Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»
Інститут прикладного системного аналізу
Кафедра системного ароектування

Лабораторна робота №5

з курсу «*Проектування інформаційних систем*»

на тему: «Модульне тестуваня (Unit-тести) та рефакторинг»

Виконав:
студент групи ДА-71
Стефура Олег

Київ — 2020

Мета: оволодіти навичками створення програмного забезпечення за метолологією TDD та ознайомитися з процедурами рефакторинга.

Задача:

1. Використовувати методологію Test Driven Development для створення класів архітектурної програмної моделі.
2. Скласти тестові сценарії, які продемонструють функціювання всіх методів проектованої моделі.
3. Виконати юніт-тестування складових частин (внутрішніх класів), що реалізують об'єкт моделювання.
4. Виконати "зовнішнє" юніт-тестування для API.
5. Провести рефакторинг коду програми, для поліпшення реалізації.

Завдання:

1. Розробити методику випробувань з використанням ISO/IEC/IEEE 29119.
2. Розробити код програми архітектурної моделі. Використовувати Test Driven Development.
3. Провести рефакторинг коду програми, щоб задовольнити вимоги технічного завдання.

Хід роботи

1. TDD

Проведемо розробку классу GoalService, який повинен надавати функції обробки моделей «Goal» програми, застосовуючи підхід Test Driven Development (TDD). На початковому етапі класс GoalService порожній, тестів немає. Створимо перший тест. Розпочнемо із додавання нових цілей.

```java
package com.delta.GoalTracker.services;

import com.delta.GoalTracker.repositories.GoalRepository;
import com.delta.GoalTracker.models.Goal;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class GoalServiceTest {
    @Autowired
    private GoalService goalService;

    @Autowired
    private GoalRepository goalRepository;

    @Test
    void addGoalTest() {
        goalService.addGoal(Goal goal);
    }
}
```

Рисунок 5.1 – Перша ітерація TDD

Реалізуємо метод, щоб задовольнити тест.

```java
@Service
public class GoalService {
    @Autowired
    private GoalRepository goalRepository;

    public Goal addGoal(Goal newGoal) {
        return new Goal( name: "New Goal", date: "2020-28-12", status: "PRIVATE", importance: "MEDIUM");
    }
}
```

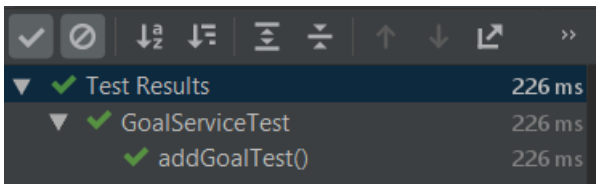


Рисунок 5.2 – Кінець першої ітерації

Перейдемо далі. Розширимо тест.

```
    @Test
    void addGoalTest() {
        goalService.addGoal(new Goal( name: "New Goal",  date: "2020-28-12",  status: "PRIVATE",  importance: "MEDIUM"));

        assertEquals( expected: 1, goalRepository.findAll().size());
    }
}
```

```
Tests failed: 1 of 1 test – 514 ms

    from
        goals goal0_

org.opentest4j.AssertionFailedError:
Expected :1
Actual   :0
 <Click to see difference>
```

Рисунок 5.3 – Друга ітерація TDD

Реалізуємо відповідну функціональність.

```
@Service
public class GoalService {
    @Autowired
    private GoalRepository goalRepository;

    public Goal addGoal(Goal newGoal) {
        return goalRepository.save(new Goal( name: "New Goal",  date: "2020-28-12",  status: "PRIVATE",  importance: "MEDIUM"));
    }
}
```

```
Test Results                604 ms
  GoalServiceTest           604 ms
    addGoalTest()           604 ms
```

Рисунок 5.4 – Кінець другої ітерації

Рухаємося далі:

```
    @Test
    void addGoalTest() {
        goalService.addGoal(new Goal( name: "Other goal",  date: "2020-30-12",  status: "PUBLIC",  importance: "MEDIUM"));

        assertEquals( expected: 1, goalRepository.findAll().size());
        assertEquals( expected: 1, goalRepository.findByStatus("PUBLIC").size());
    }
}
```

```
Run:  GoalServiceTest.addGoalTest

Tests failed: 1 of 1 test – 586 ms

Test Results                586 ms
  GoalServiceTest           586 ms
    addGoalTest()           586 ms

        goal0_.status=?

org.opentest4j.AssertionFailedError:
Expected :1
Actual   :0
 <Click to see difference>
```

