Testarea Sistemelor Software

Testare unitară în Java

Profesor coordonator:
Preduț Sorina Nicoleta

Realizat de:
Ciocan Alexandra-Diana
Georgescu Miruna-Bianca
Iliescu Gabriel Bogdan
Pintenaru-Dumitrescu Nicole Melissa

1. Noțiuni generale de testare unitară în Java

1.1. Testare unitară

Testarea unitară reprezintă o metodă de bază de testare a codului sursă. Acesta presupune verificarea individuală, izolată, a „unităților” de bază ale programului, cum ar fi clase de tip serviciu. Scopul principal în implementarea acestui tip de testare este a ne asigura că fiecare unitate funcționează conform așteptărilor.

Cu ajutorul testelor unitare, integrate în fluxul de dezvoltare continuă, dezvoltatorii de software pot detecta erorile înainte ca acestea să ajungă în producție și pot îmbunătăți calitatea codului. Corelat cu fluxul de dezvoltare continuă, acest tip de testare este esențial unui context de muncă Agile, contribuind la un ritm alert al procesului de dezvoltare și la reducerea costurilor asociate cu identificarea și rezolvarea defectelor.

În contextul Java, unul dintre cele mai populare framework-uri pentru testarea unitară este JUnit. Acesta oferă o serie de adnotări și metode de aserție care facilitează scrierea și execuția testelor. Împreună cu diverse plugins care pot fi adăugate peste IDE-ul ales, în cazul nostru IntelliJ, acesta este un tool puternic pentru analiza calității software-ului.

1.2. Sintaxa specifică JUnit

În cadrul proiectului am folosit JUnit 5, care, față de versiunile anterioare, conține module din trei sub-proiecte diferite: JUnit Platform, JUnit Jupiter, JUnit Vintage. JUnit platform este componenta care se ocupă de execuția testelor pe JVM, JUnit Jupiter oferă adnotările necesare scrierii testelor, iar JUnit Vintage conține un TestEngine pentru JUnit 3 și 4.

Condițiile de bază pentru scrierea unui test cu ajutorul este JUnit sunt: crearea unei clase de test, adnotarea metodei de test cu @Test și folosirea unei aserții în cadrul acestei metode.

Sintaxa spefcifică folosită în cadrul proiectului constă în următoarele:

- Adnotările specifice suitelor: am grupat testele într-o suită strategy-based
    - @Suite: marchează clasa ca trebuind executată ca o singură unitate
    - @SuiteDisplayName: oferă un titlu custom
    - @SelectClasses: specifică clasel de test care trebuie rulate în această unitate
- Adnotările specifice testelor:
    - @Test: marchează metoda ca test de executat
    - @BeforeEach: marchează o metodă ce vrem să fie executată înaintea executării fiecărui test, poate include un setup comun, initializarea clasei ce trebuie testată, etc…
- Aserții
    - assertEquals(expectedResult, actualResult): este folosit pentru a verifica că rezultatul întors este cel expected
    - assertThrows(exception, methodCall): folosit pentru tratarea excepțiilor

1.3. Convenții de numire

Suitele de teste pot funcționa ca o modalitate de documentare a codului, astfel încât este de ajutor ca numele testelor să fie cât mai sugestive. Documentația oficială sugerează următoarea strategie de numire:

given<tip_de_input>_when<metoda_apelata>_then<rezultat asteptat>

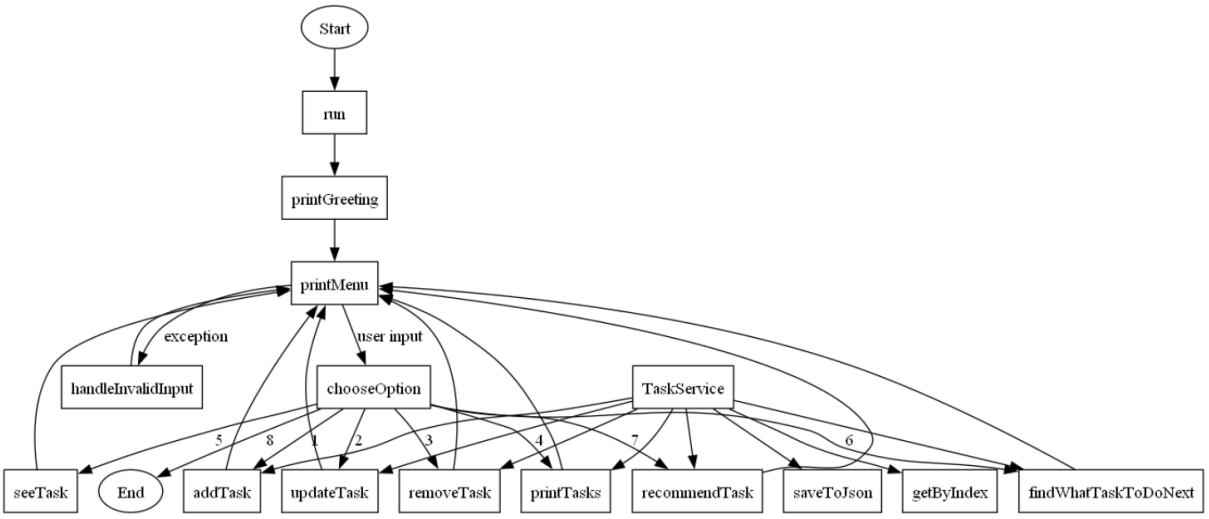
care urmează structura clasică a unui test: setup, acțiune, verificare rezultat.

2. Scurtă prezentare a aplicației

Aplicația este un To Do App cu funcționalități de bază, care permite crearea unei liste de taskuri de forma:

```
public class Task {
   private String description;
   private Priority priority;
   private Status status;
   private Integer hourEstimate;
}
```

De asemenea, aplicația pune la dispoziție o serie de operații asupra listei de taskuri, ilustrate în graful:



Graful de mai sus indică fluxul central al aplicației, de la start la alegerea unei opțiuni din meniu. Aplicația este structurată într-o clasă principală, care funcționează ca un meniu, ce generează în CLI o listă de opțiuni posibile. În funcție de opțiunea aleasă, este apelată metoda ce îi corespunde din clasa „Task Service”. Această clasă conține logica business a aplicației și, prin urmare, este clasa pe care o vom testa.

3. Testare funcțională

Testarea funcțională se concentrează pe generarea datelor de test pe baza cerințelor programului, specificațiile incluzând pre-condiții și post-condiții. În majoritatea metodelor funcționale, datele de intrare sunt împărțite în clase de echivalență.

Testarea funcțională implică mai multe strategii, dintre care vom trata: partiționarea în clase de echivalență și analiza valorilor de frontieră.

3.1. Partiționarea în clase de echivalență

Partiționarea în clase de echivalență presupune împărțirea datelor de intrare în clase de echivalență cu comportament similar și selectarea unui singur reprezentant al fiecărei clase pentru testare. Aceasta identifică în mod eficient scenariile de test și asigură acoperirea exhaustivă. De obicei, se ajută de analiza valorilor de frontieră pentru rezultate mai bune.

Ex 3.1.1. Metoda updateTask()

```
public void updateTask(int index, Task task) {
    if (index < 0 || index >= tasks.size()) {
        throw new IllegalArgumentException("Invalid task number");
    }
    tasks.set(index, task);
}
```

Metoda țintește să modifice taskul de la poziția index, astfel încât acesta să fie noul task trimis ca parametru.

Datele de intrare sunt:

- index, număr întreg
- task, obiect care se presupune ca este corect format dacă a trecut din meniul CLI în clasa TaskService

Variabila „index” ia valori întregi, care, d.p.d.v. funcțional, pot fi împărțite în 3 clase de echivalență:

I_1 = { i | i întreg, i < 0}
I_2 = { i | i întreg, i >= 0, i < numărul total de taskuri adăugate}
I_3 = { i | i întreg, i >= numărul total de taskuri adăugate}

Clasele de echivalență globale vor fi:

C_1 = { (i, t) | i întreg, i < 0, task - task corect format}
C_2 = { (i, t) | i întreg, i >= 0, i < numărul total de taskuri adăugate, task - task corect format}
C_3 = { i | i întreg, i >= numărul total de taskuri adăugate, task - task corect format}

Simulăm o stare a instanței taskService care conține o listă de 3 taskuri deja adăugate. Alegând câte un reprezentant pentru fiecare clasă, datele de test vor fi:
c_1: (0, Task("new task1", Priority.LOW, 4))
c_2: (-1, Task("new task1", Priority.LOW, 4))
c_3: (3, Task("new task1", Priority.LOW, 4))

Domeniul de ieșiri constă în următoarele 2 răspunsuri:

- modificarea taskului cu indexul „index”
- aruncarea unei excepții de tipul IllegalArgumentException

