NARZĘDZIA PROCESU TWORZENIA OPROGRAMOWANIA

PRACOWNIA SPECJALISTYCZNA

Tomasz Kuczyński

Politechnika Białostocka

Tomasz Kuczyński

NARZĘDZIA PROCESU TWORZENIA OPROGRAMOWANIA – PRACOWNIA SPECJALISTYCZNA (1)

Politechnika Białostocka
OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2026

Recenzent:
dr inż. Kamil Żyła

Redaktor naukowy dyscypliny informatyka techniczna i telekomunikacja:
dr hab. inż. Wojciech Kwedlo, prof. PB

Korekta językowa:
Katarzyna Duniewska

Skład:
Oficyna Wydawnicza Politechniki Białostockiej

Okładka:
Marcin Dominów

Zdjęcia na okładce:
https://pixabay.com/photos/programming-computer-environment-1857236/
StockSnap: https://pixabay.com/photos/laptop-apple-keyboard-technology-2592500/

Wszystkie zrzuty ekranu umieszczone w publikacji zostały wykonane przez jej autora.

ISBN 978-83-68673-37-1 (eBook)
DOI: 10.24427/978-83-68673-37-1

Oficyna Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
e-mail: oficyna.wydawnicza@pb.edu.pl
www.pb.edu.pl

Spis treści

Wstęp

Tworzenie oprogramowania jest procesem wieloetapowym i wymagającym nie tylko solidnej wiedzy teoretycznej, ale przede wszystkim umiejętności praktycznych. Współczesna inżynieria oprogramowania [1], [2], [3], [4] bazuje na wykorzystaniu specjalistycznych narzędzi, które wspomagają każdy etap cyklu życia aplikacji – od projektowania i implementacji, przez testowanie i dokumentowanie, aż po dystrybucję oraz zarządzanie błędami. Niniejszy skrypt został przygotowany przede wszystkim jako praktyczna pomoc dla studentów uczęszczających na zajęcia z przedmiotu narzędzia procesu tworzenia oprogramowania prowadzone na Wydziale Informatyki Politechniki Białostockiej [5], [6], [7]. Zagadnienia zawarte w skrypcie ułożono w takiej samej kolejności jak tematy realizowane w ramach tego przedmiotu. Skrypt zawiera konkretne, praktyczne rozwiązania zadań, pozwalające na zgłębienie i doskonalenie umiejętności obsługi kluczowych narzędzi stosowanych w branży IT.

W odróżnieniu od typowych materiałów dydaktycznych skrypt ten nie zawiera obszernej teorii, lecz skupia się na rozwiązywaniu zadań praktycznych. Dzięki temu możliwe jest nie tylko zrozumienie, ale przede wszystkim praktyczne przetestowanie mechanizmów działania omawianych narzędzi. Taki sposób nauki pozwala na lepsze przygotowanie do pracy przy rzeczywistych projektach informatycznych, gdzie liczy się nie tylko wiedza, ale i sprawność w rozwiązywaniu problemów technicznych.

W skrypcie zostały poruszone najważniejsze aspekty pracy z narzędziami wspierającymi proces tworzenia oprogramowania, obejmujące:

1. Środowiska wytwórcze: praktyczne wprowadzenie do pracy z popularnymi środowiskami programistycznymi, takimi jak MS Visual Studio, NetBeans, Eclipse oraz MS Visual Studio Code. Zawarte zadania umożliwiają zapoznanie się z funkcjami ułatwiającymi [8] codzienną pracę deweloperską.
2. Systemy kontroli wersji: szczegółowe zadania z zakresu obsługi Git [9], jednego z najpopularniejszych systemów kontroli wersji. Przedstawione zostały podstawowe i zaawansowane operacje, takie jak tworzenie repozytorium, zarządzanie gałęziami oraz rozwiązywanie konfliktów.
3. Dynamiczne testowanie aplikacji: rozwiązywanie problemów związanych z zarządzaniem pamięcią oraz lokalizowaniem błędów w aplikacjach. Zadania z tego zakresu pozwalają na zrozumienie znaczenia dynamicznego testowania w procesie tworzenia oprogramowania.