Рисунок 5.5 – Третя ітерація

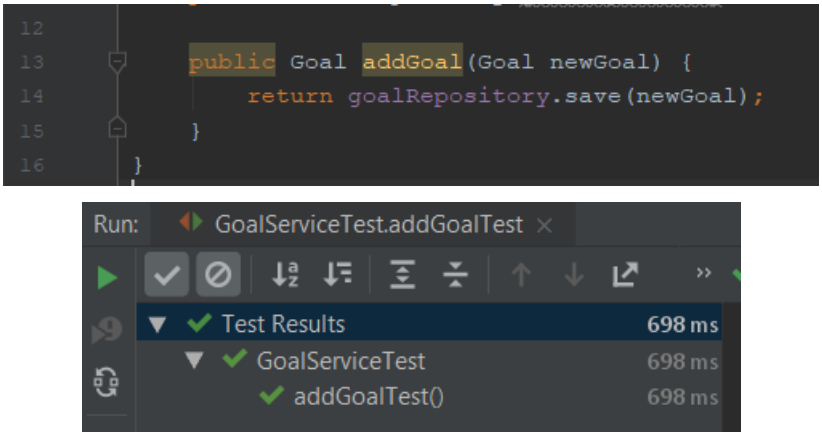


Рисунок 5.6 – Кінець третьої ітерації

На даній ітерації було завершено реалізацію методу додавання нової цілі. На наступній ітерації переходимо до створення нового тесту.

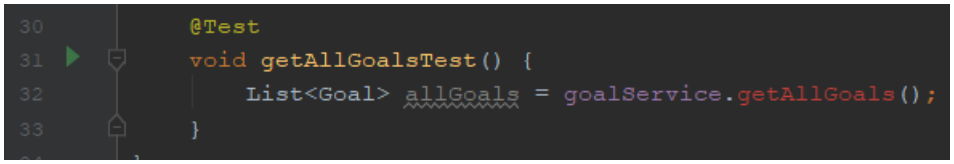


Рисунок 5.7 – Четверта ітерація TDD

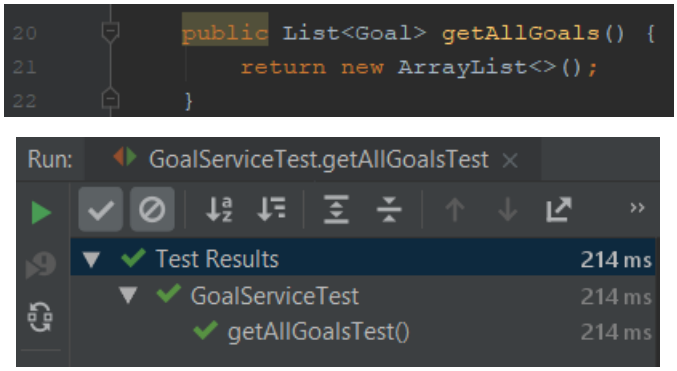
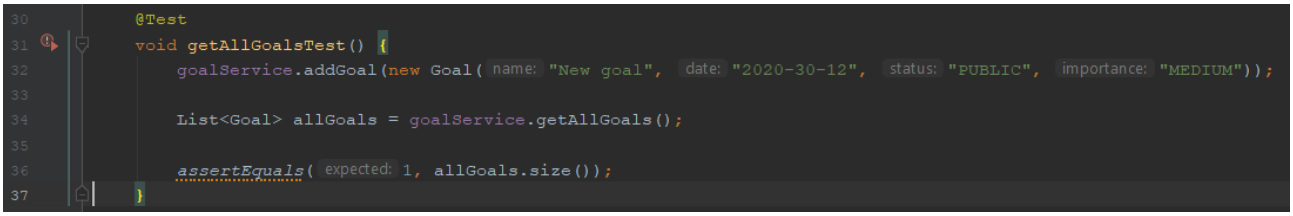


