**Proiect Testarea Sistemelor Software**

Proiectul a fost implementat sa simuleze comportamentul unei aplicatii de ridesharing, pe scurt sistemul ofera userilor functionalitati cum ar fi: creearea unor noi curse, rezervarea unui loc in masina, un sistem de recenzii pentru soferi si posibilitatea de a actualize si a vedea in timp real starea curenta a unei curse.

Arhitectura aplicatiei este bazata pe Spring Boot, cu o structura tipica pe layere (controller, service, repository) si cu stocarea datelor in MongoDB.

Obiectivul acestui proiect este testarea sistematica a unei aplicatii de ridesharing, implementate cu Spring Boot, folosind framework-ul JUnit, in combinatie cu instrumente precum Mockito, Spring Test si Testcontainers. Validarea codului existent si testarea completa a aplicatiei reprezinta scopul final al acestui proiect ce va include:

* testarea unitara a logicii de business (ex: validarea recenziilor, reguli de creare cursa),
* testarea de integrare pentru fluxuri reale (ex: endpoint-uri REST, interactiunea cu baza de date MongoDB),
* simularea componentelor externe intr-un mediu controlat (cu ajutorul mock-urilor sau containerelor de test).

Testarea devine esentiala in contextual gestionarii fluxurilor complexe de date din mai multe motive:

* Siguranta datelor – sistemul trebuie sa asigure ca doar utilizatorii validati pot interactiona (ex: doar pasagerii pot lasa recenzii).
* Corectitudinea logicii aplicatiei – rezervarile, recenziile si statusul curselor trebuie sa respecte reguli clare.
* Prevenirea regresiilor – modificarile ulterioare in cod nu trebuie sa afecteze functionalitatile existente.
* Testabilitate in mediu real – folosirea MongoDB intr-un container real permite testarea intr-un mediu apropiat de productie.

Definitii esentiale:

* Testare unitara: Verificarea functionalitatii unei unitati individuale de cod
* Testare de integrare: Testarea modului in care mai multe componente colaboreaza
* Mocking: Tehnica de simulare a unor componente externe pentru a testa doar o bucata de logica
* Coverage: Procentul de cod care este atins de teste
* CI/CD: Practica de integrare continua, care ruleaza testele automat la fiecare commit

Framework-uri folosite in proiect:

* JUnit (testare)
* Mockito (mocking)
* Spring Boot (backend)
* Testcontainers (testare cu MongoDB real in Docker)

Framework-ul JUnit in contextul aplicatiei Spring Boot:

Avantaje:

* Se integreaza nativ in Spring Boot
* Usor de rulat automat (CI/CD)
* Suporta testare structurata si modulara

Dezavantaje:

* Nu acopera testare UI sau performanta
* Pentru mocking avansat, necesita Mockito sau alte librarii

Framework-ul Mockito in contextul aplicatiei Spring Boot:

Avantaje:

* Se pot testa bucati de cod fara interventia unor componente externe
* Integrare buna cu JUnit si SpringTest
* Un framework superior pentru testele unitare

Dezavantaje:

* Folosirea excesiva a mocking poate pierde din vedere unele comportamente individuale
* Codul nu mai este usor de vizualizat atunci cand folosesti multe mockuri

Framework-ul SpringBootTest in contextul aplicatiei Spring Boot:

Avantaje:

* Se testeaza integral de la API la baza de date
* Unele adnotari ofera testare granulara

Dezavantaje:

* Testarea cu SpringBootTest dureaza o perioada semnificativ mai mare decat testarile unitare
* Poate ingreuna pipeline-urile de build

Framework-ul TestContainers in contextul aplicatiei Spring Boot:

Avantaje:

* Testarea se ruleaza pe o baza de date reala
* Fiecare test porneste un container nou fara a afecta alte teste

Dezavantaje:

* Dureaza multa vreme sa porneasca containerele docker
* Testele sunt greu de diagnosticat

Servicii externe:

* MongoDB (containerizat)
* GitHub Codespaces (dev environment)
* Docker (mediu de test izolat)

Tool-uri suport:

* Gradle (build system)
* Postman (pentru testare API manuala, optional)

Aplicatia se ruleaza cu:

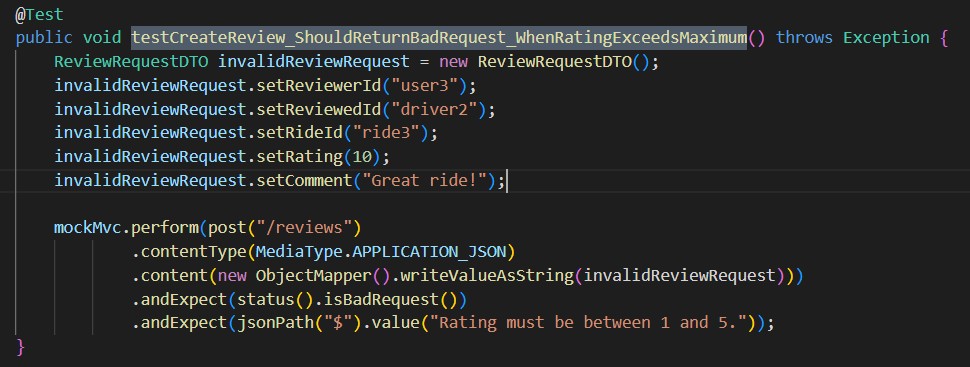
* ./gradlew build && ./gradlew bootRun (local)
* make build && ./start.sh (in Docker)
* MongoDB ruleaza local prin Testcontainers, cu setup automat in testele de integrare
* Testele sunt scrise cu JUnit, organizate pe module:
* ReviewServiceTest.java – teste unitare cu Mockito
* ReviewControllerIntegrationTest.java – test de integrare cu context Spring si DB real

# Functional Testing:

## Partitionarea in clase de echivalenta:

Aceasta tehnica presupune impartirea valorilor de intrare in clase logice echivalente: o valoare din fiecare clasa este suficienta pentru a valida comportamentul.

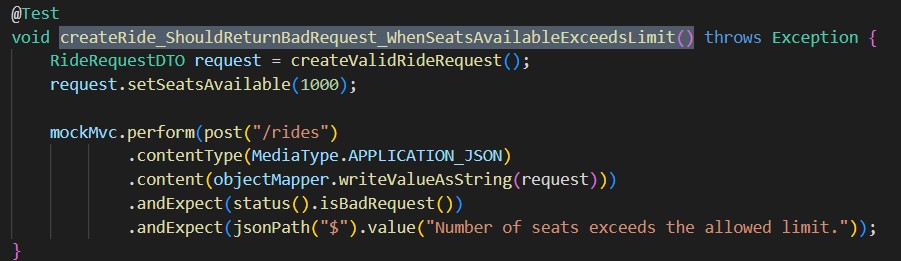
Exemplu 1: rating – valid si invalid:



Explicatie: Se testeaza clasa invalida in care ratingul depaseste valoarea maxima admisa (5). Validarea este deja implementata in serviciu si returneaza mesajul corespunzator.

Exemplu 2: seatsAvailable – clase valida, sub si peste limita



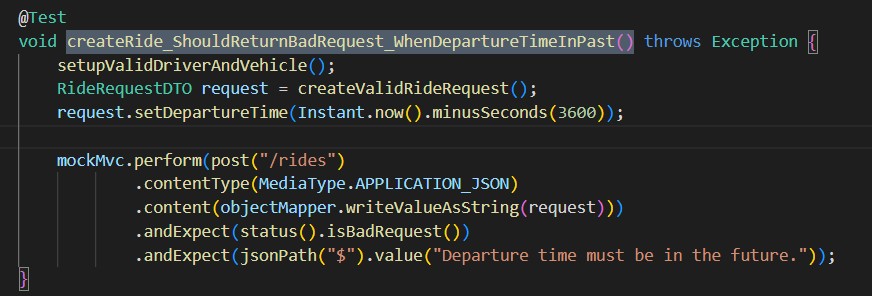


Explicatie: Testele acopera clasele invalide de input pentru numarul de locuri: sub minim (0) si peste maxim (1000). Este suficient un caz pentru fiecare, pentru ca orice alta valoare in acea clasa ar duce la acelasi rezultat.