4. Profilowanie aplikacji: praktyczne przykłady analizy czasu działania aplikacji oraz identyfikacji potencjalnych wąskich gardeł wydajnościowych. Omówione zostało, w jaki sposób można optymalizować kod na podstawie wyników profilowania.
5. Dokumentowanie kodu: zadania mające na celu rozwinięcie umiejętności tworzenia czytelnej i użytecznej dokumentacji kodu, co jest kluczowe w pracy zespołowej.
6. Zarządzanie błędami: wprowadzenie do narzędzia Bugzilla wraz z praktycznymi scenariuszami dotyczącymi zgłaszania, monitorowania i rozwiązywania błędów.
7. Dystrybucja oprogramowania: zadania obejmujące zastosowanie Dockera do tworzenia i zarządzania kontenerami aplikacyjnymi. Przedstawione zostały praktyczne aspekty konfiguracji oraz dystrybucji oprogramowania w środowisku kontenerowym.

Głównym celem niniejszego skryptu jest umożliwienie zdobycia praktycznych umiejętności, które mogą zostać wykorzystane w przyszłej pracy zawodowej. Każdy rozdział został opracowany w taki sposób, aby możliwe było samodzielne eksperymentowanie oraz rozwiązywanie problemów, co sprzyja uczeniu się przez doświadczenie.

Dzięki temu skryptowi możliwe jest nie tylko poznanie narzędzi wspierających proces tworzenia oprogramowania, ale również zrozumienie ich znaczenia w kontekście praktycznego zastosowania. Mam nadzieję, że niniejszy materiał stanie się solidnym fundamentem do dalszego rozwoju w branży IT oraz inspiracją do pogłębiania wiedzy i umiejętności w zakresie inżynierii oprogramowania.

1. Środowiska wytwórcze

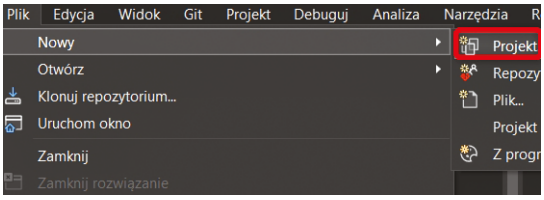
1.1. MS Visual Studio

1.1.1. Zadanie 1 – tworzenie aplikacji konsolowej

Wykonaj poniższe czynności:

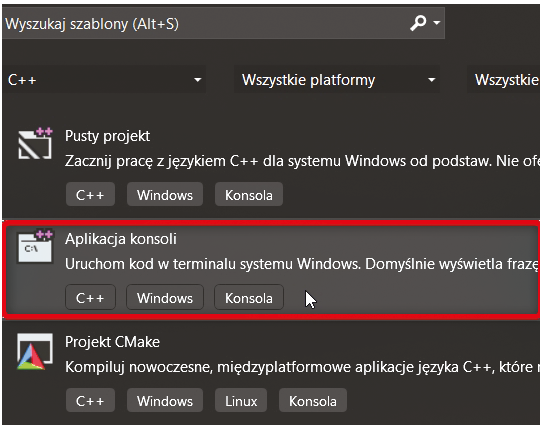
1. *Stwórz aplikację konsolową.*
2. *Napisz krótki kod, który wykorzystuje pętlę oraz instrukcję warunkową.*
3. *Sprawdź, jak działa debugger i zapoznaj się z jego funkcjami.*

Nowy projekt tworzymy, wybierając z menu: **Plik** → **Nowy** → **Projekt** (rys. 1.1).



RYSUNEK 1.1. **Tworzenie nowego projektu**

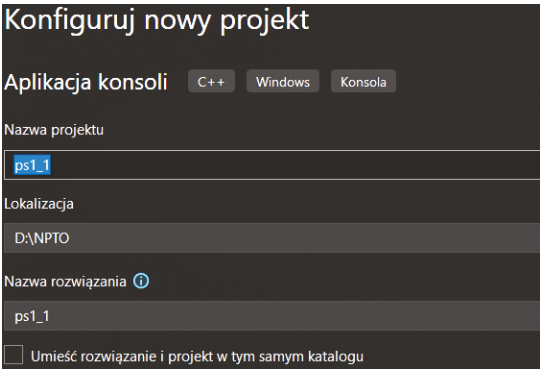
Wybieramy odpowiedni szablon dla projektu – w tym przypadku będzie to „Aplikacja konsoli". Łatwiej znajdziemy właściwy szablon, jeśli wcześniej wybierzemy potrzebny nam język programowania, czyli „C++" (rys. 1.2).



RYSUNEK 1.2. **Wybór odpowiedniego szablonu**

Klikamy przycisk „Dalej" znajdujący się w prawym dolnym rogu.