Рисунок 5.8 – Кінець четвертої ітерації



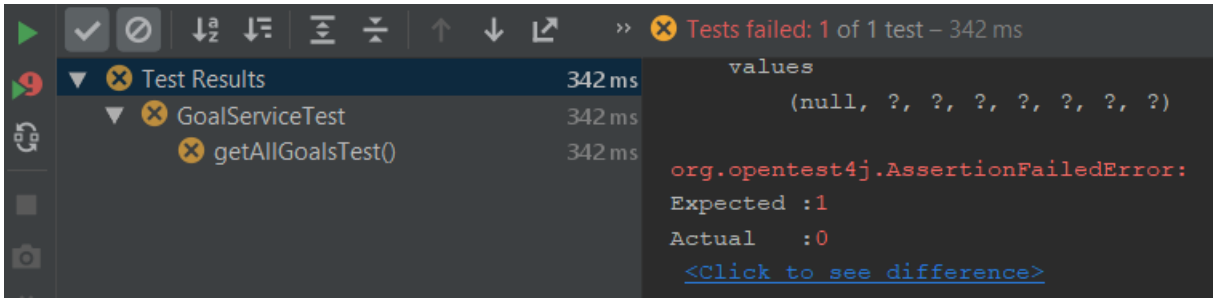


Рисунок 5.9 – П’ята ітерація

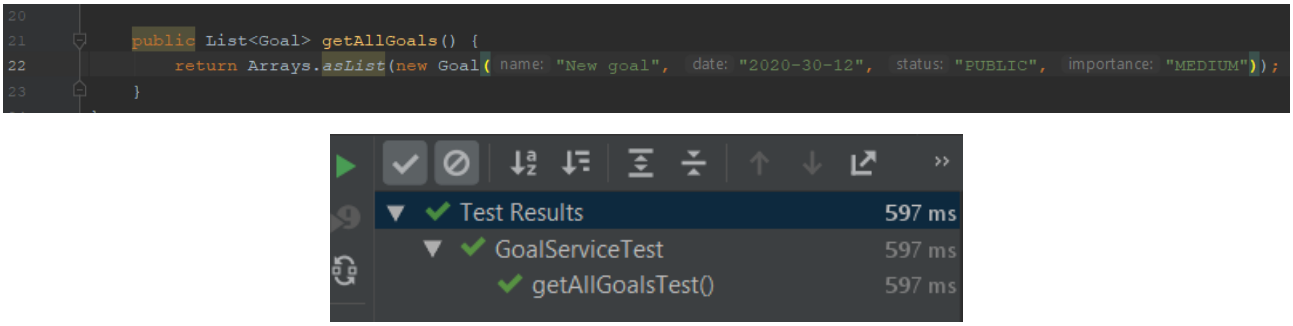
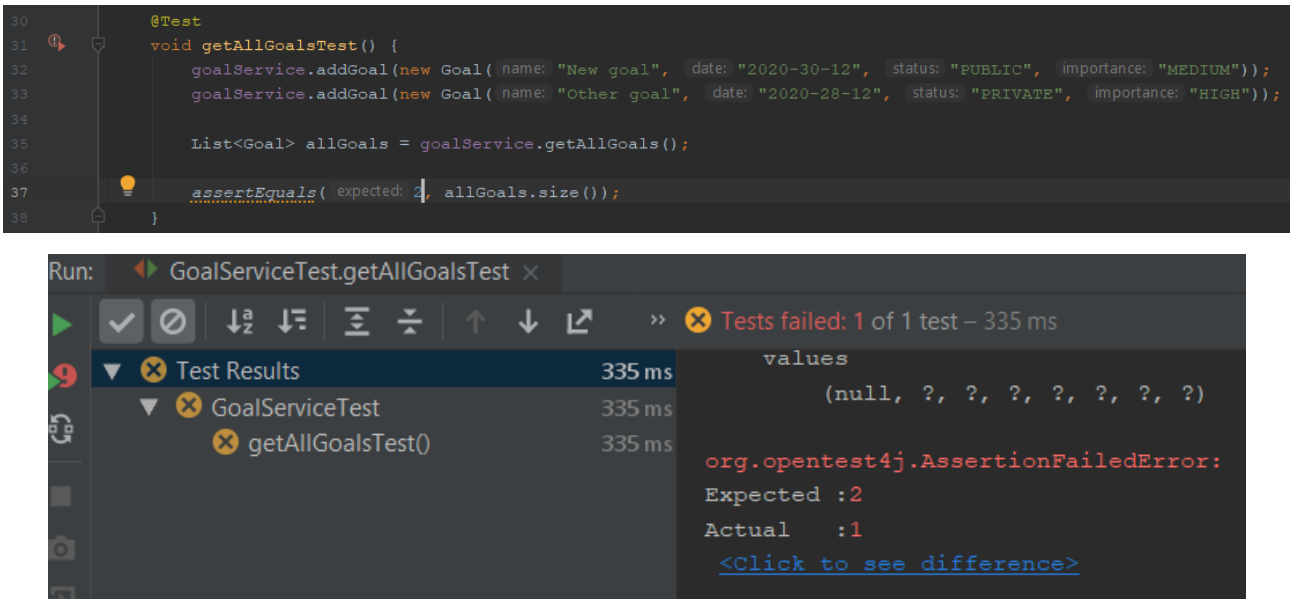