Legătura date de intrare - domeniu de ieșiri este dată de următorul tabel:

Date de intrare	Rezultat afișat (expected)
Index	
0	Modificarea taskului cu indexul 0
-1	Aruncă o excepție de tipul IllegalArgumentException
3	Aruncă o excepție de tipul IllegalArgumentException

Astfel, testele care acoperă integral clasele de echivalență sunt:

- givenValidIndex_whenUpdateTask_thenModifyTasks()

```
@Test
void givenValidIndex_whenUpdateTask_thenModifyTasks() {
   //prepare data
   taskService.addTask(new Task("task1", Priority.HIGH, 1));
   taskService.addTask(new Task("task2", Priority.HIGH, 2));
   taskService.addTask(new Task("task3", Priority.HIGH, 3));

   Task updatedTask = new Task("new task1", Priority.LOW, 4);

   //call method
   taskService.updateTask(0, updatedTask);

   //check result
   assertEquals(updatedTask, taskService.getByIndex(0));
}
```

Metoda testează modul în care un task valid este actualizat la apelul metodei

updateTask(). Mai întâi, se adaugă un set de date inițial în taskService. Se creează updatedTask, un task pe care îl vom folosi pentru a înlocui taskul cu indexul 0. Se apelează metoda updateTask() cu indexul 0 și updatedTask ca parametri. În final, verificăm dacă taskul cu indexul 0 a fost modificat corect.

- givenNegativeIndex_whenUpdateTask_thenThrowException()

```
@Test
void givenNegativeIndex_whenUpdateTask_thenThrowException() {
   //prepare data
   taskService.addTask(new Task("task1", Priority.HIGH, 1));
   taskService.addTask(new Task("task2", Priority.HIGH, 2));
   taskService.addTask(new Task("task3", Priority.HIGH, 3));

   Task updatedTask = new Task("new task1", Priority.LOW, 4);

   //call method
   assertThrows(IllegalArgumentException.class, () ->
taskService.updateTask(-1, updatedTask));
}
```

Metoda verifică dacă funcția updateTask() aruncă o excepție atunci când este apelată pe un index negativ. Se adaugă un set de date inițial, apoi se creează noul task, updatedTask, care conține datele cu care încercăm sa actualizam un task cu index negativ. Se apelează metoda updateTask cu indexul -1 și updatedTask ca parametrii și se verifică dacă returnează o excepție de tipul IllegalArgumentException.

- givenTooHighIndex_whenUpdateTask_thenThrowException()

```
@Test
void givenTooHighIndex_whenUpdateTask_thenThrowException() {
   //prepare data
   taskService.addTask(new Task("task1", Priority.HIGH, 1));
   taskService.addTask(new Task("task2", Priority.HIGH, 2));
   taskService.addTask(new Task("task3", Priority.HIGH, 3));

   Task updatedTask = new Task("new task1", Priority.LOW, 4);

   //call method
   assertThrows(IllegalArgumentException.class, () ->
taskService.updateTask(3, updatedTask));
}
```

Metoda verifică dacă funcția updateTask() aruncă excepție atunci când este apelată pe un index prea mare. Se adaugă un set inițial de date ce conține 3 taskuri. Se creează noul task, updatedTask, care conține datele cu care încercăm sa actualizam taskul cu indexul 3. Se apelează metoda updateTask cu indexul 3 și updatedTask ca parametrii și se verifică dacă returnează excepție de tip IllegalArgumentException.

Ex 3.1.2: metoda recommendTask()

```
public Task recommendTask(int priority, int timeEstimate) {
    if(priority < 1 || priority > 3) {
        throw new IllegalArgumentException("Invalid priority");
    }

    if(timeEstimate <= 0) {
        throw new IllegalArgumentException("Invalid time estimate");
    }

    Priority priorityEnum = Priority.fromLevel(priority);

    Task recommendedTask = null;
    List<Task> filteredTasks = tasks
                    .stream()
                    .filter(task -> task.getPriority() == priorityEnum &&
task.getTimeEstimate() <= timeEstimate && task.getStatus() !=
Status.COMPLETE)
                    .toList();

    for (Task task : filteredTasks) {
        if(recommendedTask == null || task.getTimeEstimate() <
recommendedTask.getTimeEstimate()) {
            recommendedTask = task;
        }
    }

    if (recommendedTask == null) {
        throw new NoSuchElementException("No task with required properties
found");
    }
    return recommendedTask;
}
```

Metoda recomandă un task de prioritate indicată, care se încadrează în limita de timp indicată, fiind cel mai rapid de acest tip.

Date de intrare:

- priority, număr întreg
- timeEstimate, număr întreg

Variabila „priority” ia valori întregi, care, d.p.d.v. funcțional, pot fi împărțite în 3 clase de echivalență:

P_1 = { p | p întreg, p < 0 }
P_2 = { p | p întreg, p >= 1, p <= 3, p există ca prioritate în taskurile deja adăugate }
P_3 = { p | p întreg, p >= 1, p <= 3, p nu există ca prioritate în taskurile deja adăugate }
P_4 = { p | p întreg, p > 3 }

Variabila „timeEstimate” ia valori întregi, care, d.p.d.v. funcțional, pot fi împărțite în 3 clase de echivalență:
T_1 = { t | t întreg, t <= 0 }
T_2 = { t | t întreg, t > 0, t <= cel mai mare estimate al unui task }
T_3 = { t | t întreg, t > 0, t > cel mai mare estimate al unui task }

Clasele de echivalență globale vor fi:
C_1 = { (p) | p în P_1 }
C_1' = { (t) | t în T_1 }
C_22 = { (p, t) | p,t întregi, p în P_2, T în T_2}
C_23 = { (p, t) | p,t întregi, p în P_2, T în T_3}
C_32 = { (p, t) | p,t întregi, p în P_3, T în T_2}
C_33 = { (p, t) | p,t întregi, p în P_3, T în T_3}
C_4 = { (p) | p întreg, p în P_4}

Simulăm o stare a instanței taskService care conține o listă de taskuri deja adăugate. Alegând câte un reprezentant pentru fiecare clasă, datele de test vor fi:
c_1 : (-1, 1)
c_1' : (1, -1)
c_22 : (3, 20)
c_23 : (1, 20)
c_32: (3, 1)
c_33 : (1, 1)
c_4 : (4,20)

Domeniul de ieșiri constă în următoarele 4 răspunsuri:
- aruncarea unei excepții de tipul IllegalArgumentException cu mesajul `Invalid priority`
- aruncarea unei excepții de tipul IllegalArgumentException cu mesajul `Invalid time estimate`
- aruncarea unei excepții de tipul NoSuchElementException
- un task cu prioritatea cerută și cel mai mic time estimate mai mic decat time estimate-ul cerut

Legătura date de intrare - domeniu de ieșiri este dată de următorul tabel:

Date de intrare		Rezultat afișat (expected)
Priority	Time estimate	
-1	1	Excepție "Invalid priority"
1	-1	Excepție "Invalid time estimate"

3	20	Task recomandat
1	20	Excepție "No task with required properties found"
3	1	Excepție "No task with required properties found"
1	1	Excepție "No task with required properties found"
4	20	Excepție "Invalid priority"

3.2. Analiza valorilor de frontieră

În cadrul analizei valorilor de frontieră se selectează valorile limită ale intervalului de intrate și se testează comportamentul pentru aceste valori, având în vedere că aceste valori cauzează erori de cele mai multe ori.

Se iau în considerare 4 zone de testare:

- limite inferioare
- valori la limita inferioară
- limite superioare
- valori la limita superioară

Ex 3.2.1. Metoda getByIndex()

```
public Task getByIndex(int index) {
    if (index < 0 || index >= tasks.size()) {
        throw new IllegalArgumentException("Invalid task number");
    }

    return tasks.get(index);
}
```

Observăm că partiționarea în clase de echivalență decurge exact ca la ex 1.1, clasele fiind:

C_1 = { (i) | i întreg, i < 0}

C_2 = { (i) | i întreg, i >= 0, i < numărul total de taskuri adăugate}

C_3 = { (i) | i întreg, i >= numărul total de taskuri adăugate}

Valorile de frontieră care decurg sunt: -2, -1, 0, 1, 2.

Domeniile de ieșire sunt:

- returnează taskul
- aruncă o excepție de tipul IllegalArgumentException

Tabelul care asociază date de intrare - domenii de ieșire este:

Date de intrare	Rezultat afișat (expected)
Index	
Limita inferioară	Returnează taskul
Sub limita inferioară	Aruncă o excepție de tipul IllegalArgumentException
Limita superioară	Returnează taskul
Peste limita superioară	Arunca o excepție de tipul IllegalArgumentException

- givenLowestValidIndex_whenGetByIndex_thenRetrieveTask()

```
@Test
public void givenLowestValidIndex_whenGetByIndex_thenRetrieveTask() {
   taskService.addTask(new Task("task1", Priority.HIGH, 1));

   assertEquals("task1", taskService.getByIndex(0).getDescription());
}
```

Metoda verifică comportamentul funcției getByIndex() atunci cand se furnizează cel mai mic index valid. Mai intai, se adaugă setul de date inițial, format de data aceasta dintr-un singur task. Apoi, se apelează metoda getByIndex() pentru indexul 0 și se verifică dacă descrierea returnată este cea corectă.

