

Studencka Pracownia Testowania Gier
Instytut Informatyki Uniwersytetu Wrocławskiego

Artur Jankowski, Paweł Sikora

Trzeci zestaw zadań

Wrocław, 12 kwietnia 2021

Wersja 0.3

Data	Numer wersji	Opis	Autor
2021-04-07	0.1	Utworzenie wstępnej wersji Dokumentu	Artur Jankowski, Paweł Sikora
2021-04-09	0.2	Dodanie nowej zawartości	Paweł Sikora
2021-04-12	0.3	Korekta dokumentu	Artur Jankowski

Spis treści

1. Użyte oprogramowanie.....	3
2. Przetestowane programy.....	4
2.1 City Builder.....	4
2.1.1 Opis programu.....	4
2.1.2 Wygenerowany raport.....	5
2.2 Peg Solitaire.....	5
2.2.1 Opis programu.....	5
2.2.2 Wygenerowany raport.....	6
3. Opis metryk.....	10
3.1 Złożoność cyklomatyczna (McCabe's Cyclomatic Complexity(MVG)).....	10
3.2 Złożoność kognitywna.....	11
3.3 Code Smells.....	12
3.4 Technical Debt.....	12

1. Użyte oprogramowanie

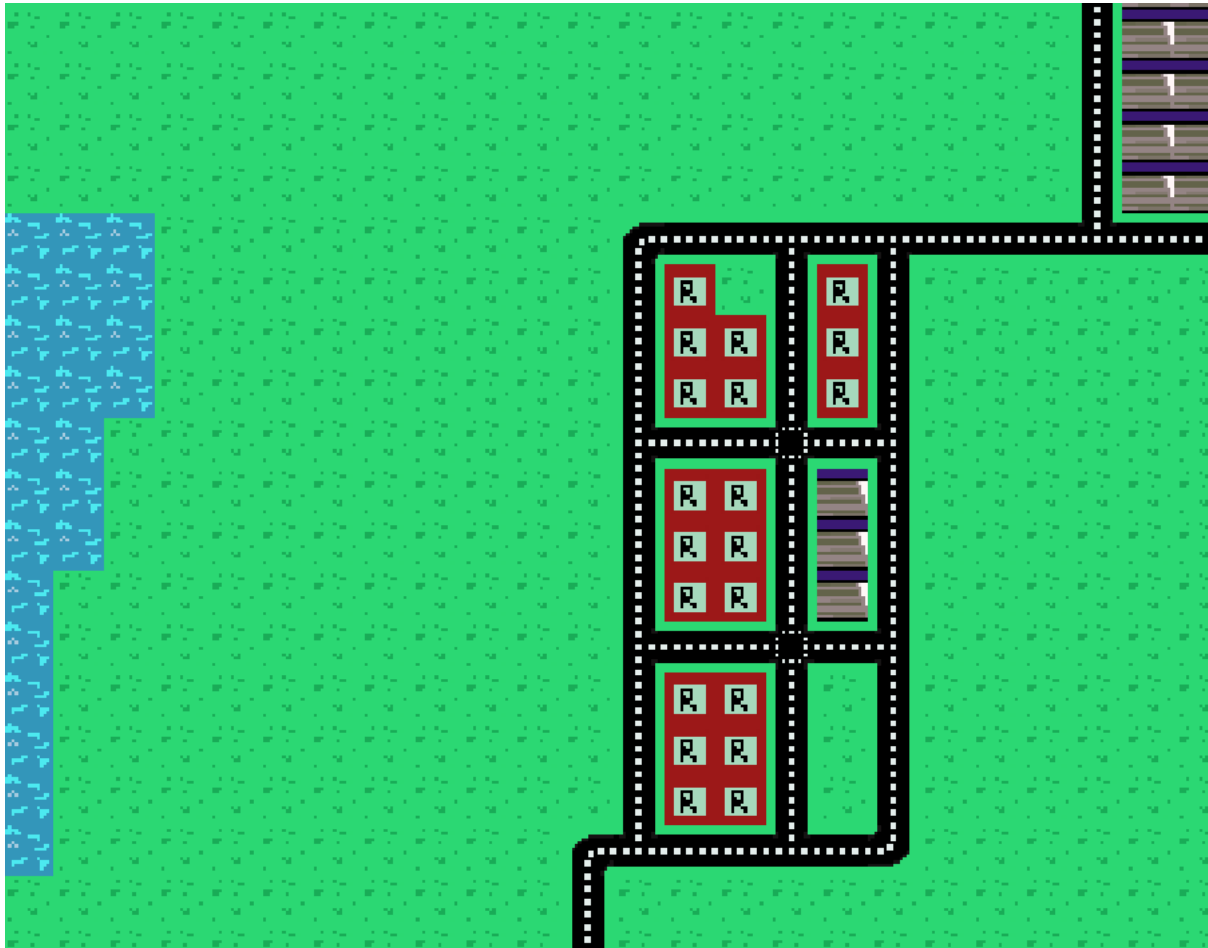
W celu wykonania zadań zapoznaliśmy się z oprogramowaniem:

- **CCCC** (<http://sourceforge.net/projects/cccc/>): statyczna analiza kodu napisanego w C, C++ lub Javie. Generuje raport, który zawiera różne metryki dotyczące np. linii kodu, ilości modułów, komentarzy, drzewa dziedziczenia itp. Nie jest już rozwijany przez oryginalnego autora, ale jest na licencji GPL.
- **SonarQube** (<https://www.sonarqube.org/>): ciągle rozwijana platforma open-source do kontroli jakości kodu, można integrować ją z wieloma innymi platformami łańcucha CI/CD. Ma wsparcie dla wielu języków. Oferuje wykrywanie błędów, luk bezpieczeństwa, ocenę jakości kodu, metryki pokrycia kodu, reużywalności, złożoności itp.

2. Przetestowane programy

2.1 City Builder

2.1.1 Opis programu



Zdjęcie 1: Okno programu

Programem CCCC przetestowaliśmy napisaną w C++ na potrzeby kursu WDPC grę wykorzystującą bibliotekę SFML.

Liczba linii kodu: 1062

2.1.2 Wygenerowany raport

Relationships which imply that changes to the client must be accompanied by the supplier's additional changes:

Metric	Tag	Overall	Per Module
Number of modules	NOM	32	
Lines of Code	LOC	1062	33.187
McCabe's Cyclomatic Number	MVG	164	5.125
Lines of Comment	COM	144	4.500
LOC/COM	L_C	7.375	
MVG/COM	M_C	1.139	
Information Flow measure (inclusive)	IF4	134	4.187
Information Flow measure (visible)	IF4v	134	4.187
Information Flow measure (concrete)	IF4c	29	0.906
Lines of Code rejected by parser	REJ	26	