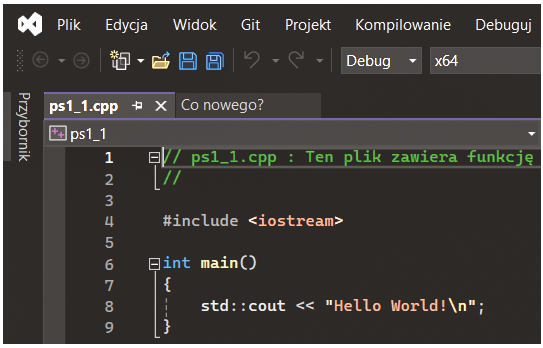
Konfigurujemy projekt, tzn. nadajemy mu nazwę oraz wskazujemy lokalizację projektu na dysku. Możemy zostawić wartości domyślne lub je zmienić (rys. 1.3).



RYSUNEK 1.3. **Konfiguracja projektu**

Klikamy przycisk „Utwórz” znajdujący się w prawym dolnym rogu.

Zostanie utworzony projekt zawierający przykładowy prosty kod programu wyświetlającego na ekranie napis „Hello World!” (rys. 1.4).



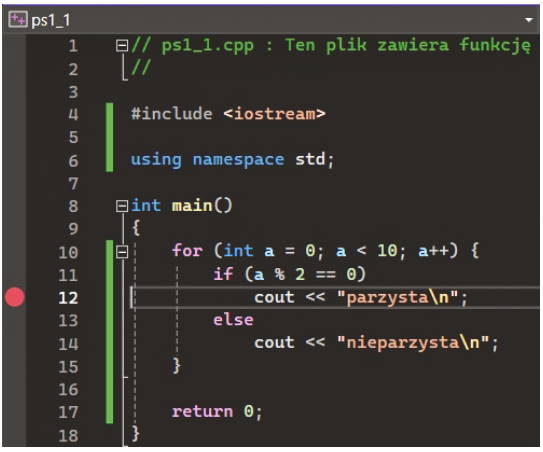
RYSUNEK 1.4. **Utworzony projekt z przykładowym kodem**

Dostosowujemy kod do naszych potrzeb, tak aby zawierał pętlę i instrukcję warunkową (listing 1.1).

LISTING 1.1. Krótki kod zawierający pętlę oraz instrukcję warunkową:

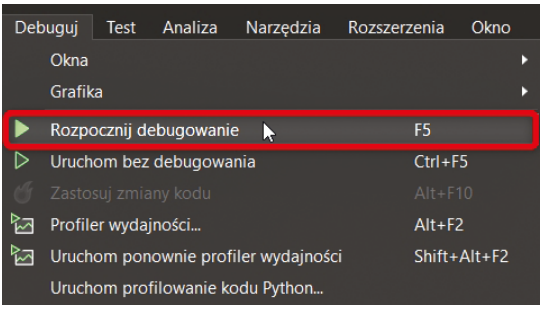
```
#include <iostream>

using namespace std;

int main()
{
     for (int a = 0; a < 10; a++) {
          if (a % 2 == 0)
               cout << „parzysta\n”;
          else
               cout << „nieparzysta\n”;
     }

     return 0;
}
```

W celu skorzystania z debugera należy wcześniej wstawić punkt przerwania (*breakpoint*). Punkt przerwania (czerwona kropka po lewej stronie) określa miejsce, w którym debuger będzie się zatrzymywał (rys. 1.5).



RYSUNEK 1.5. **Wstawianie punktu przerwania**

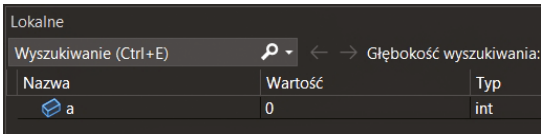
Aby rozpocząć proces debugowania, wybieramy z menu: **Debuguj** → **Rozpocznij debugowanie** (rys. 1.6).



RYSUNEK 1.6. **Rozpoczęcie procesu debugowania**

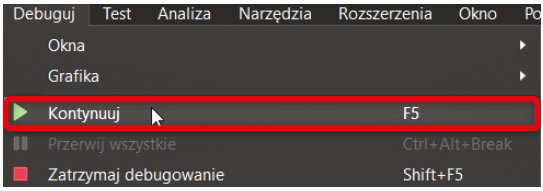
Jeśli wcześniej nie skompilowaliśmy naszego projektu, to może pojawić się okno informujące, iż projekt jest nieaktualny lub że wystąpiły błędy kompilacji. Należy wtedy podjąć odpowiednie działanie, tzn. skompilować projekt lub usunąć błędy.

