUnit4, questa estensione è in grado di effettuare degli screenshot dei failed GUI tests. Il setup ha richiesto qualche impostazione manuale, quale la registrazione della extension, annotando la classe di test con `@ExtendWith(GUITestExtension.class)` e l'istallazione di `FailOnThreadViolationRepaintManager` prima dell'inizio dei test. Questo ultimo controlla che tutto l'accesso alle componenti Swing avvenga nell'EDT.

G. GitHub Repository

Come Version Control System abbiamo adottato GitHub, il che ci ha anche permesso di operare con una collaborazione più efficiente. Per ogni nuova feature aggiunta e modifica alla codebase abbiamo adottato il procedimento di creazione di nuovi branch, integrati sul main branch attraverso Pull Requests (PR). Abbiamo inoltre sperimentato con l'apertura di Issues e relativa chiusura con PR. Inoltre, grazie a GitHub abbiamo potuto accedere allo strumento di Continua Integrazione, ottimo per l'esecuzione di build automatizzate ed esecuzione di test.

H. Continuous Integration (CI) using GitHub Actions

Poiché come detto, la repository contenente il nostro codice si trova su, GitHub abbiamo optato per GitHub Actions come servizio di CI. Per configurare GitHub Actions è necessaria la realizzazione di file YAML di workflow, che permettono l'esecuzione di processi automatizzati. I file `.yml` di configurazione devono essere posti all'interno di un'apposita cartella `.github/worflows`, localizzata nella project root directory. Per la nostra applicazione abbiamo realizzato due workflows che definiamo di seguito:

- **maven_ubuntu.yml** è il workflow principale. E' eseguito ogni qualvolta viene eseguita un'azione di push o pull request sulla repository, eccetto nei casi in cui la modifica apportata riguardi unicamente file di licenza, workflow o

[2]https://github.com/assertj/assertj-swing/issues/259

readme. Opera su ambiente Ubuntu ad è configurato con Java11.

- **maven_macos.yml** è un workflow secondario, triggerato sulle sole azioni di pull request, eccetto per quei casi sopra menzionati. E' configurato per l'utilizzo di Java8 e si occupa unicamente dell'esecuzione dei test di unità e integrazione. Inoltre inizialmente nella definizione del workflow veniva impiegata una matrix strategy contenente i due sistemi operativi Windows e MacOS. Questo ci permetteva di creare due jobs, in ambiente del rispettivo sistema operativo, con una single job definition. Tuttavia, non potendo disporre dell'immagine docker di mysql per Windows, abbiamo deciso di rinunciare a questo SO nella CI.

Entrambi i workflow sfruttano il sistema di caching delle dipendenze di Maven in modo da velocizzare il processo di build. Il file principale è diviso in più step. Ripercorriamo brevemente i più salienti:

1) Sono eseguiti i test di unità con il comando *xvfb-run*. Questo permette di eseguire l'applicazione in un ambiente "headless".
2) Esegue i mutation test di PIT solo in caso di pull requests e unicamente se l'outcome al passo precedente, riguardante l'esecuzione dei test di unità va a buon fine. Non avrebbe infatti senso eseguire questa tipologia di test su una codebase con test di unità non-green.
3) Sono eseguiti i test di integrazione, anche questi attivando lo schermo virtuale. Facciamo notare che pur eseguendo il comando `mvn verify`, i test di unità sono saltati. Questo è reso possibile dall'utilizzo del parametro `<skipTests>` nella sezione di configurazione del `maven-surefire-plugin`, responsabile dell'esecuzione degli unit tests. Impostando a true `<skipTests>` dallo step di GitHub Actions dunque i test di unità non sono eseguiti. Abbiamo optato per questa configurazione per fare in modo che tali test non siano eseguiti due volte, essendo questi già ottenuti allo step precedente. Inoltre durante questo step viene anche verificata la quality code con SonarCloud e generato il report di Coverall, il quale tiene traccia della code coverage attraverso i report di JaCoCo. Sia SonarCloud che Coverall sono collegati a GitHub Actions grazie all'utilizzo di un apposito token, salvato come GitHub Secret. Inoltre in questo workflow facciamo uso di Java11 anziché 8 come per l'altro workflow e per la build in locale di Maven, per avere compatibilità con SonarCloud.
4) Sono poi generati i report per gli unit test unicamente se lo step riguardante i test di unità fallisce.
5) Allo stesso modo sono generati i report per integration test, se questi falliscono.
6) Viene indicato di archiviare, se generati, i JUnit Reports nella cartella `**/target/site`, altrimenti dopo l'esecuzione del workflow non sarebbero più accessibili.
7) Come per i JUnit Reports, vengono archiviati anche i PIT Reports se i mutation test sono falliti.
8) Infine sono salvati anche gli eventuali screenshots prodotti durante i test falliti legati all'interfaccia. Se nessuno screenshot è stato prodotto, questo step viene ignorato grazie al comando `if-no-files-found: ignore`, il quale sopprime il warning legato alla non presenza di nessun file da archiviare nella cartella definita.

I. Testcontainers vs Docker containers with Maven build

Per i test riguardanti i database MongoDB e MySQL inizialmente utilizzavamo Testcontainers. L'utilizzo di Testcontainers è subordinato al fatto che il Docker daemon sia in esecuzione. Ci penserà poi Testcontainers stesso a creare e far partire il container contenente il database.

Tuttavia in vista dei test E2E in stile BDD, non essendo Testcontainers di facile impostazione per questo tipo di test, abbiamo deciso di passare all'avvio di containers legato alla fase pre-integration-test e post-integration-test. Sono dunque creati due Docker containers rispettivamente con MySQL e con MongoDB in modo del tutto automatizzato, che vengono utilizzati per svolgere i test. Se da una parte l'immagine di MySQL è quella ufficiale, dall'altra per MongoDB abbiamo fatto ricorso ad un'immagine che implementa una Replica-set per MongoDB[3]. In particolare un replica set in MongoDB è un gruppo di processi mongod che mantengono uno stesso insieme di dati. I replica set forniscono ridondanza e sono fondamentali per poter utilizzare le transazionei di Mongo. L'immagine che abbiamo utilizzato crea tre repliche.

Come già detto Maven ci permette di automatizzare il processo di build e testing. Tuttavia se desideriamo utilizzare l'applicazione in produzione sarà necessario preventivamente disporre dei database MySQL o MongoDB oppure avviare i relativi Doker containers.

Suggeriamo i seguenti comandi per farlo:

- MySQL:

```
docker run -d -p 3306:3306 -e MYSQL_DATABASE="
    totem" -e MYSQL_ROOT_PASSWORD="" -e
    MYSQL_ALLOW_EMPTY_PASSWORD="yes" mysql
    :8.0.28
```

- MongoDB:

```
docker run -d -p 27017:27017 -p 27018:27018 -p
    27019:27019 candis/mongo-replica-set
```

J. Behaviour Driven Development (BDD)

Come strumento di Behaviour Driven Development (BDD) abbiamo adottato JBehave. Esso permette di specificare un textual story file, il quale esprime un comportamento da verificare. Ad ogni storia è collegato almeno uno scenario composto a sua volta da una collezione di steps.

K. Come avviare ShopTotem

ShopTotem può essere avviato con un semplice `mvn verify` . In produzione devono essere preventivamente fatti partire i container come descritto in precedenza.

[3]https://github.com/CandisIO/mongo-replica-set