Procedural Metrics Summary

For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question

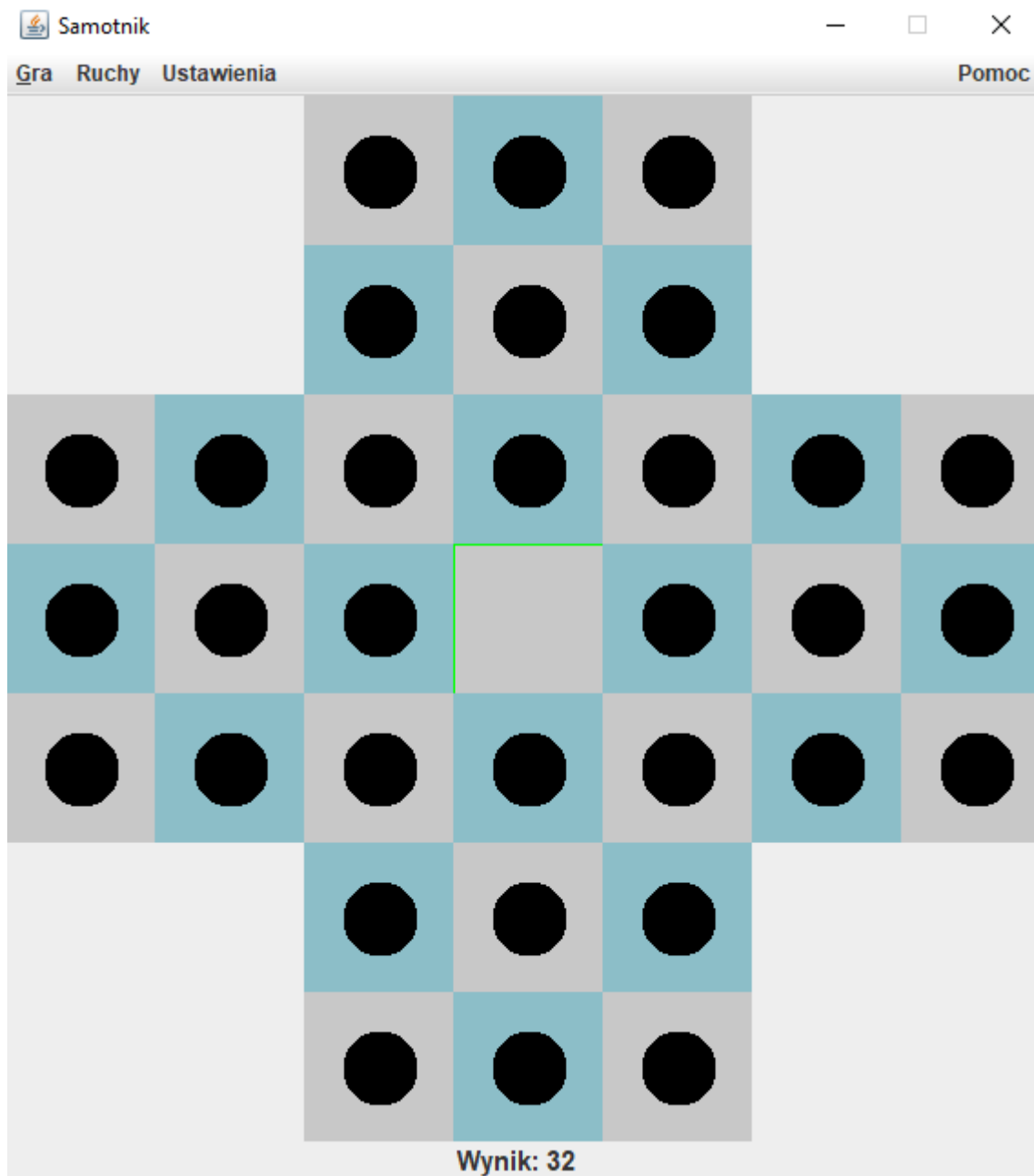
Module Name	LOC	MVG	COM	L_C	M_C
AnimationComponent	44	2	21	2.095	-----
Button	78	4	5	15.600	-----
Clock	0	0	0	-----	-----
Color	0	0	0	-----	-----
DropDownList	23	0	0	*****	-----
Font	0	0	0	-----	-----
GameLoop	90	5	15	6.000	0.333
GuiEntry	12	0	0	-----	-----
GuiStyle	21	0	0	*****	-----
IntRect	0	0	0	-----	-----
MainGameState	235	43	23	10.217	1.870
MainMenuState	112	10	6	18.667	1.667
MediaHandler	37	4	0	*****	-----
RectangleShape	0	0	0	-----	-----
RenderWindow	0	0	0	-----	-----
Sprite	0	0	0	-----	-----
State	10	0	2	-----	-----

2.2 Peg Solitaire

2.2.1 Opis programu

Przy pomocy SonarQube wygenerowaliśmy raport do Peg Solitaire - implementacji gry planszowej wykonanej na kursie języka Java.

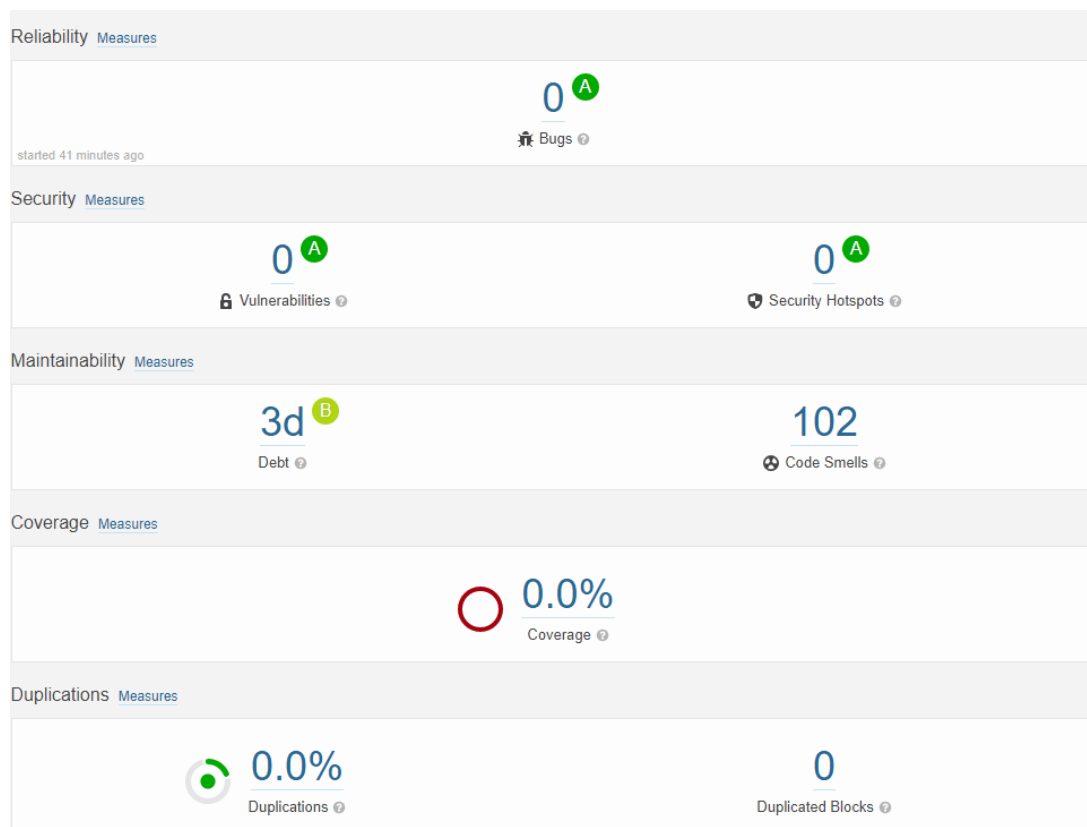
Liczba linii kodu: 516 w Java.