Рисунок 5.10 Кінець п'ятої ітерації



5.11 – Шоста ітерація

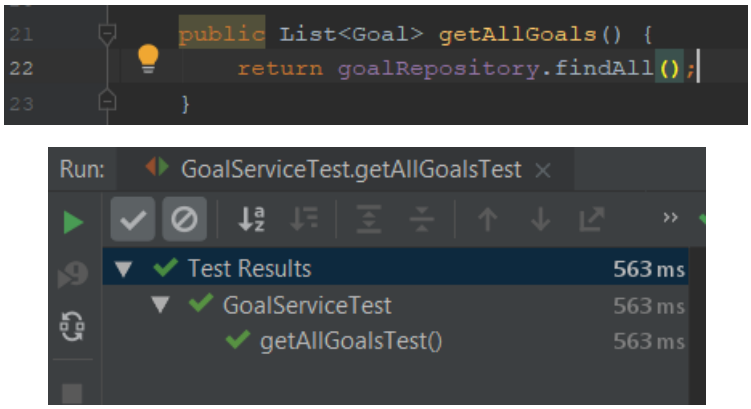


Рисунок 5.12 – Кінець шостої ітерації

На шостій ітерації закінчили реалізацію другого методу. Продовжуючи далі, весь необхідний функціонал класу буде реалізовано належним чином.

2. Unit-тести

Остаточно покриємо клас GoalService тестами щоб переконатися в тому, що усі методи класу працюють як належиться.

```
@SpringBootTest
public class GoalServiceTest {
    @Autowired
    private GoalService goalService;

    @Autowired
    private UserService userService;

    @Autowired
    private GoalRepository goalRepository;

    @Test
    void addGoalTest() {
        goalRepository.deleteAll();

        goalService.addGoal(new Goal("Other goal", "2020-30-12", "PUBLIC",
"MEDIUM"));

        assertEquals(1, goalRepository.findAll().size());
        assertEquals(1, goalRepository.findByStatus("PUBLIC").size());
    }

    @Test
    void getAllGoalsTest() {
        goalRepository.deleteAll();

        goalService.addGoal(new Goal("New goal", "2020-30-12", "PUBLIC",
"MEDIUM"));
        goalService.addGoal(new Goal("Other goal", "2020-28-12", "PRIVATE",
"HIGH"));

        List<Goal> allGoals = goalService.getAllGoals();

        assertEquals(2, allGoals.size());
    }
```

```
    @Test
    void getGoalByIdTest() {
        goalRepository.deleteAll();

        Goal newGoal = new Goal("New goal", "2020-30-12", "PUBLIC", "MEDIUM");
        newGoal.setId(1);
        goalService.addGoal(newGoal);

        assertEquals(newGoal, goalService.getGoalById(newGoal.getId()));
    }

    @Test
    void getGoalsByStatusTest() {
        goalRepository.deleteAll();

        goalService.addGoal(new Goal("First goal", "2020-30-12", "PUBLIC",
"MEDIUM"));
        goalService.addGoal(new Goal("Second goal", "2020-30-12", "PUBLIC",
"MEDIUM"));
        goalService.addGoal(new Goal("Third goal", "2020-30-12", "PRIVATE",
"MEDIUM"));

        assertEquals(1, goalService.getGoalsByStatus("PRIVATE").size());
        assertEquals(2, goalService.getGoalsByStatus("PUBLIC").size());
    }

    @Test
    void removeGoal() {
        goalRepository.deleteAll();

        Goal firstGoal = new Goal("First goal", "2020-30-12", "PUBLIC",
"MEDIUM");
        Goal secondGoal = new Goal("Second goal", "2020-30-12", "PUBLIC",
"MEDIUM");

        goalService.addGoal(firstGoal);
        goalService.addGoal(secondGoal);

        assertEquals(2, goalService.getAllGoals().size());
        goalService.removeGoal(firstGoal);
        assertEquals(1, goalService.getAllGoals().size());
    }

    @Test
    void updateUserGoals() {
        goalRepository.deleteAll();

        Goal firstGoal = new Goal("First goal", "2020-30-12", "PUBLIC",
"MEDIUM");
        Goal secondGoal = new Goal("Second goal", "2020-30-12", "PUBLIC",
"MEDIUM");

        User user = new User("user.email@ukr.net", "Joe", "Doe", "joe.d_12");
        userService.addUser(user);

        firstGoal.setUser(user);

        goalRepository.save(firstGoal);
        goalRepository.save(secondGoal);

        assertEquals(1, goalService.getUserGoals(user).size());
    }
}
```