Analiza valorilor de frontiera:

Aceasta tehnica testeaza exact valorile de **la limita** ale unui domeniu de intrare (minimul/maximul acceptabil + imediat sub/peste).

Exemplu: departureTime – valoare in trecut



**Explicatie**: Se testeaza limita inferioara a timpului de plecare – cu o valoare exact sub minimul acceptat . Comportamentul corect este respingerea cu eroare.

Exemplu: password – sub limita de lungime



**Explicatie:** Se testeaza cazul in care parola este mai scurta decat limita impusa (de ex. < 6 caractere). Asta e o valoare de **la limita inferioara** invalida.

Exemplu: rating – peste limita maxima admisa



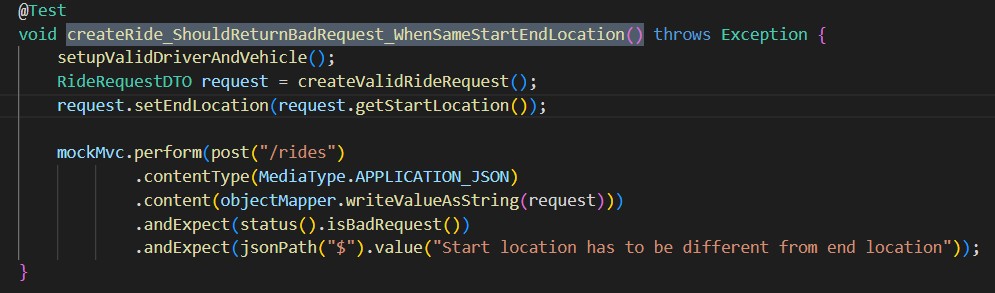
**Explicatie:**

* Se testeaza cazul in care valoarea ratingului trimis in review depaseste limita maxima permisa (5).
* Valoarea 10 este o **valoare de la limita superioara invalida,** adica max + 1, folosita pentru **analiza valorilor de frontiera (Boundary Value Analysis).**

Category Partitioning

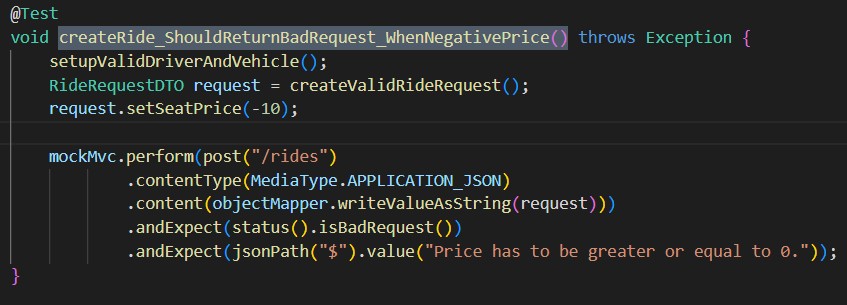
Aceasta abordare presupune clasificarea valorilor de intrare in categorii si combinarea acestor categorii in scenarii de test.

Exemplu: startLocation == endLocation



Explicatie: Aceasta este o categorie logica (start si end nu trebuie sa fie egale). In test se invalideaza doar aceasta regula, restul valorilor raman valide.