Zdjęcie 2: Okno programu

2.2.2 Wygenerowany raport

Program nie znalazł żadnych bugów ani słabych punktów. Oba te kryteria zostały ocenione maksymalnie.




Ciekawy jest za to wynik *code smells*, który wskazuje aż na 102 fragmenty kodu, które należałoby poprawić, lecz część z nich faktycznie nie powinna zostać zmieniona. Zwrócona jest uwaga na użycie pól “*public static*” bez użycia “*final*”, ale taki wygląd tych pól wynika z budowy aplikacji.

Code Smells 102


DrawBoard.java	5
GameData.java	41
Main.java	1
Menu.java	2
Menu_Gra.java	4
Menu_Pomoc.java	4
Menu_Ruchy.java	7
Menu_Ustawienia.java	2
MenuPopup.java	1
MoveKeyboard.java	11
MoveMouse.java	9
Screen.java	2
Solitaire.java	13

Maintainability rating został wyznaczony jedynie na podstawie wyników Code Smells, gdyż nie zostały znalezione inne błędy, które wymagałyby uwagi.

Maintainability Rating B 

 DrawBoard.java	A
 GameData.java	D
 Main.java	B
 Menu.java	A
 Menu_Gra.java	A
 Menu_Pomoc.java	A
 Menu_Ruchy.java	A
 Menu_Ustawienia.java	A
 MenuPopup.java	A
 MoveKeyboard.java	B
 MoveMouse.java	B
 Screen.java	A
 Solitaire.java	A

Program po analizie oszacował, że naprawa znalezionych błędów zajęłaby ponad 2 dni, wynik ten jest znacznie zawyżony, gdyż nawet proste zmiany, jak np. dodanie “*final*” do jednej linijki kodu, oszacowane zostało na ponad 20 minut pracy.

Overview 	
Overall	
Code Smells	102
Debt	2d 4h
Debt Ratio	7.9%
Rating	B
Effort to Reach A	7h 29min

Dla pojedynczych plików złożoność programu jest niska, przez co jest ona prosta do zrozumienia. W złożoności wyróżniają się dwa pliki: “*Solitaire*” oraz “*DrawBoard*”. Oba te pliki mają w sobie zaimplementowane ważne metody odpowiadające odpowiednio za sterowanie danymi i grą oraz za rysowanie planszy. Występuje w nich dużo if’ów i switch case’ów, przez co mają rozbudowane grafy przepływu.

Project Overview

> Reliability ⓘ

> Security ⓘ

> Security Review ⓘ

▼ Maintainability ⓘ

Overview ⓘ

Overall

Code Smells102

Debt2d 4h

Debt Ratio7.9%

Rating ⓘ

Effort to Reach A7h 29min

> Coverage

> Duplications

> Size

▼ Complexity ⓘ CYCLOMATIC COM...

Cyclomatic Complexity109

Cognitive Complexity93

> Issues

Peg-solitaire

View as

Tree

↑

↓

to select files

←

→

to navigate

13 files

Cyclomatic Complexity 109 ⓘ

DrawBoard.java

12

GameData.java

9

Main.java

1

Menu.java

1

Menu_Gra.java

4

Menu_Pomoc.java

5

Menu_Ruchy.java

9

Menu_Ustawienia.java

7

MenuPopup.java

2

MoveKeyboard.java

14

MoveMouse.java

10

Screen.java

1

Solitaire.java

34

13 of 13 shown

3. Opis metryk

3.1 Złożoność cyklomatyczna (McCabe's Cyclomatic Complexity(MVG))

Metryka oprogramowania opracowana przez Thomasa J. McCabe'a w 1976, używana do pomiaru stopnia skomplikowania programu. Podstawą do wyliczeń jest liczba dróg w schemacie blokowym danego programu, co oznacza wprost liczbę punktów decyzyjnych w tym programie.¹

Formalnie:

Liczba liniowo niezależnych ścieżek przez graf przepływu subprogramu, wyraża się wzorem:

$C = E - N + 2P^2$, gdzie:

C – złożoność cyklomatyczna,

E – liczba krawędzi grafu,

N – liczba wierzchołków grafu,

P – liczba grafów spójnych.