Лістинг 4.1 – Unit-тести

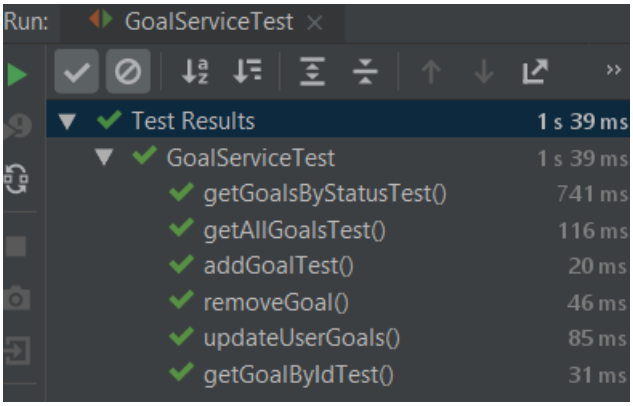


Рисунок 5.13 – Результати тестування

Всі тести закінчились успішно. Отже, функціонал реалізовано належним чином.

3. Рефакторинг

Проведемо рефакторинг наявного коду програми щоб збільшити ясність кодової бази.

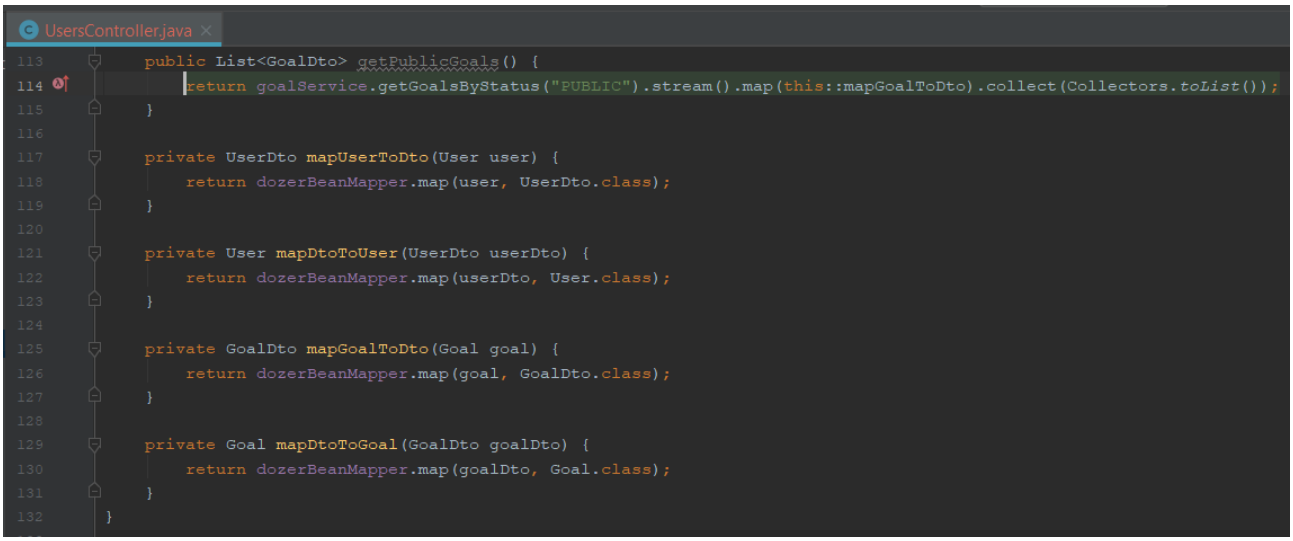


Рисунок 5.14 – “Зайві” методи

На рисунку 5.14 приведені методи класу «UsersControlles», який відповідає за обробку запитів на URL REST API. Очевидно, що вказані функції не є спільними за призначенням даного класу. Доцільно вивести їх в окремий клас, пов’язаний із мапінгом класів.

```java
@Service
public class MappingService {
    @Autowired
    private static DozerBeanMapper dozerBeanMapper;

    public static UserDto mapUserToDto(User user) {
        return dozerBeanMapper.map(user, UserDto.class);
    }

    public static User mapDtoToUser(UserDto userDto) {
        return dozerBeanMapper.map(userDto, User.class);
    }

