Progetto ATTSW - My Shop

Lorenzo Giannella

October 28, 2023

1 Introduzione

L'elaborato presenta un prototipo di applicazione di e-commerce. L'applicazione consente all'utente di esplorare una lista di prodotti disponibili in un negozio e, contemporaneamente, di gestire un carrello per l'acquisto di tali prodotti. Per mantenere la semplicità dell'applicazione non è stata implementata una funzionalità di pagamento.

L'applicazione è basata su un'architettura Model-View-Presenter (MVP). Questo approccio separa chiaramente le responsabilità tra le componenti dell'applicazione e favorisce la modularità e la manutenibilità del codice.

- **Model**: Il modello dell'applicazione comprende la rappresentazione dei dati, come i prodotti e il carrello dell'utente. Questi dati sono gestiti in un database.
- **View**: La vista dell'utente è l'interfaccia grafica dell'applicazione. In questo caso, ci sono due liste visualizzate all'utente: una lista di prodotti disponibili e una lista del carrello dell'utente insieme ai bottoni necessari per effettuare i casi d'uso.
- **Presenter (Controller)**: Il presenter (chiamato anche controller) funge da intermediario tra la vista e il modello. Contiene la logica dell'applicazione, inclusa la gestione degli eventi utente e le interazioni con il database.

Questa relazione documenta il processo di sviluppo dell'applicazione e mette in evidenza l'approccio di sviluppo **Behavior-Driven Development, BDD** utilizzato per garantire la qualità e la correttezza dell'applicazione. Verranno descritte in dettaglio le fasi di sviluppo, inclusi gli Unit Test, gli Integration Tests e gli End-e-end tests.

1.1 Setup dell'ambiente di sviluppo

- **JUnit**: framework di test.
- **AssertJ**: test dependency.
- **Mockito**: framework utilizzato per il Mocking.
- **MongoDB**: database per implementare la Repository.
- **Picocli**: parser per command line Java.
- **PIT**: framework per il mutation testing.
- **Cucumber**: tool per il supporto del Behavior-Driven Development

In questo elaborato il processo di Continous Integration è implementato utilizzando:

- **GitHub** come Repository host.
- **GitHub Actions** come CI server, dove utilizzeremo **Maven** per eseguire gestire le dipendenze, compilare e testare il progetto e generare i reports.
- **Jacoco & Coveralls** per memorizzare e analizzare il code coverage.
- **SonarCloud** per eseguire l'analisi della qualità sul codice

2 Behavior-Driven Development, BDD Cycle

Nel processo di sviluppo di questa applicazione, ho adottato il Behavior-Driven Development (BDD) come metodologia principale. Il BDD è un approccio che si concentra sulla descrizione del comportamento del software in termini di scenari di utilizzo, prima di procedere all'implementazione. Questo approccio mette l'accento sulla comunicazione chiara tra il team di sviluppo e gli stakeholder, garantendo che tutti abbiano una comprensione comune dei requisiti e delle aspettative del software.

Il ciclo di sviluppo BDD seguito nel progetto comprende diverse fasi chiave:

- **Definizione dei Casi d'Uso**: all'inizio del processo, ho identificato e definito una serie di casi d'uso chiave che rappresentano le interazioni principali dell'utente con l'applicazione.
- **Scrittura delle Specifiche**: per ogni caso d'uso identificato, ho scritto le specifiche utilizzando la sintassi **Gherkin**, un linguaggio di descrizione dei comportamenti leggibile da chiunque. Le specifiche Gherkin definiscono il comportamento atteso dell'applicazione in termini di "features" e "scenari".
- **Automatizzazione dei Test BDD**: le specifiche Gherkin sono state utilizzate per automatizzare i test BDD utilizzando uno strumento di testing compatibile con BDD, come **Cucumber**. Questi test BDD servono come documentazione e come insieme di test automatizzati che verificano che l'applicazione risponda correttamente ai comportamenti definiti nelle specifiche.
- **Implementazione**: dopo aver scritto le specifiche e automatizzato i test BDD, si procede con l'implementazione dell'applicazione. Il codice è stato sviluppato in modo incrementale, rispettando la **Test Pyramid**, implementando prima gli Unit tests e successivamente Integration e End-to-end tests.
- **Continuous Integration**: durante tutto il ciclo di sviluppo, è stata praticata la filosofia dell' "integrazione continua". Ogni modifica al codice è stata seguita da una esecuzione completa dei tests per rilevare tempestivamente eventuali regressioni.