W okienku na dole powinniśmy mieć podgląd na aktualne wartości zmiennych (rys. 1.7).



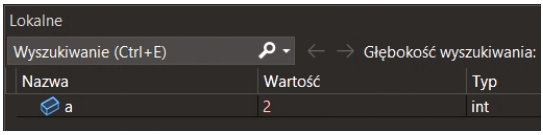
RYSUNEK 1.7. **Okno z podglądem na aktualne wartości zmiennych**

Z menu wybieramy: **Debuguj** → **Kontynuuj** (rys. 1.8).



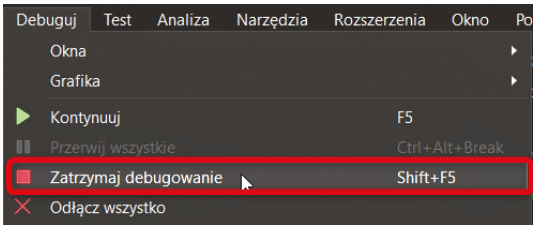
RYSUNEK 1.8. **Kontynuowanie procesu debugowania**

W okienku na dole obserwujemy kolejne wartości, jakie przyjmują nasze zmienne, dzięki czemu łatwiej możemy wyłapać błędy (rys. 1.9).



RYSUNEK 1.9. **Okno z podglądem na aktualne wartości zmiennych**

Aby zakończyć proces debugowania, wybieramy w menu: **Debuguj** → **Zatrzymaj debugowanie**. Oczywiście możemy też wielokrotnie wybierać opcję **Debuguj** → **Kontynuuj**, aż do zakończenia działania naszego programu (rys. 1.10).

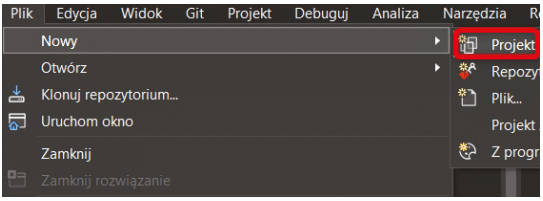


RYSUNEK 1.10. **Zatrzymanie procesu debugowania**

1.1.2. Zadanie 2 – tworzenie aplikacji graficznej WPF

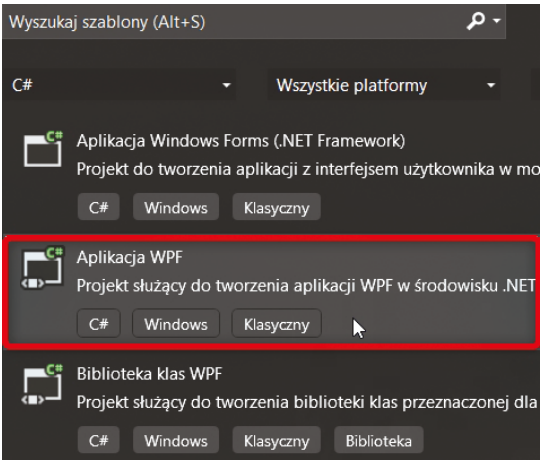
Wykonaj poniższe czynności:
1. *Stwórz aplikację graficzną WPF.*
2. *Dodaj kilka kontrolek z „Przybornika" („Toolbox").*
3. *Pozmieniaj ich właściwości poprzez okno „Właściwości".*
4. *Przejdź z warstwy prezentacji na warstwę logiki/danych.*
5. *Dodaj „ręcznie" (poprzez kod XAML) dowolną kontrolkę* [5].

Nowy projekt tworzymy, wybierając z menu: **Plik** → **Nowy** → **Projekt** (rys. 1.11).