- givenHighestValidIndex_whenGetByIndex_thenRetrieveTask()

```
@Test
public void givenHighestValidIndex_whenGetByIndex_thenRetrieveTask() {
   taskService.addTask(new Task("task1", Priority.HIGH, 1));
   taskService.addTask(new Task("task2", Priority.HIGH, 2));
   taskService.addTask(new Task("task3", Priority.HIGH, 3));

   assertEquals("task3", taskService.getByIndex(2).getDescription());
}
```

- givenIndexAboveHighest_whenGetByIndex_thenThrowException()

```
@Test
public void givenIndexAboveHighest_whenGetByIndex_thenThrowException() {
```

```
    assertThrows(IllegalArgumentException.class, () ->
taskService.getByIndex(1));
}
```

Metoda verifică dacă este aruncată o excepție atunci când se apelează metoda getByIndex() pentru un index mai mare decât cel mai mare index valid. Așadar, se apelează funcția getByIndex() pentru indicele 1 și se verifică dacă este aruncată o excepție de tipul IllegalArgumentException.

- givenIndexBelowLowest_whenGetByIndex_thenThrowException()

```
@Test
public void givenIndexBelowLowest_whenGetByIndex_thenThrowException() {
    assertThrows(IllegalArgumentException.class, () ->
taskService.getByIndex(-1));
    assertThrows(IllegalArgumentException.class, () ->
taskService.getByIndex(-2));
}
```

Metoda verifică dacă este aruncată o excepție atunci când se apelează metoda getByIndex() pentru un index mai mic decât cel mai mic index valid, adică un index negativ. Prin urmare, se apelează funcția getByIndex() pentru indicii -1 și -2 și se verifică dacă este aruncată o excepție de tipul IllegalArgumentException.

4. Testare structurală

Testarea structurală implică o metodologie complet diferită față de cea funcțională. Nu ne interesează ce ar trebui sa facă aplicația, ci luăm ca punct de plecare structura internă a acesteia. Aceasta presupune reprezentarea programului sub forma unui graf de flux de control(CFG) și implementarea testelor, atât ca scenariu, cât și ca date de intrare, pe baza acestuia. Scopul urmărit este acoperirea maximă a grafului și identificarea unor posibile probleme de logică.

4.1. CFG

Metoda pe baza căreia am modelat graful este findWhatTaskToDoNext, o metodă care primește ca parametru numărul de taskuri și identifică cel mai important task disponibil în funcție de priorități și de timpul estimat pentru completarea sa.

Mai întâi, se verifică dacă numărul de sarcini specificat este egal cu dimensiunea listei de taskuri și dacă lista nu e goală. Dacă cele două condiții nu sunt îndeplinite, metoda aruncă o excepție pentru a indica o problemă cu parametrii de intrare.

Apoi, metoda parcurge lista de sarcini și selectează cea mai importantă sarcină neterminată, luând în considerare atât prioritățile, cât și estimările de timp pentru fiecare în parte.

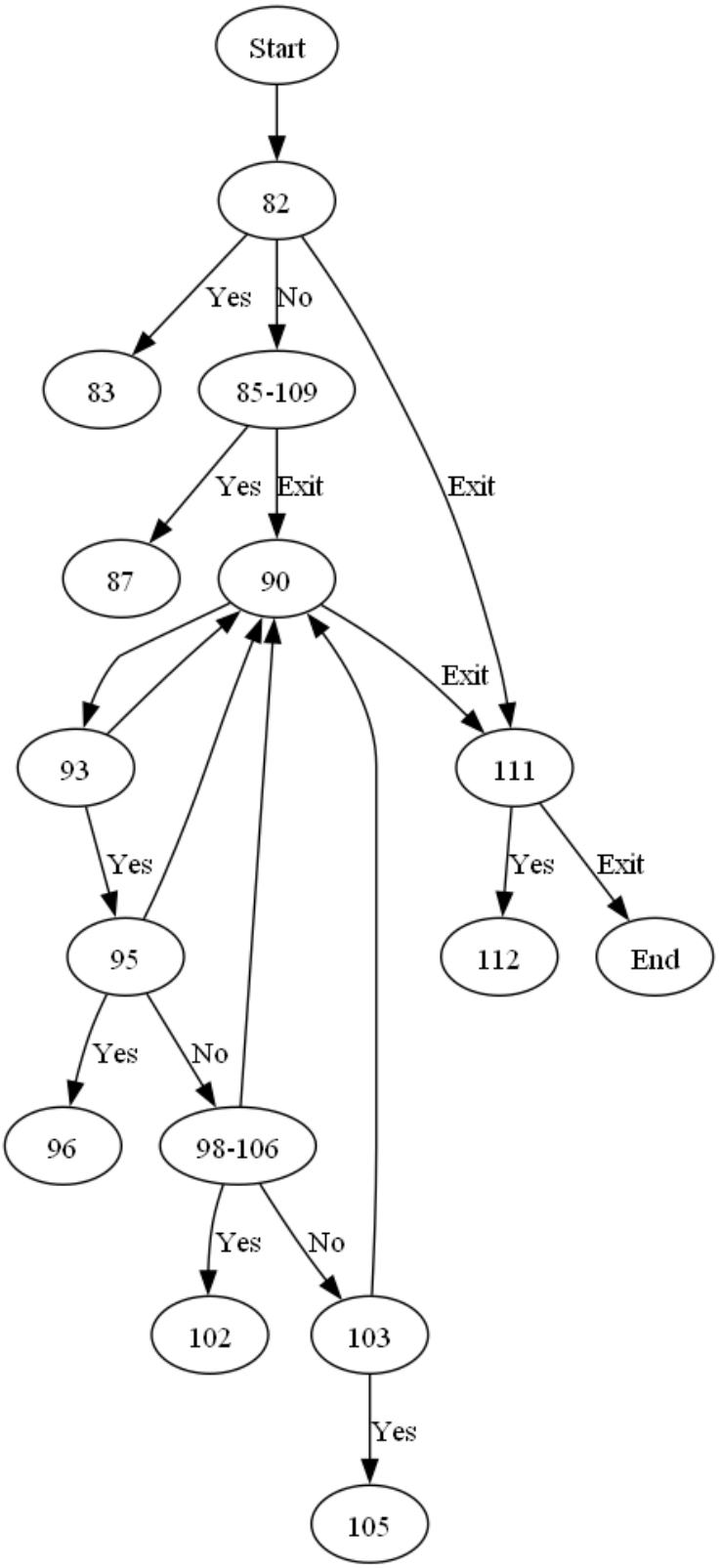
Dacă nu este găsit niciun task disponibil, se aruncă o excepție pentru a indica acest lucru.

```java
public Task findWhatTaskToDoNext(int numberOfTasks) {
    Task highestPriorityShortestEstimateTask = null;

    //numberoftasks must be the same as the size of the lists
    if (numberOfTasks != tasks.size()) {
        throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
    } else {
        // numberOfTasks must be between 1 and 20
        if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
            throw new IllegalArgumentException("Invalid number of tasks or empty list");
        }

        for (int i = 0; i < numberOfTasks; i++) {
            Task task = tasks.get(i);
            // make sure it's not completed
            if (task.getStatus() != Status.COMPLETE) {
                // if no prior task was selected, select task
                if (highestPriorityShortestEstimateTask == null) {
                    highestPriorityShortestEstimateTask = task;
                } else {
                    // check to see if the task is of higher priority and if tasks have the same priority check
                    // time estimate
                    if (task.getPriority().getLevel() >
                            highestPriorityShortestEstimateTask.getPriority().getLevel()) {
                        highestPriorityShortestEstimateTask = task;
                    } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
                            task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
                        highestPriorityShortestEstimateTask = task;
                    }
                }
            }
        }
```

```java
        if (highestPriorityShortestEstimateTask == null) {
            throw new IllegalStateException("No available tasks to do");
        }

    }
    return highestPriorityShortestEstimateTask;
}
```

Pe baza acestei metode am realizat următorul CFG, în care fiecare instrucțiune este reprezentată de un nod al grafului, iar muchiile reprezintă fluxul dat de valorile de adevăr

atribuite expresiilor din structurile decizionale:



Testele la care am ajuns pe baza diagramei sunt:

```
package tests;

import model.Priority;
```

```java
import model.Status;
import model.Task;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import services.TaskService;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class FindWhatTaskToDoNextTestCoverageTests {
    TaskService taskService;

    @BeforeEach
    public void setUp() {
        taskService = new TaskService();
    }

    @Test
    void
givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestP
riorityTask() {
        // adding two tasks with different priorities to the list of tasks
        taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
        taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));

        Task returnedTask = taskService.findWhatTaskToDoNext(2);
        Task expectedTask = taskService.getByIndex(0);

        assertEquals(expectedTask, returnedTask);
    }

    @Test
    void
givenTasksWithSamePriorityAndShorterTime_whenFindWhatTaskToDoNext_thenSelectS
horterTimeTask() {
        // adding two tasks with different priorities to the list of tasks
        taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
        taskService.addTask(new Task("Task 2", Priority.HIGH, 2));

        Task returnedTask = taskService.findWhatTaskToDoNext(2);
        Task expectedTask = taskService.getByIndex(1);

        assertEquals(expectedTask, returnedTask);
    }

    @Test
    void givenNoAvailableTasks_whenFindWhatTaskToDoNext_thenThrowException() {
        // set two tasks
        Task task1 = new Task("Task 1", Priority.HIGH, 3);
        Task task2 = new Task("Task 2", Priority.MEDIUM, 5);

        // set them as complete
        task1.setStatus(Status.COMPLETE);
        task2.setStatus(Status.COMPLETE);

        // add tasks to the list
```

```
        taskService.addTask(task1);
        taskService.addTask(task2);

        // check if method throws IllegalStateException: "No available tasks to
do"
        assertThrows(IllegalStateException.class, () -> {
            taskService.findWhatTaskToDoNext(2);
        }, "No available tasks to do");
    }

    @Test
    void
givenTasksWithDifferentPrioritiesAndTimeEstimates_whenFindWhatTaskToDoNext_th
enSelectHighestPriorityAndShortestTimeTask() {
        // add tasks to list
        taskService.addTask(new Task("Task 1", Priority.LOW, 3));
        taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));
        taskService.addTask(new Task("Task 3", Priority.HIGH, 1));

        Task returnedTask = taskService.findWhatTaskToDoNext(3);
        Task expectedTask = taskService.getByIndex(2);

        assertEquals(expectedTask, returnedTask);
    }

    @Test
    void givenEmptyTaskList_whenFindWhatTaskToDoNext_thenThrowException() {
        assertThrows(IllegalArgumentException.class, () ->
taskService.findWhatTaskToDoNext(0));
    }

    @Test
    void givenWrongNoOfTasks_whenFindWhatTaskToDoNext_thenThrowException() {
        assertThrows(IllegalArgumentException.class, () ->
taskService.findWhatTaskToDoNext(3));
    }

    @Test
    void givenTooHighNoOfTasks_whenFindWhatTaskToDoNext_thenThrowException() {
        taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
        taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));
        taskService.addTask(new Task("Task 3", Priority.LOW, 1));
        taskService.addTask(new Task("Task 4", Priority.HIGH, 5));
        taskService.addTask(new Task("Task 5", Priority.MEDIUM, 4));
        taskService.addTask(new Task("Task 6", Priority.LOW, 3));
        taskService.addTask(new Task("Task 7", Priority.HIGH, 7));
        taskService.addTask(new Task("Task 8", Priority.MEDIUM, 6));
        taskService.addTask(new Task("Task 9", Priority.LOW, 5));
        taskService.addTask(new Task("Task 10", Priority.HIGH, 9));
        taskService.addTask(new Task("Task 11", Priority.MEDIUM, 8));
        taskService.addTask(new Task("Task 12", Priority.LOW, 7));
        taskService.addTask(new Task("Task 13", Priority.HIGH, 11));
        taskService.addTask(new Task("Task 14", Priority.LOW, 7));
        taskService.addTask(new Task("Task 15", Priority.LOW, 9));
        taskService.addTask(new Task("Task 16", Priority.HIGH, 13));
        taskService.addTask(new Task("Task 17", Priority.MEDIUM, 12));
```

```
        taskService.addTask(new Task("Task 18", Priority.LOW, 11));
        taskService.addTask(new Task("Task 19", Priority.HIGH, 15));
        taskService.addTask(new Task("Task 20", Priority.MEDIUM, 14));
        taskService.addTask(new Task("Task 21", Priority.LOW, 13));

        assertThrows(IllegalArgumentException.class, () ->
taskService.findWhatTaskToDoNext(21));
    }
}
```

4.2. Acoperire la nivel de instrucțiune (Statement Coverage)

Trebuie să alegem date de test care ne obligă să trecem prin fiecare instrucțiune, fapt ce ne permite să identificăm logica faultoasă în cadrul programului. Pe lângă datele de intrare, rezultatul este determinat și de starea în care se află datele stocate de serviciu în lista tasks, deci le vom include și pe acestea în tabel. Seturile de date alese sunt:

Nr.	Numele metodei	Date de intrare / de setup (numberOfTasks)		Rezultat	Instrucțiuni
		tasks	numberOfTasks		
1.	givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask	Task("Task 2", Priority.MEDIUM, 2), Task("Task 1", Priority.HIGH, 3)	2	Alege Task 1	82,85-109, 86, 90, 93, 95, 96, 90, 93, 98-106, 100, 102, end
2.	givenTasksWithSamePriorityAndShorterTime_whenFindWhatTaskToDoNext_thenSelectShorterTimeTask	Task("Task 2", Priority.HIGH, 2), Task("Task 1", Priority.HIGH, 3)	2	Alege task 2	82, 85-109, 86, 90, 93, 95, 96, 90, 93, 95, 98-106, 100, 102, end
3.	givenNoAvailableTasks_whenFindWhatTaskToDoNext_thenThrowException	Task("Task 2", Priority.COMPLETE, 3), Task("Task 1", Priority.COMPLETE, 5)	2	No available tasks to do	82, 85-109, 86, 90, 93, 90, 93, 111, 112
4.	givenTasksWithDifferentPrioritiesAndTimeEstimates_whenFindWhatTaskToDoNext_thenSel	Task("Task 1", Priority.LOW, 3), Task("Task 2", Priority.MEDIUM, 2),	3	Task 3	82, 86-109, 90, 93, 95, 96, 90, 93, 95, 98-196, 100, 102, 90, 93, 95, 98-196,

	ectHighestPriorityAndShortestTimeTask	Task("Task 3", Priority.HIGH, 1)			100, 102, 90, 93, 95, 98-196, 100, 102, 111, end
5.	givenTooLowIndex_whenFindWhatTaskToDoNext_thenThrowException	-	0	Invalid number of tasks or empty list	82, 86-109, 87
6.	givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException()	Task("Task 1", Priority.HIGH, 3), Task("Task 2", Priority.MEDIUM, 2)	3	Number of tasks does not match the size of the tasks list	82, 83

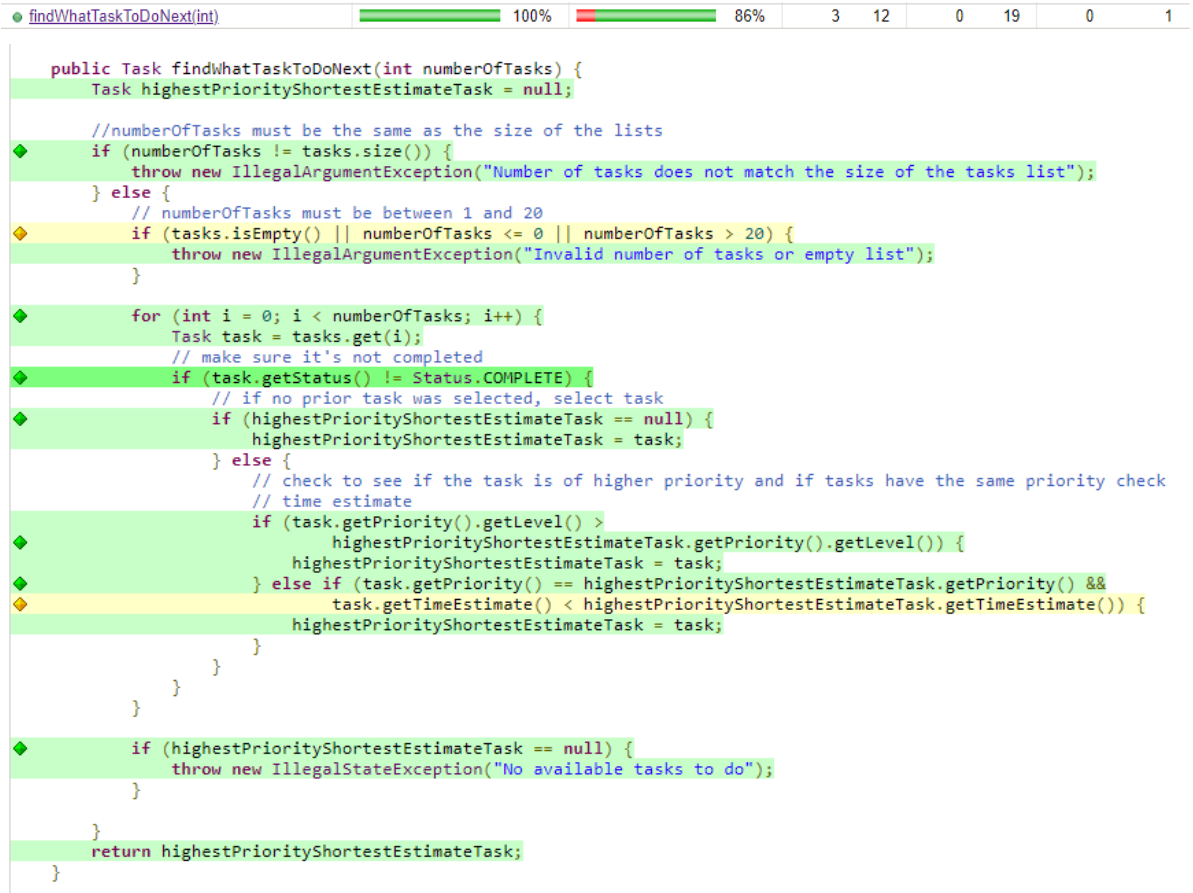
După cum se observă pe coloana de rezultate, testele acoperă în mod minimal instrucțiunile metodei. Totuși, asta nu înseamnă că am acoperit toate ramurile și toate condițiile. Pentru acestea, e nevoie să scriem mai multe metode, pe care le vom acoperi în următoarele secțiuni.