2.1 Scenario di Esempio

Un esempio concreto di come ho applicato questa metodologia, insieme alle fasi di testing, è illustrato attraverso il seguente scenario, scritto con **Gherkin** in un **Cucumber feature file** specificando gli **steps**:

Feature: Shop View High Level

Specifications of the behavior of the Shop View

Background:

Given The database contains a few products

And The shop view is shown

Scenario: Add a product to cart

Given The user selects a product from the shop list

When The user clicks the "Add to Cart" button

Then The cart list contains the product

Questo scenario verifica il corretto funzionamento dell'aggiunta di prodotti al carrello. Vengono successivamente creati i file DatabaseSteps.java e ShopSwingSteps.java con i rispettivi tests.

2.1.1 Nota: imprecisione nel primo ciclo di sviluppo

Nel primo ciclo di sviluppo BDD non sono stati implementati i failing steps prima di procedere all'implementazione, anche se il processo TDD è stato correttamente rispettato durante tutto il ciclo di sviluppo per quanto riguarda Unit, Integration e End-to-end tests. Questo errore è stato corretto dal successivo ciclo.

3 Domain Model

Il domain model di questo elaborato è molto semplice e comprende due classi/entità:

- **Product**: caratterizzato da un nome e un "id", il cui scopo è rappresentare la chiave primaria di ricerca nel database e unica per il prodotto.
- **Cart**: caratterizzato da un "id" e una lista di prodotti.

Entrambe le classi hanno i propri metodi getters, setters, equals, hashCode, toString.

4 Unit Tests

Il primo passo nel ciclo di sviluppo è quello di definire Unit tests per le singole componenti. Importante è che queste siano testate in isolamento.

4.1 ShopControllerTest

Questi Unit tests hanno la responsabilità di implementare correttamente la classe ShopController, ovvero il Controller dell' applicazione: il suo compito è di processare le richieste degli utenti dalla View delegando al database le operazioni sulla Repository e di presentare il risultato all'utente delegando il lavoro alla View. Come test dependency sono utilizzati AssertJ e Mockito per il mock della Repository e della View, che per il momento sono interfacce.

Per la corretta implementazione dello scenario, nel rispetto della metodologia TDD, vengono scritti i test, che in un primo momento falliscono, per poi procedere alla corretta implementazione degli stessi nella classe ShopController. Vengono quindi implementati solo i metodi necessari alla realizzazione dello scenario d'esempio:

- **allProducts**: delega alla Repository il compito di trovare i prodotti nel database e alla View il compito di mostrarli nel negozio.
- **getCart**: delega alla Repository il compito di trovare il carrello nel database e alla View il compito di mostrarlo.
- **addProductToCart**: dato un prodotto da acquistare presente nel database, delega alla Repository il compito di trovare il prodotto nel database e spostarlo nel carrello e alla View di visualizzarlo.

L'ultimo metodo viene implementato solamente nel caso positivo in quanto è necessario testare e implementare solamente i metodi che permettono allo scenario di avere successo. Il caso d'uso in cui il prodotto non è presente nel database con il messaggio di errore associato sarà trattato nel successivo ciclo.

4.2 ShopMongoRepositoryTest

Questi Unit tests mirano a una corretta implementazione di ShopRepository usando **MongoDB**. Non essendo consigliabile utilizzare il mocking per testare il database, viene posta la scelta di utilizzare un in-memory database o di testare direttamente su una vera istanza di MongoDB utilizzando TestContainers o Docker. Trattandosi di Unit tests ho quindi scelto di optare per la prima opzione in quanto le operazioni CRUD effettuate sono molto semplici. Le ultime due opzioni infatti che sono considerate a tutti gli effetti Integration tests in quanto comunicano con un database reale.

Sempre seguendo la metodologia TDD, di seguito i metodi implementati per la realizzazione dello scenario:

- **findAllProducts**: restituisce i prodotti contenuti nel database sottoforma di lista.
- **findProductById**: restituisce il prodotto il cui id corrisponde a quello in ingresso.
- **findCart**: restituisce il carello il cui id corrisponde a quello in ingresso insieme ai prodotti contenuti nella sua "productList".
- **moveProductToCart**: trova il prodotto desiderato nel database, lo elimina dalla Collection dei prodotti e lo aggiunge alla "productList" del carrello desiderato.

4.2.1 Nota: errori nel ciclo di sviluppo

Il primo ciclo di sviluppo contiene due errori di implementazione. Entrambi sono stati scoperti e corretti grazie agli Integration Tests che hanno evidenziato il problema.