Exemplu: seatPrice < 0



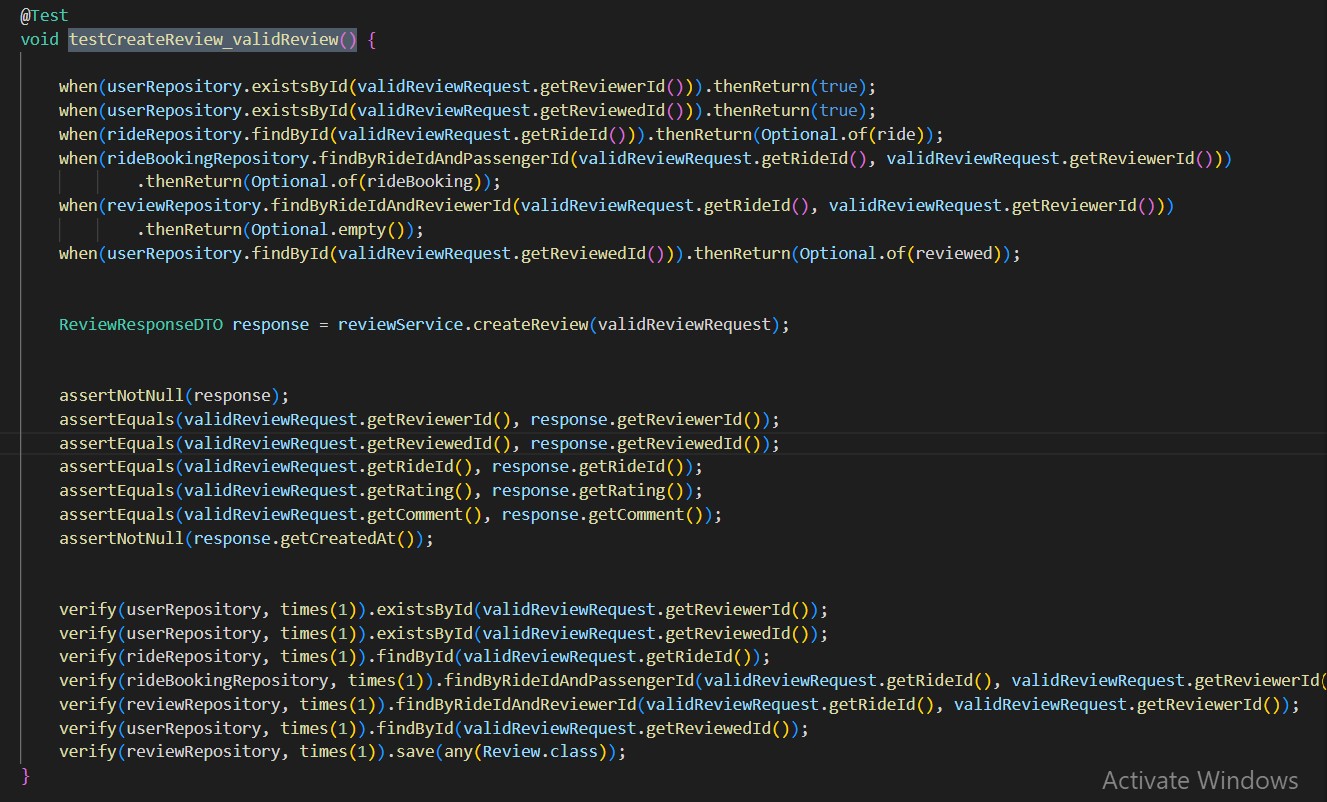
**Explicatie**: seatPrice este tratat ca o categorie numerica, iar testul acopera cazul in care acesta este invalid (negativ). Este singurul parametru invalid in test.

Structural Testing

**Statement Testing**

**Definitie:**Se asigura ca **fiecare instructiune** dintr-o metoda este executata **cel putin o data** de un test.

Exemplu: ReviewServiceTest.testCreateReview\_validReview()



**Explicatie:**

Acest test apeleaza calea completa de succes din metoda createReview(), **executand toate instructiunile:** validari de ID-uri, cautari in DB, salvarea review-ului, construirea obiectului DTO etc.

I**nstructiunile liniare** (if-uri fara else) sunt parcurse.

Acoperire: **Statement coverage 100%** pentru ramura „happy path”.

**Decision Testing**

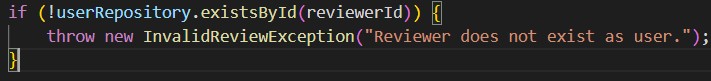
**Definitie:**Asigura ca **fiecare decizie (if, else)** are ambele ramuri (true si false) acoperite de cel putin un test.