RYSUNEK 1.11. **Tworzenie nowego projektu**

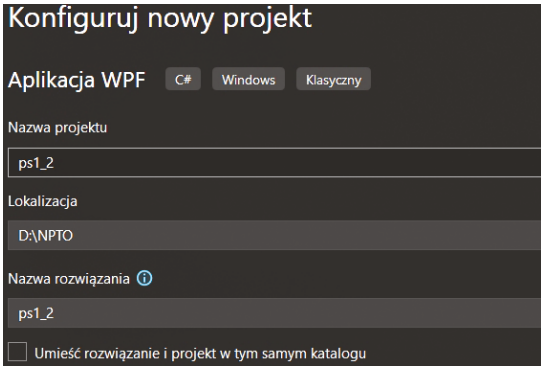
Wybieramy odpowiedni szablon dla projektu – w tym przypadku będzie to „Aplikacja WPF". Łatwiej znajdziemy właściwy szablon, jeśli wcześniej wybierzemy potrzebny nam język programowania, czyli „C#" (rys. 1.12).



RYSUNEK 1.12. **Wybór odpowiedniego szablonu**

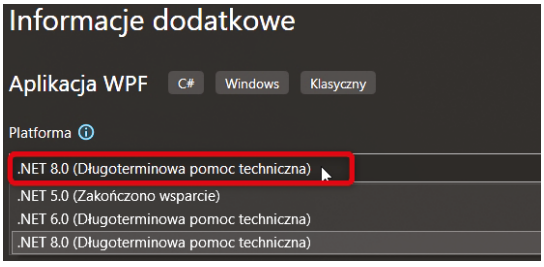
Klikamy przycisk „Dalej" znajdujący się w prawym dolnym rogu.

Konfigurujemy projekt, tzn. nadajemy mu nazwę oraz wskazujemy lokalizację projektu na dysku. Możemy zostawić wartości domyślne lub je zmienić (rys. 1.13).



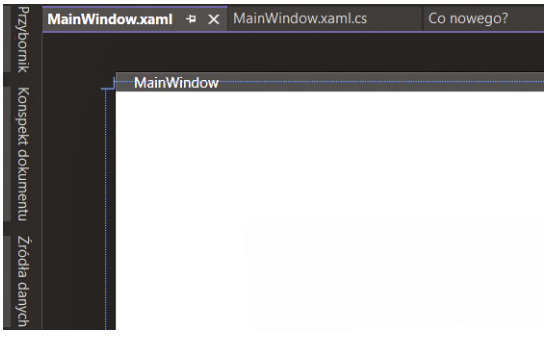
RYSUNEK 1.13. **Konfiguracja projektu**

Klikamy przycisk „Dalej" znajdujący się w prawym dolnym rogu.
Wybieramy platformę dla naszej aplikacji WPF (rys. 1.14).



RYSUNEK 1.14. **Wybór platformy**

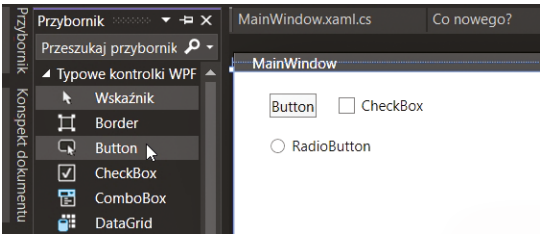
Klikamy przycisk „Utwórz" znajdujący się w prawym dolnym rogu.
Zostanie utworzony projekt zawierający puste okienko, do którego będziemy dodawali kontrolki z „Przybornika" (rys. 1.15).



RYSUNEK 1.15. **Utworzony projekt z pustym oknem**

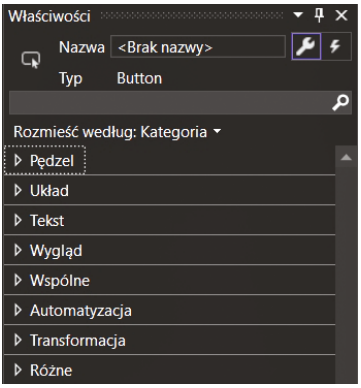
Jeśli na pasku po lewej stronie nie widzimy „Przybornika", to możemy go uwidocznić, wybierając z menu: **Widok → Przybornik** (`Ctrl + Alt + X`).

Wybieramy z „Przybornika" kilka dowolnych kontrolek i przenosimy je do okna „MainWindow". Przenoszenia kontrolek dokonujemy metodą „przeciągnij i upuść" (rys. 1.16).