- **findCart**: In un primo momento il metodo era stato implementato per restituire solamente l'oggetto Cart desiderato. Successivamente, nel test riguardante il metodo *getCart* di ShopSwingViewIT, è emersa la necessita del metodo di restituire anche l'insieme di prodotti ad esso associati. Questo viene realizzato ottenendo dal database la lista di documenti "productList" associati al documento (carrello) di interesse e aggiungendoli, una volta convertiti in prodotti, all'oggetto Cart.

  La prima versione richiedeva l'uso di un suppressWarning("unchecked") dovuta al cast:

  ```
  List<Document> productList =
   (List<Document>) d.get(‘‘productList’’);
  ```

  Quindi è stata successivamente corretta per effettuare il Type Check:

  ```
  List<Document> productList =
   d.getList(‘‘productList’’, Document.class);
  ```

- **moveProductToCart**: Questo errore è dovuto ad una scelta di implementazione. In un primo momento, pensando a una reale implementazione di un negozio di e-commerce, ho pensato che il prodotto dovesse essere visibile nel carello senza effettivamente effettuare modifche al database prima di aver effettuato il checkout. Questa scelta avrebbe portato ad una eccessiva complessità nell'applicazione, rendendo il prodotto visibile sia nel carello che nel negozio. Per rispettare le linee guida ho scelto di mantenere l'implementazione semplice ed evitare un codice troppo complesso. Questo metodo viene implementato in un ciclo successivo, quando la necessita di avere una effettiva modifica nel database viene evidenziata nella classe ModelViewControllerIT. Viene quindi creata un'appositàbranch su gitHub, aprendo una "issue" con il tag "bug".

4.2.2 Utilizzo di Filters e Updates in ShopMongoRepository

Nell'elaborato vengono utilizzati i metodi Updates.push, Updates.pull, Filters.and e Filters.elemMatch per modificare ed eseguire queries sui documenti.

- **Updates.push("fieldName", value)**: viene utilizzato insieme al metodo updateOne(filter, updateFilter) per aggiungere uno o più elementi ad un campo array all'interno di un documento.
- **Updates.pull("fieldName", value)**: viene utilizzato insieme al metodo updateOne(filter, updateFilter) per rimuovere uno o più elementi ad un campo array all'interno di un documento.
- **Filters.and(filter, filter, ...)**: è un metodo utilizzato per combinare più filtri con un operatore logico AND, quindi il documento dovrà soddisfare tutte le condizioni elencate per essere selezionato.
- **Filters.elemMatch("fieldName", filter)**: metodo utilizzato per specificare un criterio di selezione su documenti che contengono elementi in un array.

4.2.3 Utilizzo di Optional in ShopMongoRepository

Optional è una classe introdotta in Java 8 per la gestione di valori assenti o nulli in maniera robusta. Un Optional può "rivestire" l'istanza di una classe e fornisce un modo per esprimere se il valore è presente o no, eliminando il bisogno di avere un null check esplicito. Nell'elaborato utilizzo il metodo isPresent() per controllare se il valore è stato trovato e, in caso affermativo, il metodo get() per recuperarlo.

4.3 ShopSwingView

L'implementazione della GUI, in figura 1, ha le seguenti caratteristiche:

- Il bottone "Add to Cart" dovrà essere abilitato solo quando è selezionato un prodotto nel negozio. Quando viene premuto deve chiamare il

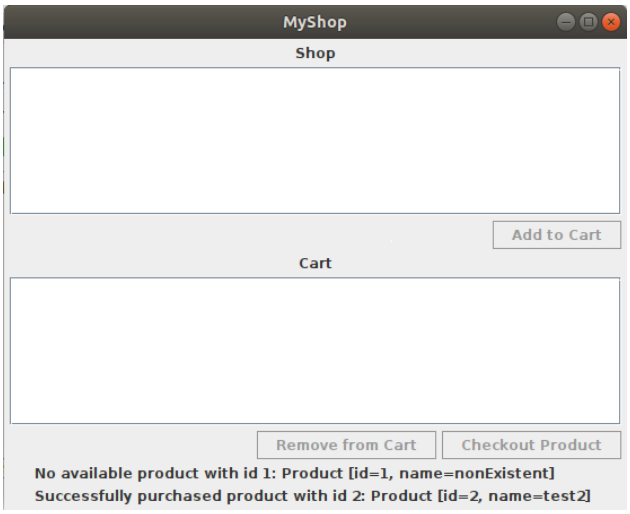

Figure 1: GUI dell'applicazione

metodo *addProductToCart* del Controller passando l'id del carrello e l'oggetto Product selezionato.

