Zawansowane Języki Programowania

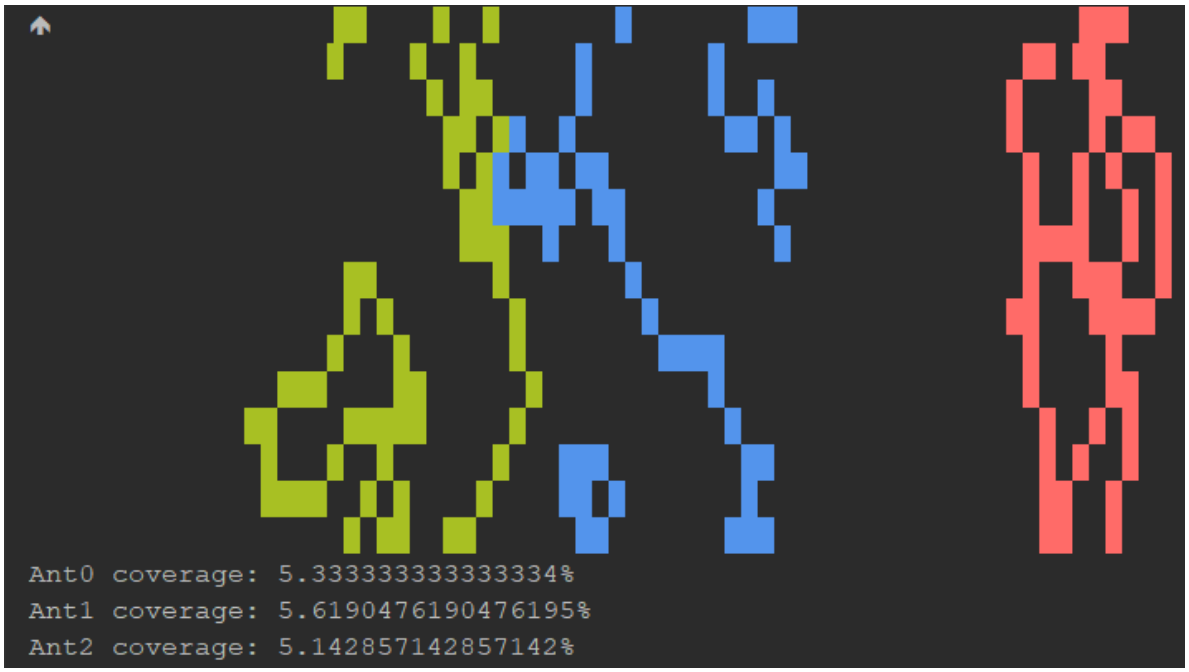
Projekt

refaktoryzacja kodu

Aleksander Kosma | 238193
Dawid Ćwik | 238137
15.01.2019.r

1. Opis projektu

Wykonany przez nas projekt był rozbudowaną wersją symulacji mrówek. W naszej wersji istnieje możliwość wybrania ilości mrówek biorącej udział w zdobywaniu jak największej ilości powierzchni. Procentowy udział każdej z mrówek jest wyświetlany wraz z mapą. Na mapie każda mrówka ma swój wlasny kolor, co pozwala odróżniać je od siebie. Poniżej screen z gry:



Problem refaktoryzacji kodu zaczęliśmy od stworzenia projektu, który jak najniższym kosztem czasowym spełniałby wymagania postawione na początku. Proces ten zabrał jeden wieczór kodowania. Miało to na celu wykonanie kodu o niskiej jakości, który byłby idealny do rozpoczęcia procesu refaktoringu.

2. Wykorzystane narzedzia / technologie

By ulatwić sobie szybką diagnoze i lokalizacje błędów, ostrzeżeń czy *smell codes*, wykorzystaliśmy następujące wtyczki, możliwe do wykorzystania w IDE Intellij Community:

- **SonarLint**
- **CheckStyle – IDEA**
- **QAPlugin**

Dodatkowo korzystaliśmy z:

- **serwisu BetterCodeHub**
- **książki Clean Code by Robert C. Martin**

3. Kroki refaktoryzacji kodu

Poniżej będą prezentowane kolejne kroki głównych przemian w kodzie:

- squid : S00115

Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that all constant names match a provided regular expression.