Exemplu: testCreateReview\_reviewerDoesNotExist() + testCreateReview\_validReview()

****

Explicatie:

In createReview(), exista:



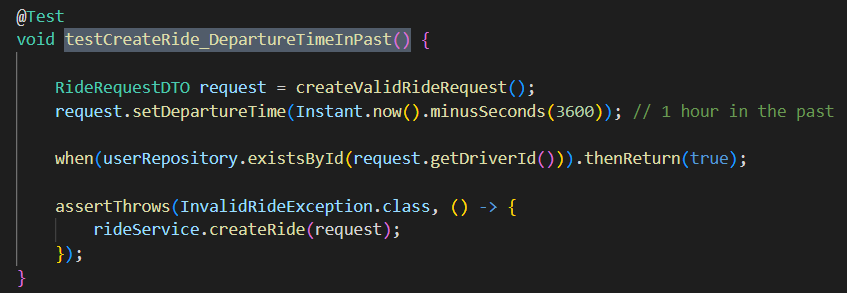
Acest test asigura ca ramura „false” a fost testata.

Combinat cu testul de succes (validReview), ambele ramuri ale deciziei sunt acoperite.

**Condition Testing**

**Definitie:**Testarea fiecarei **conditii simple** dintr-o expresie compusa. Chiar daca if (a && b) este testat ca „true”, trebuie sa verificam si cazuri cand a este true dar b false, si invers.

Exemplu aplicabil: RideService.createRide()

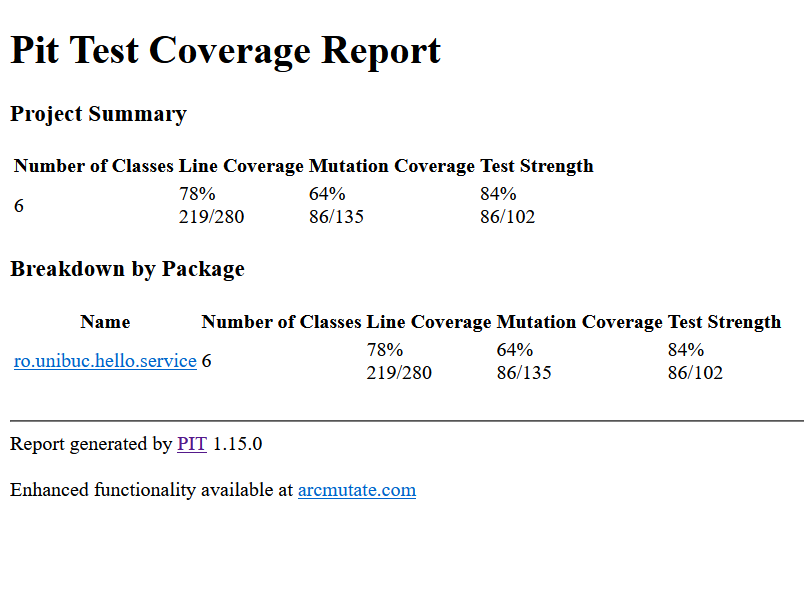


Explicatie:

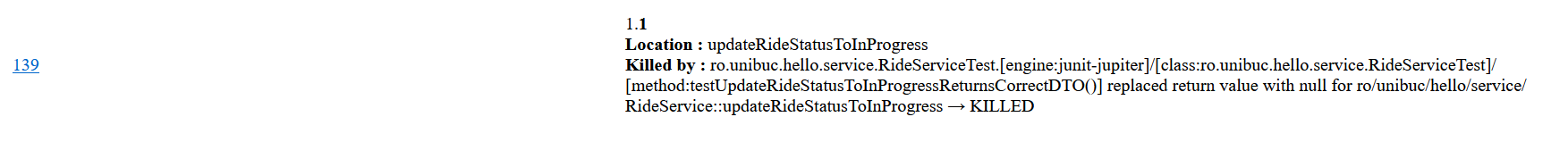
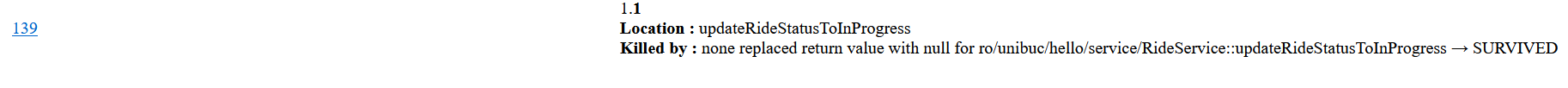
Aici avem un if (conditie) cu o singura conditie, deci testul ofera si condition coverage, deoarece:

* ride.getDepartureTime().isBefore(Instant.now()) == true → ramura testata
* Complementul este testat de scenariile valide (ex: testCreateRide\_Success), cand conditia este false.