    public static GoalDto mapGoalToDto(Goal goal) {
        return dozerBeanMapper.map(goal, GoalDto.class);
    }

    public static Goal mapDtoToGoal(GoalDto goalDto) {
        return dozerBeanMapper.map(goalDto, Goal.class);
    }
}
```

Рисунок 5.15 – Виділення класу

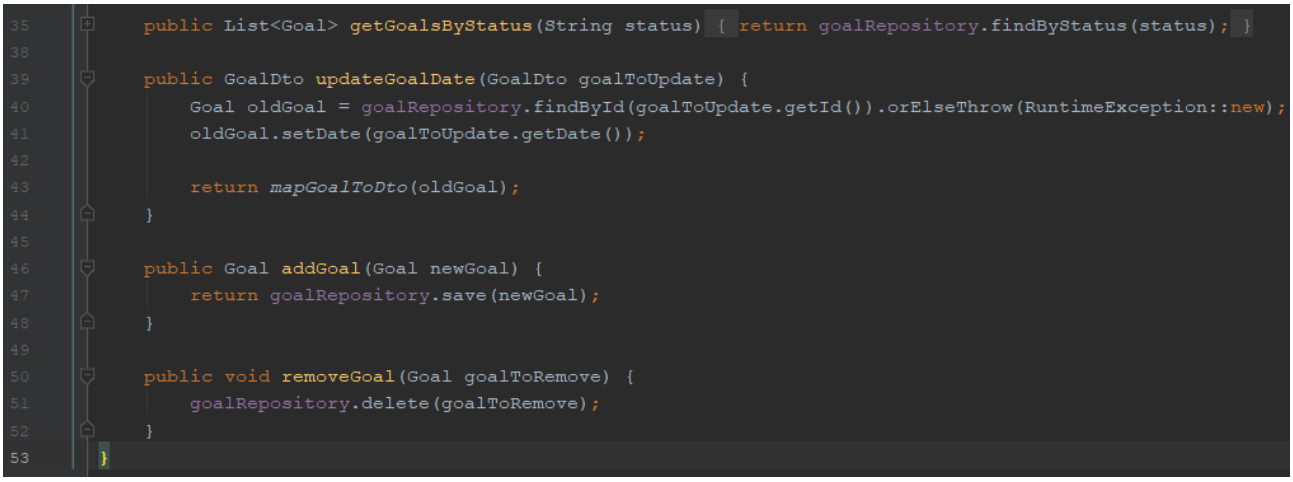
```java
@DeleteMapping("/{id}")
@ResponseBody
public UserDto deleteUserById(@PathVariable int id) {
    User userToDelete = userService.getUserById(id);
    userService.removeUser(userToDelete);

    return mapUserToDto(userToDelete);
}
```

Рисунок 5.16 – Нечіткі назви змінних

На рисунку 5.16 приведено приклад незмістовно названих змінних, із назви ких не є ясно, що вони означають. Проведемо рефакторинг:

```java
@DeleteMapping("/{userId}")
@ResponseBody
public UserDto deleteUserById(@PathVariable int userId) {
    User userToDelete = userService.getUserById(userId);
    userService.removeUser(userToDelete);

    return mapUserToDto(userToDelete);
}
```