- Il bottone "Remove from Cart" dovrà essere abilitato solamente quando è selezionato un prodotto nel carrello. Quando viene premuto deve chiamare il metodo *removeProductFromCart* del Controller passando l'id del carrello e l'oggetto Product selezionato.
- Il bottone "Checkout Product" dovrà essere abilitato solamente quando è selezionato un prodotto nel carrello. Quando viene premuto deve chiamare il metodo *checkoutProductFromCart* del Controller passando l'id del carrello e l'oggetto Product selezionato.
- Due liste, una per indicare i prodotti presenti nel negozio e una per i prodotti presenti nel carrello.
- I metodi *productAddedToCart* e *productRemovedFromCart* spostano rispettivamente un prodotto dal negozio al carrello e viceversa. Inoltre devono anche resettare le labels di successo e errore.
- il metodo *showErrorProductNotFound* deve mostrare il messagio di errore e il nome del prodotto nell'apposito campo, inoltre rimuove il prodotto dalla lista e resetta la label di successo.
- il metodo *showPurchaseSuccessMessage* deve mostrare il messaggio di avvenuto acquisto del prodotto dal carrello.
- il metodo *productPurchased* deve rimuovere il prodotto dal carrello e resettare il campo errore.

Dalla prospettiva dell'utente la GUI avrà i seguenti comportamenti

- Il bottone "Add to Cart" verrà utilizzato per spostare il prodotto selezionato dall'area Shop all'area Cart.
- Il bottone "Remove from Cart" verrà utilizzato per spostare il prodotto selezionato dall'area Cart all'area Shop.
- Il bottone "Checkout Product" verrà utilizzato per cancellare il prodotto selezionato dal database. Il prodotto sparirà dalla lista.
- Errori o messaggi di avvenuto acquisto saranno mostrati in basso.

4.4 UI Unit tests - ShopSwingViewTest

Questi Unit tests mirano a una corretta implementazione della user interface ShopSwingView con l'ausilio di **AssertJ Swing**. Il Controller in questo caso sarà un mock in quanto vengono trattati Unit Tests e si vuole testare la View in isolamento. Nell'elaborato procedo in maniera similare a quella vista a lezione ma facendo attenzione a testare e implementare solo i metodi che permettono la realizzazione dello scenario in questione:

- Tests per i controlli della GUI:

  Lo stato iniziale degli elementi della lista di prodotti nello Shop.

  Lo stato iniziale del bottone "Add to Cart".

  Lo stato iniziale degli elementi della lista di prodotti nel Cart.

La condizione che abilita il bottone "Add to Cart".

- Tests per l'implementazione dell'interfaccia:

  I metodi *showAllProducts* e *showCart* devono mostrare il contenuto del negozio e del carrello.

  Il metodo *productAddedToCart* deve spostare un prodotto dal negozio al carrello.

- Tests per la la logica:

  il bottone "Add to Cart" deve chiamare il metodo *addProductToCart* del Controller.

4.5 Utilizzo di USER_CART_ID per View e Controller

Nell'elaborato viene fatto uso di un "Magic number" per l'id del carrello, estratto in una costante. In una reale applicazione di e-commerce ci si aspetta che un utente abbia associato un solo carrello al suo account e che sia determinato al momento della sua iscrizione o comunque fissato per la sessione. Non avendo ovviamente implementato un sistema di identificazione, autenticazione e autorizzazione che salva i dati utente su database, ho scelto di utilizzare sempre l'id "10" nel testing dell'applicazione.

5 Integration Tests

5.1 ShopMongoRepository-TestcontainersIT

Gli integration tests per la Repository sono effettuati con l'utilizzo di un vero server MongoDB ottenuto grazie a Testcontainers e Docker. I tests non duplicano tutti gli unit tests precedentemente effettuati ma si concentrano solo sui casi positivi.

5.2 ShopControllerIT

Gli integration tests che riguardano il Controller sono realizzati effettuando un mock della ShopView in quanto non ancora implementata e con un l'utilizzo un vero server MongoDB ottenuto grazie a Testcontainers e Docker. Come detto per la Repository anche in questo caso si testano solo i casi positivi.

5.3 ShopViewSwingIT

Gli integration tests per la GUI useranno ShopSwingView con uno ShopController reale e una ShopMongoRepository reale. Come database viene sempre utilizzata un vero server MongoDB con Testcontainers e Docker. In questi tests si verifica che la GUI (il SUT) si comporti correttamente rispettando la logica del Controller reale. Quindi nel nostro scenario d'esempio vogliamo verificare che:

1. I valori dei prodotti nel database siano correttamente mostrati nella lista "Shop"
2. Dopo aver selezionato un prodotto si clicca il pulsante "Add to Cart"
3. I valori del prodotto selezionato sono spostati dalla lista "Shop" alla lista "Cart"

Non sono stati scritti tests che hanno a che fare solo con la View poichè sono già stati testati negli unit tests in precedenza.