The following code snippet illustrates this rule when the regular expression value is "^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$":

```
private static final int x = 30;
private static final int y = 60;
private static final int X = 30;
private static final int Y = 60;
```

- squid : S1125

Remove literal boolean values from conditional expressions to improve readability. Anything that can be tested for equality with a boolean value must itself be a boolean value, and boolean values can be tested atomicly.

```
while (moved == false) {
while (!moved) {
```

- squid:S1659

Multiple variables should not be declared on the same line.

```
boolean possible0 = false, possible1 = false, possible2 = false, possible3 = false, possible4 =

boolean possible0 = false;
boolean possible1 = false;
boolean possible2 = false;
boolean possible3 = false;
boolean possible4 = false;
boolean possible5 = false;
boolean possible6 = false;
boolean possible7 = false;
```

- squid:S2189

An infinite loop is one that will never end while the program is running, i.e., you have to kill the program to get out of the loop. Whether it is by meeting the loop's end condition or via a break, every loop should have an end condition.

```
while (true) {
double sumOfAverages = 0.0;

while (sumOfAverages != 100.0) {
```

- squid:S3503

A magic number is a number that comes out of nowhere, and is directly used in a statement. Magic numbers are often used, for instance to limit the number of iterations of a loops, to test the value of a property, etc.

Using magic numbers may seem obvious and straightforward when you're writing a piece of code, but they are much less obvious and straightforward at debugging time.

Przykład 1

```
if (lastMove[i] == 0) {
    possibilities[7] = true;
    possibilities[0] = true;
    possibilities[1] = true;
if (lastMove[i] == MOVE_FORWARD) {
    possibilities[MOVE_FORWARD_LEFT] = true;
    possibilities[MOVE_FORWARD] = true;
    possibilities[MOVE_FORWARD_RIGHT] = true;
```

Przykład 2

```
while (sumOfAverages != 100.0) {
    Thread.sleep(200);
while (sumOfAverages != MAX_SUM_OF_COVERAGE_AVG) {
    Thread.sleep(THREAD_SLEEP_TIME);
```

Przykład 3

```
private static final int x = 30;
private static final int y = 60;

public static void main(String[] args) throws IOException, InterruptedException {
    List<String> colors = new ArrayList<>();
    colors.add(ANSI_RED_BACKGROUND);
    colors.add(ANSI_GREEN_BACKGROUND);
    colors.add(ANSI_YELLOW_BACKGROUND);
    colors.add(ANSI_BLUE_BACKGROUND);
    colors.add(ANSI_PURPLE_BACKGROUND);
    int x = 30;
    int y = 60;
```

- Nazwy metod

Zmiana nazw metod na odpowiadające ich roli w kodzie. Metoda odpowiedzialna za wypelnianie pustej tablicy spacjami zdecydowanie lepiej brzmi w poprawionej wersji:

```
setSpacesInTable(table);
fillArrayWithBlankSpaces(table);
changePosition(possibilities, ants, lastMove, i);
setAntsPosition(ants, i);
makeMove(possibilities, ants, lastMove, i);
setCorrectPositionIfAntIsOutOfBound(ants, i);
```

- Zmienne globalne

Skrócenie metody przy wykorzystaniu globalnych zmiennych. Przez przypadek przesylano parametry x i y, które już były dostępne jako wartości globalne.

```
private static double countAvgCoverage(int placeCovered, int x, int y) {
    return (double) placeCovered / (x * y) * PERCENTAGE_MULTIPLIER;
private static double countAvgCoverage(int placeCovered) {
    return (double) placeCovered / (X * Y) * PERCENTAGE_MULTIPLIER;
```

- Pola klasy

Zmienne które wykorzystywane są w większości metod zostały zmienione na pola przypisane do klasy odpowiedzialnej za przepływ informacji w grze. Dzięki temu nie ma potrzeby przesyłania wielu parametrów w metodach.

```
printMapOnConsole(table, colors, coverage);
printMapOnConsole();
```

- Obiektowość

Utworzenie klasy odpowiedzialnej za uruchamianie i przepływ informacji w grze. Uzyskano dzięki temu wyodrębnienie samej gry od procesu uruchomienia programu jak i podział odpowiedzialności.

```
public class Game {
    private static final String ANSI_RESET = "\u001B[0m";
```

```
public static void main(String[] args) throws InterruptedException, IOException {

    Game game = new Game(NUMBER_OF_ANTS);
    game.run();
}
```

Utworzenie klasy, która reprezentowała mrówkę, tworząc spójną całość i nierozerwalność zmiennych x oraz y kluczowych do zdefiniowania obiektu mrówka.

```
public class Ant {

    int positionX;
    int positionY;

    public Ant() {
    }
```

```
List<Ant> ants = new ArrayList<>();
ants.add(new int[]{newX, newY});
ants.add(new Ant(newX, newY));
```

4. Zasady SOLID:

S - single responsibility principle – przy refaktorze klasy Main, która na początku zawierała cały kod kierowaliśmy się tą zasadą I stworzyliśmy osobne klasy które dzielą funkcje programu.