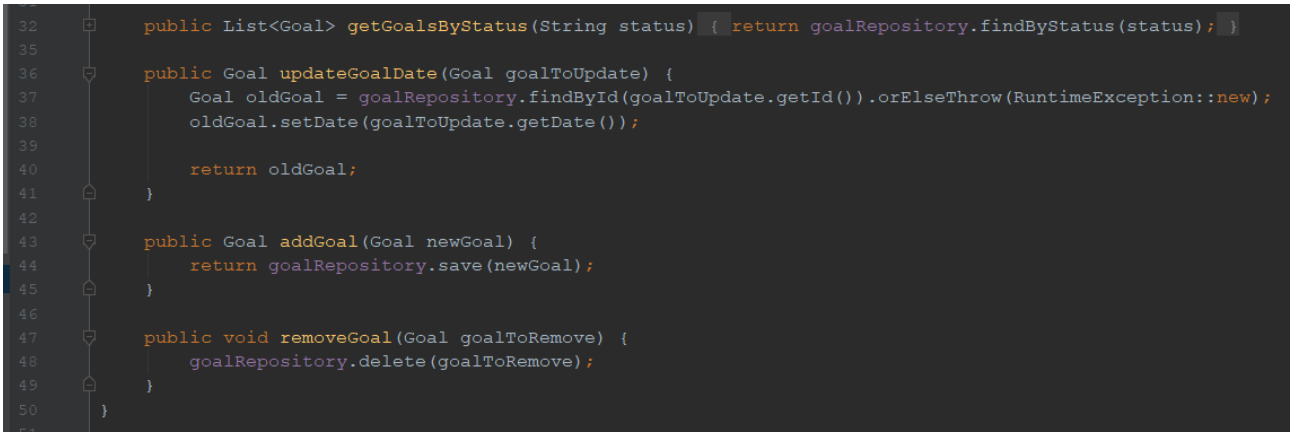
Рисунок 5.17 – Зміна назви змінної



```
    public List<Goal> getGoalsByStatus(String status) { return goalRepository.findByStatus(status); }

    public GoalDto updateGoalDate(GoalDto goalToUpdate) {
        Goal oldGoal = goalRepository.findById(goalToUpdate.getId()).orElseThrow(RuntimeException::new);
        oldGoal.setDate(goalToUpdate.getDate());

        return mapGoalToDto(oldGoal);
    }

    public Goal addGoal(Goal newGoal) {
        return goalRepository.save(newGoal);
    }

    public void removeGoal(Goal goalToRemove) {
        goalRepository.delete(goalToRemove);
    }
}
```

Рисунок 5.18 – Нелогічна сигнатура методу

В методі UpdateGoalDate, вхідний та вихідний параметри не співпадають із загальною тенднцією в классі. Виведемо мапінг на рівень вище, привівши сигнатуру до стандартного вигляду.



```
    public List<Goal> getGoalsByStatus(String status) { return goalRepository.findByStatus(status); }

    public Goal updateGoalDate(Goal goalToUpdate) {
        Goal oldGoal = goalRepository.findById(goalToUpdate.getId()).orElseThrow(RuntimeException::new);
        oldGoal.setDate(goalToUpdate.getDate());

        return oldGoal;
    }

    public Goal addGoal(Goal newGoal) {
        return goalRepository.save(newGoal);
    }

    public void removeGoal(Goal goalToRemove) {
        goalRepository.delete(goalToRemove);
    }
}
```

Рисунок 5.19 – Виправлена сигнатура методу

Висновки: в процесі виконання пунктів роботи, було успішно застосовано підхід TDD для створення класу програми. Відмічаємо, що такий підхід дозволив з першого разу реалізувати задуману функціональність, до того ж по завершенню реалізації в програмі вже були наявні готові тести, які залишаться і на майбутнє і не дозволять допустити помилкову втрату функціоналу.
Тестами було покрито і інші ділянки коду, що забезпечує його надійність та дає змогу переконатися у справному функціонуванні ще до запуску додатку.
Вкінці проведено рефакторинг коду. Виділено клас, змінено назви параметрів та сигнатру методу. Разом це дало позитивний ефект з точки зору читабельності

коду, разом із тим не змінивши функіональність (відповідно, ризиків порушити функції програми немає).