- Conform analizei date de pluginul integrat, Code Coverage for Java, rezultatele sunt următoarele:

```
78      public Task findWhatTaskToDoNext(int numberOfTasks) {
79          Task highestPriorityShortestEstimateTask = null;
80
81          //numberOfTasks must be the same as the size of the lists
82          if (numberOfTasks != tasks.size()) {
83              throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
84          } else {
85              // numberOfTasks must be between 1 and 20
86              if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
87                  throw new IllegalArgumentException("Invalid number of tasks or empty list");
88              }
89
90              for (int i = 0; i < numberOfTasks; i++) {
91                  Task task = tasks.get(i);
92                  // make sure it's not completed
93                  if (task.getStatus() != Status.COMPLETE) {
94                      // if no prior task was selected, select task
95                      if (highestPriorityShortestEstimateTask == null) {
96                          highestPriorityShortestEstimateTask = task;
97                      } else {
98                          // check to see if the task is of higher priority and if tasks have the same priority check
99                          // time estimate
100                         if (task.getPriority().getLevel() >
101                                 highestPriorityShortestEstimateTask.getPriority().getLevel()) {
102                             highestPriorityShortestEstimateTask = task;
103                         } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
104                                 task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
105                             highestPriorityShortestEstimateTask = task;
106                         }
107                     }
108                 }
109             }
110
111             if (highestPriorityShortestEstimateTask == null) {
112                 throw new IllegalStateException("No available tasks to do");
113             }
114
115         }
116         return highestPriorityShortestEstimateTask;
117     }
118
```

Linia 87 este colorata cu roșu,indicand faptul ca testele nu o acoperă, deși testele din suita acoperă toate cazurile inclusiv excepția în care numărul de task-uri este invalid. (Implicit testul 5)

- Conform analizei date de pluginul JaCoCo, rezultatele sunt următoarele:



Liniile 86 și 104 sunt colorate cu galben deoarece nu sunt toate branch-urile atinse.

4.2. Acoperire la nivel de ramură (Branch Coverage)

Această metodă vine în completarea celei precedente pentru a verifica că fiecare ramură este atinsă, inclusiv ramurile negative care nu sunt tratate cu „else”.

Nr.	Decizie
1	if (numberOfTasks != tasks.size())
2	if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20)
3	for (int i = 0; i < numberOfTasks; i++)
4	if (task.getStatus() != Status.COMPLETE)
5	if (highestPriorityShortestEstimateTask == null)

6	if (task.getPriority().getLevel() > highestPriorityShortestEstimateTask.getPriority().getLevel())
7	if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() && task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate())

Nr.	Numele metodei	Date de intrare / de setup (numberOfTasks)		Rezultat	Decizie acoperită
		tasks	numberOf Tasks		
1.	givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException()	Task("Task 1", Priority.HIGH, 3), Task("Task 2", Priority.MEDIUM, 2)	3	Number of tasks does not match the size of the tasks list	82
2.	givenTooHighNoOfTasks_whenFindWhatTaskToDoNext_thenThrowException	[lista de 21 de taskuri, de văzut în cod]	21	Invalid number of tasks or empty list	82,86
3.	givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask	Task("Task 2", Priority.MEDIUM, 2), Task("Task 1", Priority.HIGH, 3)	2	Task 1	82, 86, 93, 95, 100. 111
4.	givenTasksWithSamePriorityAndShorterTime_whenFindWhatTaskToDoNext_thenSelectShorterTimeTask	Task("Task 2", Priority.HIGH, 2), Task("Task 1", Priority.HIGH, 3)	2	Task 2	82,86,93,95, 103, 111
5.	givenTasksWithDifferentPrioritiesAndTimeEstimates_whenFindWhatTaskToDoNext_thenSelectHighestPriorityAndShortestTimeTask	Task("Task 1", Priority.LOW, 3), Task("Task 2", Priority.HIGH, 2), Task("Task 3", Priority.HIGH, 1)	3	Task 3	82,86,93,95,100, 103, 111

6.	givenNoAvailableTasks_whenFindWhatTaskToDoNext_thenThrowException	Task("Task 2", Priority.COMPLETE, 3), Task("Task 1", Priority.COMPLETE, 5)	2	No available tasks to do	82,86,93, 111

Testele care acoperă aceste decizii sunt:

- Conform analizei date de pluginul integrat, Code Coverage for Java, rezultatele sunt următoarele:

```
77
78      public Task findWhatTaskToDoNext(int numberOfTasks) {
79          Task highestPriorityShortestEstimateTask = null;
80
81          //numberOfTasks must be the same as the size of the lists
82          if (numberOfTasks != tasks.size()) {
83              throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
84          } else {
85              // numberOfTasks must be between 1 and 20
86              if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
87                  throw new IllegalArgumentException("Invalid number of tasks or empty list");
88              }
89
90              for (int i = 0; i < numberOfTasks; i++) {
91                  Task task = tasks.get(i);
92                  // make sure it's not completed
93                  if (task.getStatus() != Status.COMPLETE) {
94                      // if no prior task was selected, select task
95                      if (highestPriorityShortestEstimateTask == null) {
96                          highestPriorityShortestEstimateTask = task;
97                      } else {
98                          // check to see if the task is of higher priority and if tasks have the same priority check
99                          // time estimate
100                         if (task.getPriority().getLevel() >
101                                 highestPriorityShortestEstimateTask.getPriority().getLevel()) {
102                             highestPriorityShortestEstimateTask = task;
103                         } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
104                                 task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
105                             highestPriorityShortestEstimateTask = task;
106                         }
107                     }
108                 }
109             }
110
111             if (highestPriorityShortestEstimateTask == null) {
112                 throw new IllegalStateException("No available tasks to do");
113             }
114
115         }
116         return highestPriorityShortestEstimateTask;
117     }
```

Toate cazurile sunt atinse

- Conform analizei date de pluginul JaCoCo, rezultatele sunt următoarele:

```
74.    public Task findWhatTaskToDoNext(int numberOfTasks) {
75.        Task highestPriorityShortestEstimateTask = null;
76.
77.        //numberOfTasks must be the same as the size of the lists
78.        if (numberOfTasks != tasks.size()) {
79.            throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
80.        } else {
81.            // numberOfTasks must be between 1 and 20
82.            if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
83.                throw new IllegalArgumentException("Invalid number of tasks or empty list");
84.            }
85.
86.            for (int i = 0; i < numberOfTasks; i++) {
87.                Task task = tasks.get(i);
88.                // make sure it's not completed
89.                if (task.getStatus() != Status.COMPLETE) {
90.                    // if no prior task was selected, select task
91.                    if (highestPriorityShortestEstimateTask == null) {
92.                        highestPriorityShortestEstimateTask = task;
93.                    } else {
94.                        // check to see if the task is of higher priority and if tasks have the same priority check
95.                        // time estimate
96.                        if (task.getPriority().getLevel() >
97.                                highestPriorityShortestEstimateTask.getPriority().getLevel()) {
98.                            highestPriorityShortestEstimateTask = task;
99.                        } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
100.                               task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
101.                           highestPriorityShortestEstimateTask = task;
102.                       }
103.                   }
104.               }
105.           }
106.
107.           if (highestPriorityShortestEstimateTask == null) {
108.               throw new IllegalStateException("No available tasks to do");
109.           }
110.
111.       }
112.       return highestPriorityShortestEstimateTask;
113.   }
114.
```

(linia 82 arata 2/6 branches missed, linia 100 arata 1/2 branches missed)

4.3. Acoperire la nivel de condiție(Condition Coverage)

Această metodă este o extensie a metodei precedente, astfel încât urmărește mai mult decât asignarea de valori de adevăr fiecărei decizii, și nume asignarea de valori de adevăr a fiecărei condiție care constituie fiecare decizie.