5.4 ModelViewControllerIT

Questi tests mirano a verificare che le azioni effettuate sulla View portino a modifiche allo stato del database. Nello scenario d'esempio vogliamo quindi verificare che il bottone "Add to Cart" vada effettivamente a spostare il prodotto dalla "productCollection" alla "ProductList" del carrello dell'utente. I tests usano sempre un vero server MongoDB con Testcontainers e Docker.

6 End-to-end tests

Questi tests considerano l'applicazione nella sua interezza, verificando che tutte le componenti interagiscano correttamente. Inoltre utilizzano la GUI dell'applicazione senza mai chiamare direttamente i metodi delle classi. Per creare gli scenari per i nostri tests avremo bisogno di un accesso diretto al

database, pertanto utilizzeremo un'istanza di MongoClient. Nel setup si parte sempre con un database vuoto e poi vengono aggiunti gli elementi necessari come supporto per i tests: due prodotti nel negozio e un carrello con due ulteriori prodotti. Per la verifica dello scenario d'esempio ci basta verificare che gli elementi del database siano correttamente mostrati al lancio dell'applicazione e che quando viene selezionato un prodotto e premuto il bottone "Add to Cart", lo stesso prodotto sia visibile nel carrello.

7 Fine del primo ciclo di sviluppo BDD

Completati gli End-to-end tests si completa anche il ciclo di sviluppo dello scenario d'esempio. Quindi si procede ad implementare alla stessa maniera i successivi scenari:

- **Scenario**: Add a not existing product to cart

  Given The user selects a product from the shop list

  But The product is in the meantime removed from the database

  When The user clicks the "Add to Cart" button

  Then An error is shown containing the name of the selected product

  And The product is removed from the list

- **Scenario**: Remove a product from cart

  Given The user selects a product from the cart list

  When The user clicks the "Remove from Cart" button

  Then The shop list contains the product

- **Scenario**: Checkout a product from cart

  Given The user selects a product from the cart list

  When The user clicks the "Checkout Product" button

  Then The product is removed from the cart list

  And A successful purchase message is shown containing the name of the selected product

- **Scenario**: Checkout a not existing product from cart

  Given The user selects a product from the cart list

  But The product is in the meantime removed from the cart in the database

  When The user clicks the "Checkout Product" button

  Then An error is shown containing the name of the selected product in the cart

  And The product is removed from the cart list

8 Code Coverage in JaCoCo e SonarCloud

Seguendo correttamente la metodologia TDD è stato semplice raggiungere il 100% di code coverage sia su JaCoCo che su SonarCloud. Per questo scopo sono state escluse le classi Product e Cart, il domain model per la nostra applicazione che non contiene logica da testare, e la classe con il metodo main, in quanto non ha senso fare code coverage su quest'ultima.

9 Mutation Testing

Il 100% del code coverage però non garantisce la correttezza del codice, per questo motivo è necessario utilizzare PIT, un framework per il mutation testing. Nell'elaborato il mutation testing è concentrato solo sulle parti che contengono la logica dell'applicazione, ovvero il Controller, che ne contiene la maggior parte e la Repository, che ne contiene una parte. PIT fornisce un Maven plugin, configurato per specificare le classi da mutare: il mutation test è quindi concentrato sugli unit tests di Controller e Repository abil-

itando gli “Stronger Mutators” in quanto “All Mutators” genera falsi positivi o utilizza mutanti sperimentali che possono contere bugs. Il profilo con id “mutation-testing” è eseguito periodicamente e nel Continous Integration server, quando viene effettuata una Pull Request.

10 Falsi positivi nell’analisi della qualità del codice

- ShopSwingView ha molti code smells che riguardano i nomi delle variabili locali, siccome questi nomi sono stati generati da WindowBuilder decidiamo di ignorare questi avvertimenti.
- ShopSwingViewTest ha molti code smells che riguardano la mancanza di assertion nei tests. In realtà questi test hanno delle assertion in quanto verificano la presenza dei controlli usando i metodi di AssertJ Swing, è SonarCloud a non rilevarli. Per questo motivo decidiamo di ignorare questi avvertimenti.
- ShopSwingAppE2E ha un code smell che riguarda il suo nome, questo perchè SonarCloud non riconosce E2E come un nome convenzionale per un test. Siccome E2E è un tag chiaro decidiamo di ignorare questo avvertimento.