O – open/close principle - Ponownie utworzenie odrębnej klasy Game, która przy konstukcji przyjmuje jako parametr ilość mrówek biorących udział w grze odnosi się do tej zasady. Rownież dzięki wyodrębnieniu metod

spelniających daną funkcjonalność jak np. sprawdzenie możliwego ruchu czy obliczenie średniej zajmowanego pola przez mrówki.

L – Liskov substitution principle – w naszym projekcie brak wskazników jak I rownież przykładu dziedziczenia. Jednak dzięki zachowaniu nazewnictwa, prostej struktury klasy oraz jej funkcjonalnosci dziedziczenie po istniejacych klasa jest możliwe.

I - Interface segregation principle – W naszym projekcie również brakuje interfejsów. Jednak zasada ta ogólniej mówi o dostarczeniu wszyskich możliwych funkcjonalności oraz możliwosci dostępu do nich jednostkowo, aniżeli stworzeniu jednej rzeczy generalnej. Jako przykład może posłużyć znowu klasa Main zrefaktoryzwana na mniejsze, pelniące swoje funkcjonalnosci jednostkowo.

D - Dependency inversion principle – zasada odnosząca się do budowania struktury projektu. Mówi o braku odpowiedzialności pomiędzy modułami wysokiego i niskiego poziomu. Inspiracja do powstania projektu Spring dla javy. Nasz projekt uznaliśmy za zbyt mały (nasz projekt realizuje tylko jedna metodę jaką jest run()) i skupiliśmy się na zadaniu refaktoringu złego, niezrozumiałego kodu.

5. Testy

Ze względu na potrzeby przetestowania niektórych metod, zdecydowaliśmy się na przeniesienie ich do oddzielnej klasy *Utils*. Zawarliśmy w niej statyczne metody, które są wygodne i proste do przetestowania, jednocześnie mogliśmy sprawdzić działanie części kodu. Poniżej fragmenty niektórych testów:

```
@Test
public void checkUtilsCountAvgCoverage() {
    char[][] table = new char[10][10];
    double avg = Utils.countAvgCoverage( placeCovered: 10, table);
    assertEquals(avg,  actual: 10.0,  delta: 0.2);
}

@Test
public void checkUtilsCountAvgCoverageSecond() {
    char[][] table = new char[100][100];
    double avg = Utils.countAvgCoverage( placeCovered: 10, table);
    assertEquals(avg,  actual: 0.1,  delta: 0.2);
}

@Test
public void checkFillArrayWithBlankSpaces() {
    boolean isTrue = true;
    char[][] checkingTable = Utils.fillArrayWithBlankSpaces(new char[10][10]);

    for (int i = 0; i < checkingTable.length; i++) {
        for (int j = 0; j < checkingTable[0].length; j++) {
            if (checkingTable[i][j] != ' ') {
                isTrue = false;
            }
        }
    }

    assertTrue(isTrue);
}

@Test
public void checkFillArrayWithBlankSpacesWithAddedSign() {
    boolean isTrue = true;
    char[][] checkingTable = new char[10][10];
    checkingTable[1][1] = 'x';
    checkingTable = Utils.fillArrayWithBlankSpaces(new char[10][10]);
    for (int i = 0; i < checkingTable.length; i++) {
        for (int j = 0; j < checkingTable[0].length; j++) {
            if (checkingTable[i][j] != ' ') {
```

6. Podsumowanie

Dzięki projektowi w którym skupialiśmy się na poprawie kodu w oparciu o clean code nauczyliśmy się wielu ciekawych narzędzi jak i idącej za tym wiedzy. Niektóre ostrzeżenia komunikowane przez narzedzia do refactoringu nie były dla nas oczywiste. Samo zagadnienie pisania czystego kodu okazało się bardzo rozległe i pełne ciekawych informacji. Bez wątpienia wyciągnęliśmy z tego wiele doświadczenia które posłuży nam w przyszłości.