Nr.	Decizie	Condiții individuale
1	if (numberOfTasks != tasks.size())	numberOfTasks != tasks.size()
2	if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20)	tasks.isEmpty()
3		numberOfTasks <= 0, numberOfTasks > 20
4	for (int i = 0; i < numberOfTasks; i++)	i < numberOfTasks
5	if (task.getStatus() != Status.COMPLETE)	task.getStatus() != Status.COMPLETE
6	if (highestPriorityShortestEstimateTask == null)	highestPriorityShortestEstimateTask == null
7	if (task.getPriority().getLevel() > highestPriorityShortestEstimateTask.get	task.getPriority().getLevel() > highestPriorityShortestEstimateTask.getPr

	Priority().getLevel())	iority().getLevel()
8	if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() && task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate())	task.getPriority() == highestPriorityShortestEstimateTask.getPriority()
9		task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()

Legătura dintre testele implementate și fiecare condiție este:

Nr.	Numele metodei	Date de intrare / de setup (numberOfTasks)		Rezultat	Condiție acoperită
		tasks	numberOfTasks		
1.	givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException()	Task("Task 1", Priority.HIGH, 3), Task("Task 2", Priority.MEDIUM, 2)	3	Number of tasks does not match the size of the tasks list	82
2.	givenTooHighNoOfTasks_whenFindWhatTaskToDoNext_thenThrowException	[lista de 21 de taskuri, de văzut în cod]	21	Invalid number of tasks or empty list	82,86
3.	givenTooLowIndex_whenFindWhatTaskToDoNext_thenThrowException	-	0	Invalid number of tasks or empty list	82,86
4.	givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask	Task("Task 2", Priority.MEDIUM, 2), Task("Task 1", Priority.HIGH, 3)	2	Task 1	82, 86, 93, 95, 100, 111
5.	givenTasksWithSamePriorityAndShorterTime_whenFindWhatTaskTo	Task("Task 2", Priority.HIGH, 2),	2	Task 2	82, 86, 93, 100, 103, 111

	DoNext_thenSelectShorterTimeTask	Task("Task 1", Priority.HIGH, 3)			
6.	givenTasksWithDifferentPrioritiesAndTimeEstimates_whenFindWhatTaskToDoNext_thenSelectHighestPriorityAndShortestTimeTask	Task("Task 1", Priority.LOW, 3), Task("Task 2", Priority.MEDIUM, 2), Task("Task 3", Priority.HIGH, 1)	3	Task 3	82, 86, 93, 95, 100, 111
7.	givenNoAvailableTasks_whenFindWhatTaskToDoNext_thenThrowException	Task("Task 2", Priority.COMPLETE, 3), Task("Task 1", Priority.COMPLETE, 5)	2	No available tasks to do	82, 86, 93, 111

- Conform analizei date de pluginul integrat, Code Coverage for Java, rezultatele sunt următoarele:

```
77
78      public Task findWhatTaskToDoNext(int numberOfTasks) {
79          Task highestPriorityShortestEstimateTask = null;
80
81          //numberOfTasks must be the same as the size of the lists
82          if (numberOfTasks != tasks.size()) {
83              throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
84          } else {
85              // numberOfTasks must be between 1 and 20
86              if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
87                  throw new IllegalArgumentException("Invalid number of tasks or empty list");
88              }
89
90              for (int i = 0; i < numberOfTasks; i++) {
91                  Task task = tasks.get(i);
92                  // make sure it's not completed
93                  if (task.getStatus() != Status.COMPLETE) {
94                      // if no prior task was selected, select task
95                      if (highestPriorityShortestEstimateTask == null) {
96                          highestPriorityShortestEstimateTask = task;
97                      } else {
98                          // check to see if the task is of higher priority and if tasks have the same priority check
99                          // time estimate
100                         if (task.getPriority().getLevel() >
101                                 highestPriorityShortestEstimateTask.getPriority().getLevel()) {
102                             highestPriorityShortestEstimateTask = task;
103                         } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
104                                 task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
105                             highestPriorityShortestEstimateTask = task;
106                         }
107                     }
108                 }
109             }
110
111             if (highestPriorityShortestEstimateTask == null) {
112                 throw new IllegalStateException("No available tasks to do");
113             }
114
115         }
116         return highestPriorityShortestEstimateTask;
117     }
118
```

Toate conditiile sunt atinse

- Conform analizei date de pluginul JaCoCo, rezultatele sunt următoarele:

```
74.     public Task findWhatTaskToDoNext(int numberOfTasks) {
75.         Task highestPriorityShortestEstimateTask = null;
76.
77.         //numberOfTasks must be the same as the size of the lists
78.         if (numberOfTasks != tasks.size()) {
79.             throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
80.         } else {
81.             // numberOfTasks must be between 1 and 20
82.             if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
83.                 throw new IllegalArgumentException("Invalid number of tasks or empty list");
84.             }
85.
86.             for (int i = 0; i < numberOfTasks; i++) {
87.                 Task task = tasks.get(i);
88.                 // make sure it's not completed
89.                 if (task.getStatus() != Status.COMPLETE) {
90.                     // if no prior task was selected, select task
91.                     if (highestPriorityShortestEstimateTask == null) {
92.                         highestPriorityShortestEstimateTask = task;
93.                     } else {
94.                         // check to see if the task is of higher priority and if tasks have the same priority check
95.                         // time estimate
96.                         if (task.getPriority().getLevel() >
97.                                 highestPriorityShortestEstimateTask.getPriority().getLevel()) {
98.                             highestPriorityShortestEstimateTask = task;
99.                         } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
100.                                task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
101.                            highestPriorityShortestEstimateTask = task;
102.                        }
103.                    }
104.                }
105.            }
106.
107.            if (highestPriorityShortestEstimateTask == null) {
108.                throw new IllegalStateException("No available tasks to do");
109.            }
110.
111.        }
112.        return highestPriorityShortestEstimateTask;
113.    }
```

(linia 82 arata 1/6 branches missed, linia 100 arata 1/2 branches missed)

4.4. Circuite independente

Conform formulei lui McCabe pentru Complexitate Ciclomatică:

Dat fiind un graf complet conectat G cu e arce și n noduri, atunci numărul de circuite linear independente este dat de:

V(G) = e – n + 1, unde e = numărul de muchii ale graficului, n = numărul de noduri ale graficului

