JUnit

JUnit è essenzialmente un framework che ti permette di scrivere e automatizzare i test per il tuo codice Java. Pensa a JUnit come a un assistente che esegue controlli di qualità sul tuo software in modo sistematico e ripetibile.

Quando scrivi un programma, normalmente dovresti testarlo manualmente ogni volta che fai una modifica: esegui il codice, inserisci dei dati, controlli che il risultato sia corretto. Questo processo diventa rapidamente noioso e soggetto a errori, specialmente quando il progetto cresce. JUnit risolve questo problema permettendoti di scrivere dei test automatici che verificano se il tuo codice funziona correttamente.

In pratica, scrivi dei metodi speciali - contrassegnati dall'annotazione `@Test` che rappresentano dei casi di prova. Ogni metodo di test crea una situazione specifica (nel tuo caso, un biglietto con certi parametri), esegue il codice da testare (calcola il prezzo) e verifica che il risultato sia quello atteso (controlla che il prezzo sia corretto). Se tutti i test passano, hai una ragionevole sicurezza che il tuo codice funzioni; se qualche test fallisce, JUnit ti indica esattamente quale test ha avuto problemi, aiutandoti a localizzare rapidamente il bug.

Il grande vantaggio è che puoi eseguire tutti questi test con un singolo comando, in pochi secondi, ogni volta che modifichi il codice. Questo ti dà la tranquillità di sapere che le tue modifiche non hanno rotto funzionalità esistenti. È come avere una rete di sicurezza che ti protegge dagli errori mentre sviluppi, e diventa particolarmente prezioso nei progetti grandi dove è impossibile ricordare manualmente tutti i casi da testare.

I metodi di asserzione in JUnit sono strumenti fondamentali per verificare che il codice si comporti come ci aspettiamo. Pensa a loro come a delle "sentinelle" che controllano se ciò che il tuo programma produce corrisponde a quello che ti aspetti.

assertTrue - assertFalse

assertTrue e assertFalse sono probabilmente i più semplici da capire. Con assertTrue stai dicendo "mi aspetto che questa condizione sia vera", mentre con assertFalse dici l'opposto. Nel tuo codice, per esempio, usi `assertTrue(price == 20.0)` per verificare che il prezzo calcolato sia esattamente 20.0. Se la condizione risulta vera, il test passa senza problemi; se è falsa, il test fallisce e JUnit ti segnala l'errore.

assertEquals

assertEquals è più specifico e generalmente preferibile quando confronti valori. Invece di scrivere `assertTrue(price == 20.0)`, potresti scrivere `assertEquals(20.0, price)`. La differenza sostanziale è che assertEquals confronta direttamente due valori e, in caso di fallimento, ti mostra esattamente quali erano i due valori che non corrispondevano. Questo rende il debugging molto più semplice: immagina di avere un errore nel calcolo del prezzo - con assertEquals vedresti immediatamente "mi aspettavo 20.0 ma ho ottenuto 25.0", mentre con assertTrue vedresti solo "condizione falsa", costringendoti a investigare ulteriormente.

Un altro vantaggio di assertEquals è che gestisce meglio i confronti tra tipi diversi, specialmente per i numeri in virgola mobile dove piccole differenze di precisione possono causare problemi. Per i double, esiste anche una versione di assertEquals che accetta un valore di tolleranza, permettendoti di dire "questi due numeri devono essere uguali a meno di 0.001", cosa molto utile quando lavori con calcoli matematici che potrebbero introdurre piccoli errori di arrotondamento.

Nel contesto del tuo codice sui biglietti, sarebbe più robusto riscrivere i test usando assertEquals invece di assertTrue, proprio per avere messaggi di errore più chiari quando qualcosa non funziona come dovrebbe.

I metodi più utili di JUnit in modo schematico:

Confronto valori:

- `assertEquals(atteso, ottenuto)` - verifica che due valori siano uguali
- `assertNotEquals(atteso, ottenuto)` - verifica che due valori siano diversi
- `assertTrue(condizione)` - verifica che la condizione sia vera
- `assertFalse(condizione)` - verifica che la condizione sia falsa

Controllo oggetti nulli:

- `assertNull(oggetto)` - verifica che l'oggetto sia null
- `assertNotNull(oggetto)` - verifica che l'oggetto non sia null

Gestione eccezioni:

- `assertThrows(TipoEccezione.class, () -> codice)` - verifica che il codice lanci un'eccezione

Confronto collezioni:

- `assertArrayEquals(arrayAtteso, arrayOttenuto)` - confronta due array

Asserzioni multiple:

- `assertAll(() -> assert1, () -> assert2, ...)` - esegue tutte le asserzioni anche se alcune falliscono

Performance:

- `assertTimeout(durata, () -> codice)` - verifica che il codice termini entro un tempo limite

Annotazioni utili:

- `@BeforeEach` - esegue codice prima di ogni test
- `@AfterEach` - esegue codice dopo ogni test
- `@BeforeAll` - esegue codice una volta all'inizio di tutti i test
- `@AfterAll` - esegue codice una volta alla fine di tutti i test