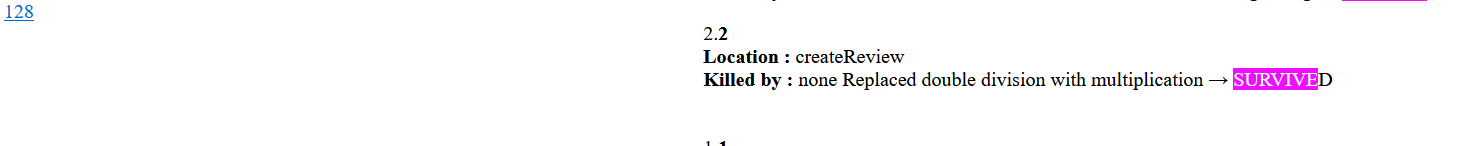
**Analiza raportului creat de generatorul de mutanti:**

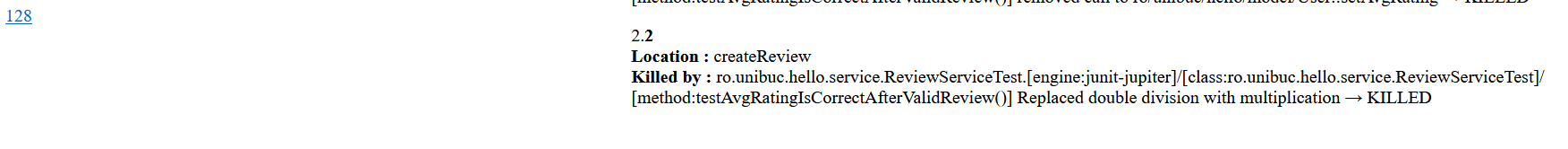


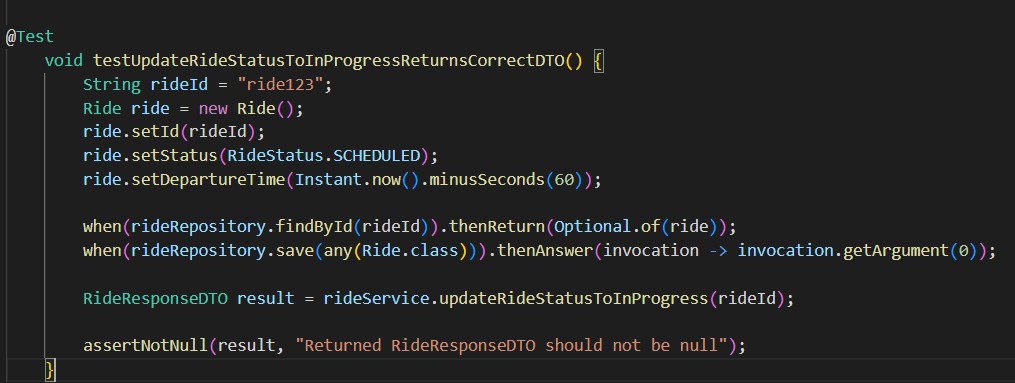
**Teste suplimentare pentru a omori doi dintre mutantii ramasi in viata**











Articole stiintifice si resurse relevante:

* https://www.vogella.com/tutorials/JUnit/article.html
* <https://katalon.com/resources-center/blog/integration-testing>
* <https://www.researchgate.net/publication/363363196_A_Review_on_the_Process_of_Automated_Software_Testing>
* <https://thatapicompany.com/a-comprehensive-guide-to-api-testing-strategies-and-tools/>
* <https://www.lotus-qa.com/blog/java-testing-frameworks/>