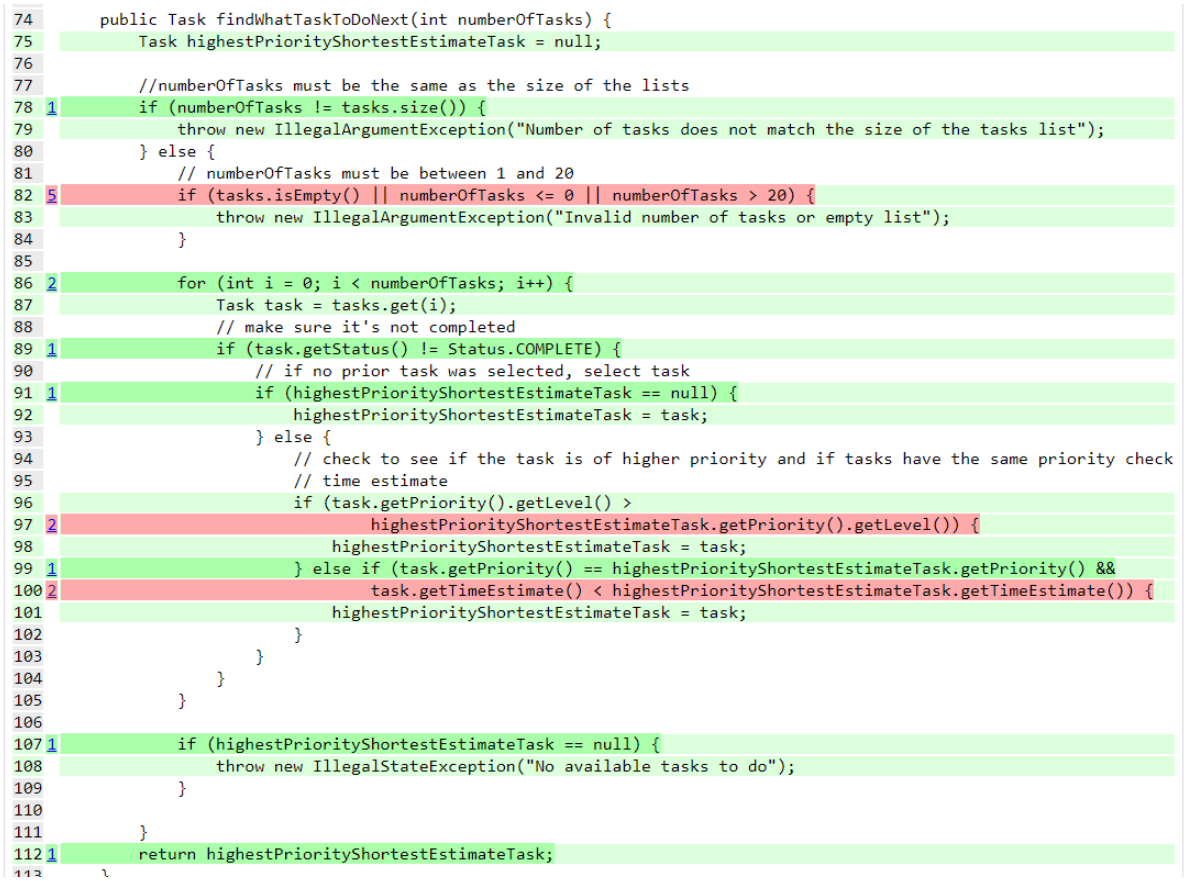
În cadrul grafului nostru, n=16, e=20, deci V(G) = 20-16+1 = 5.

5. Analiza mutanților

Metodele prezentate în secțiunea anterioară au ca dezavantaj evident faptul că testează faptul că liniile de cod au fost executate, nu și că use-case-urile dorite au fost testate cu succes. Astfel, acestea sunt ineficiente ca metrică de măsurare a calității.

Scopul mutation testing este îmbunătățirea testelor și identificarea defectelor din cod schimbând dinamic forma codului sursă și urmând principiul „Good tests shall fail”. Un set de teste bine format ar trebui să poată identifica modificări în logica codului sursă, și să fail în consecință.

Pentru generarea mutanților am folosit generatorul PITest și am exemplificat pe testele scrise în capitolul anterior. Raportul generat de PITest este:



```
74      public Task findWhatTaskToDoNext(int numberOfTasks) {
75          Task highestPriorityShortestEstimateTask = null;
76
77          //numberOfTasks must be the same as the size of the lists
78  1       if (numberOfTasks != tasks.size()) {
79              throw new IllegalArgumentException("Number of tasks does not match the size of the tasks list");
80          } else {
81              // numberOfTasks must be between 1 and 20
82  5           if (tasks.isEmpty() || numberOfTasks <= 0 || numberOfTasks > 20) {
83                  throw new IllegalArgumentException("Invalid number of tasks or empty list");
84              }
85
86  2           for (int i = 0; i < numberOfTasks; i++) {
87                  Task task = tasks.get(i);
88                  // make sure it's not completed
89  1               if (task.getStatus() != Status.COMPLETE) {
90                      // if no prior task was selected, select task
91  1                   if (highestPriorityShortestEstimateTask == null) {
92                          highestPriorityShortestEstimateTask = task;
93                      } else {
94                          // check to see if the task is of higher priority and if tasks have the same priority check
95                          // time estimate
96                          if (task.getPriority().getLevel() >
97  2                               highestPriorityShortestEstimateTask.getPriority().getLevel()) {
98                              highestPriorityShortestEstimateTask = task;
99  1                       } else if (task.getPriority() == highestPriorityShortestEstimateTask.getPriority() &&
100 2                              task.getTimeEstimate() < highestPriorityShortestEstimateTask.getTimeEstimate()) {
101                             highestPriorityShortestEstimateTask = task;
102                         }
103                     }
104                 }
105             }
106
107 1           if (highestPriorityShortestEstimateTask == null) {
108                 throw new IllegalStateException("No available tasks to do");
109             }
110
111         }
112 1       return highestPriorityShortestEstimateTask;
113     }
```

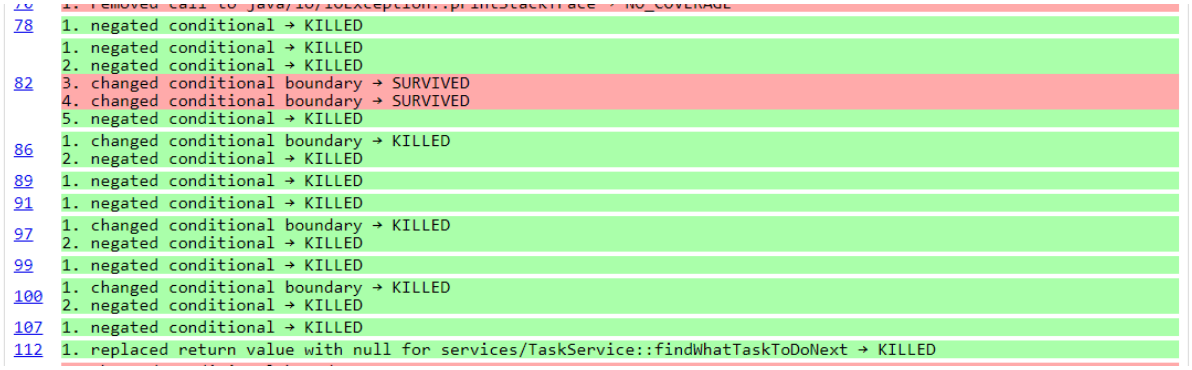
```
78  1. negated conditional → KILLED
    1. negated conditional → KILLED
    2. negated conditional → KILLED
82  3. changed conditional boundary → SURVIVED
    4. changed conditional boundary → SURVIVED
    5. negated conditional → KILLED
86  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
89  1. negated conditional → KILLED
91  1. negated conditional → KILLED
97  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
99  1. negated conditional → KILLED
100 1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
107 1. negated conditional → KILLED
112 1. replaced return value with null for services/TaskService::findWhatTaskToDoNext → KILLED
```

Testele noastre au omorat 13/17 mutanți generați. Observăm că pentru liniile de cod 97 și 100 au supraviețuit mutanții care au transformat < în <=. Testul corespunzător pentru a omorî acești mutanți este:

```
@Test
void givenBestTasksWIthSameTimeEstimate_whenFindWhatTaskToDoNext_thenReturnFirst()
{
   taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
   taskService.addTask(new Task("Task 2", Priority.HIGH, 3));

   int numberOfTasks = 2;

   Task returnedTask = taskService.findWhatTaskToDoNext(numberOfTasks);
   Task expectedTask = taskService.getByIndex(0);

   assertEquals(expectedTask, returnedTask);
```

```
}
```

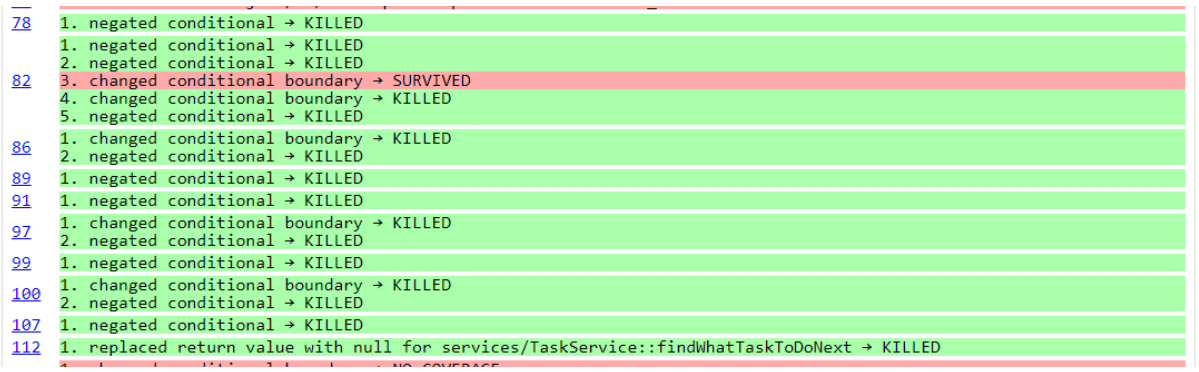
Acesta testează și cazul în care primim 2 taskuri cu aceeași prioritate și time estimate.