Wartości są interpretowane następująco:

Przedział	Interpretacja
1-10	Bardzo prosty, łatwo rozszerzalny kod
11-20	Kod dość złożony
21-50	Wysoce skomplikowany kod
Powyżej 50	Kod ekstremalnie ciężki w modyfikacji

W praktyce w CCCC liczona przy pomocy liczenia wystąpień: 'if', 'while', 'for', 'switch', 'break', '&&', '||'. Na uwagę zasługuje liczenie warunków logicznych ze względu na możliwe rozgałęzienia oraz słowa kluczowego break zamiast case, ze względu na mechanizm fall-through w C++.

1 <https://en.wikipedia.org>

2 <http://www.pzielinski.com/?p=387>

3.2 Złożoność kognitywna³

Stworzona przez SonarSource S.A. mająca niwelować wady matematycznego modelu złożoności cyklomatycznej. Celem było opisanie intuicyjnego zrozumienia trudności kodu.

Przykładowy problem:

```
int sumOfPrimes(int max) { // +1
    int total = 0;
    OUT: for (int i = 1; i <= max; ++i) { // +1
        for (int j = 2; j < i; ++j) { // +1
            if (i % j == 0) { // +1
                continue OUT;
            }
        }
        total += i;
    }
    return total;
} // Cyclomatic Complexity 4

String getWords(int number) { // +1
    switch (number) {
        case 1: // +1
            return "one";
        case 2: // +1
            return "a couple";
        case 3: // +1
            return "a few";
        default:
            return "lots";
    }
} // Cyclomatic Complexity 4
```

Dla programisty kod na lewo jest o wiele trudniejszy do zrozumienia jednak w modelu cyklometrycznym oba fragmenty mają taką samą złożoność.

A tak jest liczony przy pomocy złożoności kognitywnej:

```
int sumOfPrimes(int max) {
    int total = 0;
    OUT: for (int i = 1; i <= max; ++i) { // +1
        for (int j = 2; j < i; ++j) { // +2
            if (i % j == 0) { // +3
                continue OUT; // +1
            }
        }
        total += i;
    }
    return total;
} // Cognitive Complexity 7

String getWords(int number) {
    switch (number) { // +1
        case 1:
            return "one";
        case 2:
            return "a couple";
        case 3:
            return "a few";
        default:
            return "lots";
    }
} // Cognitive Complexity 1
```

Tego typu zmiany sprawiają, że np. wielokrotne użycie if'a wykonującego prawie to samo (które jest łatwiejsze w zrozumieniu od zagnieżdżonej pętli ze skomplikowaną logiką) będzie według tej metryki miało mniej punktów. Proste warunki nie zwiększają złożoności kognitywnej, co jest zgodne z naszym rozumieniem trudności.

3 <https://www.sonarsource.com/docs/CognitiveComplexity.pdf>

3.3 Code Smells

"Code smell, znany również jako bad smell, w kodzie programowania komputerowego, odnosi się do dowolnego objawu w kodzie źródłowym programu, który prawdopodobnie wskazuje na głębszy problem. (...) Code smells zwykle nie są błędami - nie są one technicznie niepoprawne i nie uniemożliwiają aktualnie działania programu. Zamiast tego wskazują na słabości w projekcie, które mogą spowalniać rozwój lub zwiększać ryzyko błędów, lub awarii w przyszłości. Bad code smells mogą wskazywać na czynniki, które przyczyniają się do długu technicznego." - Robert C. Martin

3.4 Technical Debt

Inaczej koszt przyszłych prac, wynikających z wykorzystywania najszybszych, ale nie optymalnych długoterminowo rozwiązań. Przedstawia szacowaną ilość czasu, którą należałoby poświęcić, aby naprawić problemy występujące w programie. Im mniejszy wynik, tym lepiej, ponieważ lepszą opcją jest poświęcić małą ilość czasu na naprawę małej ilości problemów w kodzie, niż poświęcić jednorazowo dużo czasu na naprawę dużej ilości błędów.