Pentru mutantul 4 generat la linia 82, într-adevar nu am scris teste de analiză a valorilor de frontieră, deci schimbarea conditional boundary nu este detectată. Pentru a omorî acest mutant am scris testul:

```
@Test
void givenMaximumNumberOfTasks_whenFindWhatTaskToDoNext_thenReturnTask() {
   taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
   taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));
   taskService.addTask(new Task("Task 3", Priority.LOW, 1));
   taskService.addTask(new Task("Task 4", Priority.HIGH, 5));
   taskService.addTask(new Task("Task 5", Priority.MEDIUM, 4));
   taskService.addTask(new Task("Task 6", Priority.LOW, 3));
   taskService.addTask(new Task("Task 7", Priority.HIGH, 7));
   taskService.addTask(new Task("Task 8", Priority.MEDIUM, 6));
   taskService.addTask(new Task("Task 9", Priority.LOW, 5));
   taskService.addTask(new Task("Task 10", Priority.HIGH, 9));
   taskService.addTask(new Task("Task 11", Priority.MEDIUM, 8));
   taskService.addTask(new Task("Task 12", Priority.LOW, 7));
   taskService.addTask(new Task("Task 13", Priority.HIGH, 11));
   taskService.addTask(new Task("Task 14", Priority.LOW, 7));
   taskService.addTask(new Task("Task 15", Priority.LOW, 9));
   taskService.addTask(new Task("Task 16", Priority.HIGH, 13));
   taskService.addTask(new Task("Task 17", Priority.MEDIUM, 12));
   taskService.addTask(new Task("Task 18", Priority.LOW, 11));
   taskService.addTask(new Task("Task 19", Priority.HIGH, 15));
   taskService.addTask(new Task("Task 20", Priority.MEDIUM, 14));

   Task expectedResult = taskService.getByIndex(0);

   assertEquals(expectedResult, taskService.findWhatTaskToDoNext(20));
}
```

Se observă la o nouă rulare că mutantul a fost omorât.



6. Analiză și comparare a testelor folosind Codium

Pentru a verifica eficacitatea și acuratețea testelor pe care le-am scris pentru metoda findWhatTaskToDoNext am folosit Codium. Acest plugin oferă un set variat de functionalitati care ajuta la îmbunătățirea procesului de scriere a codului, iar printre acestea se numara si Codiumate. Codiumate este un companion care utilizează AI pentru a îmbunătăți calitatea codului prin generarea de teste, scrierea și analizarea de cod.

Asa cum spuneam, Codiumate oferă o serie de functionalitati pentru îmbunătățirea procesului de dezvoltare software, inclusiv generarea automată de teste. Interfata include trei taburi: Test Suite, Code Explanation si Code Suggestions.

Code Explanation oferă o analiza detaliată a codului sursa, inclusiv date de intrare/ieșire, descrierea functionalitatii metodei și a flow-ului.

Revenind la functionalitatea de generare automată a testelor, Behaviors Coverage arata ca au fost identificate 14 comportamente sau scenarii diferite pentru metoda data, findWhatTaskToDoNext. Acestea sunt împărțite în trei categorii principale: happy path, edge case si other.

6.1. Happy Path

Happy Path se refera la scenariul ideal in care doul functioneaza fara probleme, deci intrarile sunt valide, iar procesul se desfășoară conform așteptărilor.

În cadrul scenariului Happy Path, Codium a identificat următoarele comportamente ale metodei findWhatTaskToDoNext():

1. Returns the task with the highest priority and shortest time estimate that is not completed
2. Returns the only available task if there is only one task and it is not completed
3. Returns the only available task if there are multiple tasks with the same priority and time estimate and they are not completed
4. Returns the only available task if there are multiple tasks with the same priority and time estimate and only one of them is not completed
5. Returns the only available task if there are multiple tasks with the same priority and time estimate and all of them are not completed
6. Returns the task with the highest priority and shortest time estimate if there are multiple tasks with different priorities and time estimates and they are not completed

7. Returns the task with the highest priority and shortest time estimate if there are multiple tasks with different priorities and time estimates and only one of them is not completed
8. Returns the task with the highest priority and shortest time estimate if there are multiple tasks with different priorities and time estimates and all of them are not completed

Pentru cazul returns the task with the highest priority and shortest time estimate that is not completed Codiumate genrează un test numit test_returns_highest_priority_shortest_time_estimate_task, echivalent testului givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask scris de noi. Să le analizăm mai îndeaproape

test_returns_highest_priority_shortest_time_estimate_task	givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask

@Test
public void test_returns_highest_priority_shortest_time_estimate_task() {
 	
  TaskService taskService = new TaskService();
 	
  Task task1 = new Task("Task 1", Priority.HIGH, 2, Status.IN_PROGRESS);
  Task task2 = new Task("Task 2", Priority.LOW, 4, Status.IN_PROGRESS);
  Task task3 = new Task("Task 3", Priority.MEDIUM, 3, Status.IN_PROGRESS);
 	
  taskService.addTask(task1);
  taskService.addTask(task2);
  taskService.addTask(task3);
 	
  Task result = taskService.findWhatTaskToDoNext(3);
 	
  assertEquals(task1, result);
}
	@Test
	void givenTasksWithDifferentPriorities_whenFindWhatTaskToDoNext_thenSelectHighestPriorityTask()
	  {
	  // adding two tasks with different priorities to the list of tasks
	  taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
	  taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));
	
	  Integer numberOfTasks = 2;
	
	  Task returnedTask = taskService.findWhatTaskToDoNext(numberOfTasks);
	  Task expectedTask = taskService.getByIndex(0);
	
	  assertEquals(expectedTask, returnedTask);
	  }

Diferența principală dintre testul generat de AI și testul scris manual constă în setul de date. Testul generat de AI adauga in task service 3 taskuri cu priorități diferite (HIGH, LOW si MEDIUM), în timp ce în testul scris manual am adăugat doar două taskuri cu priorități diferite (MEDIUM si HIGH) in listă.

6.2. Edge case

În ceea ce privește edge cases, acestea sunt scenariile care se abat de la fluxul ideal sau care prezintă situații neașteptate sau limită. Edge cases implică intrări invalide, limitele superioare/ inferioare ale valorilor etc. Iată ce situații a identificat Codium pentru metoda findWhatTaskToDoNext:

1. Throws an IllegalArgumentException if the number of tasks does not match the size of the tasks list
2. Throws an IllegalArgumentException if the tasks list is empty or the number of tasks is less than or equal to 0 or greater than 20
3. Throws an IllegalStateException if there are no available tasks to do

Alegând scenariul Throws an IllegalArgumentException if the number of tasks does not match the size of the tasks list, Codium generează un test test_throws_illegal_argument_exception_if_number_of_tasks_does_not_match_list_size care corespunde testului givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException scris manual.

Să analizăm testele:

test_throws_illegal_argument_exception_if_number_of_tasks_does_not_match_list_size	givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException

@Test
public void test_throws_illegal_argument_exception_if_number_of_tasks_does_not_match_list_size() {
TaskService taskService = new TaskService();
Task task1 = new Task("Task 1", Priority.HIGH, 2, Status.IN_PROGRESS);
Task task2 = new Task("Task 2", Priority.LOW, 4, Status.IN_PROGRESS);
taskService.addTask(task1);
taskService.addTask(task2);
	@Test
	void givenDifferentNumberOfTasksAndTasksListSize_whenFindWhatTaskToDoNext_thenThrowException() {
	taskService.addTask(new Task("Task 1", Priority.HIGH, 3));
	taskService.addTask(new Task("Task 2", Priority.MEDIUM, 2));
	Integer numberOfTasks = 3;

assertThrows(IllegalArgumentException.class, () -> {
 	
taskService.findWhatTaskToDoNext(3);
});
}
	assertThrows(IllegalArgumentException.class, () ->
	taskService.findWhatTaskToDoNext(numberOfTasks));
	}
	
	}

Se poate observa că nu există diferențe majore între cele două. Testul generat de IA urmează același principiu ca testul scris manual: se adaugă doua taskuri în lista și de apelează metoda pentru 3 taskuri.

6.3. Other

În cadrul categoriei other se regăsesc următoarele scenarii:

1. Returns the only available task if there is only one task and it is completed
2. Returns the task with the highest priority and shortest time estimate if there are multiple tasks with different priorities and time estimates and some of them are completed
3. Returns the task with the highest priority and shortest time estimate if there are multiple tasks with different priorities and time estimates and some of them are completed and some are not completed

7. Bibliografie

Cursuri
https://junit.org/junit5/docs/current/user-guide/
https://www.baeldung.com/java-junit-test-suite
https://www.baeldung.com/cs/software-testing-equivalence-partitioning
https://www.geeksforgeeks.org/structural-software-testing/
https://www.baeldung.com/java-mutation-testing-with-pitest
https://codiumate-docs.codium.ai/