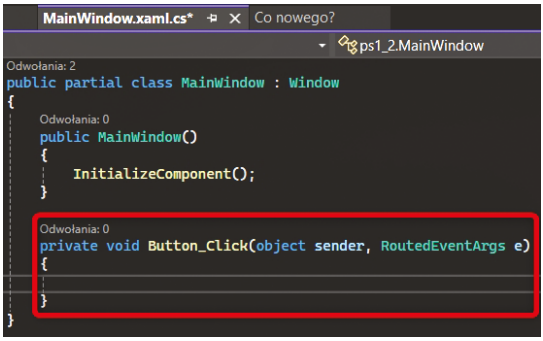
RYSUNEK 1.16. **Dodawanie kontrolek z „Przybornika"**

Klikamy w kontrolki i zmieniamy ich właściwości poprzez okno „Właściwości" (rys. 1.17). Jeśli okno „Właściwości" jest niewidoczne, to uwidaczniamy je, wybierając z menu: **Widok → Okno właściwości** (`F4`).



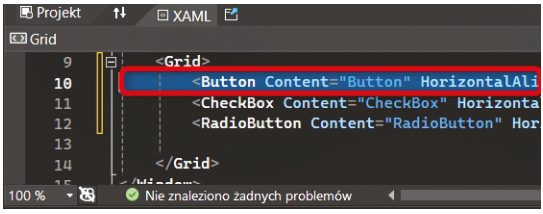
RYSUNEK 1.17. **Okno „Właściwości"**

Klikamy dwukrotnie w wybraną kontrolkę (np. przycisk) w celu przejścia z warstwy prezentacji na warstwę logiki/danych. Zostaniemy przeniesieni do okienka z kodem w języku C#. Zostanie także dodana nowa metoda, którą możemy uzupełnić odpowiednim kodem (rys. 1.18).



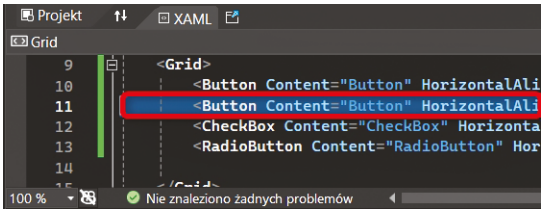
RYSUNEK 1.18. **Przejście z warstwy prezentacji na warstwę logiki/danych**

Dodajemy „ręcznie" (poprzez kod XAML) kontrolkę przycisku. W tym celu należy w okienku z kodem XAML zaznaczyć kod dotyczący wcześniej już dodanej kontrolki przycisku (rys. 1.19).



RYSUNEK 1.19. **Ręczne dodawanie kontrolki (poprzez kod XAML)**

Kopiujemy zaznaczony kod i wklejamy poniżej (rys. 1.20). Po takiej operacji należy oczywiście w nowo dodanej kontrolce zmienić właściwości na odpowiednie (np. położenie, rozmiar, kolor).



RYSUNEK 1.20. **Ręcznie dodana kontrolka przycisku**

W widoku okna „MainWindow” możemy już zauważyć ręcznie dodaną kontrolkę przycisku (rys. 1.21).

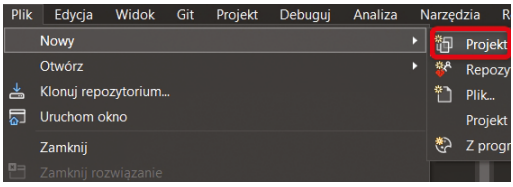
RYSUNEK 1.21. **Widok okna „MainWindow” z ręcznie dodaną kontrolką**

1.1.3. Zadanie 3 – tworzenie aplikacji internetowej

Wykonaj poniższe czynności:

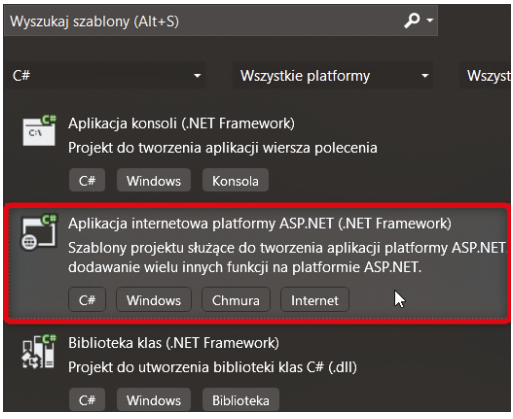
1. *Stwórz aplikację internetową.*
2. *Dodaj kilka kontrolek z „Przybornika”.*
3. *Zobacz, jak wygląda aktualna strona w podglądzie.*

Nowy projekt tworzymy, wybierając z menu: **Plik** → **Nowy** → **Projekt** (rys. 1.22).



RYSUNEK 1.22. **Tworzenie nowego projektu**

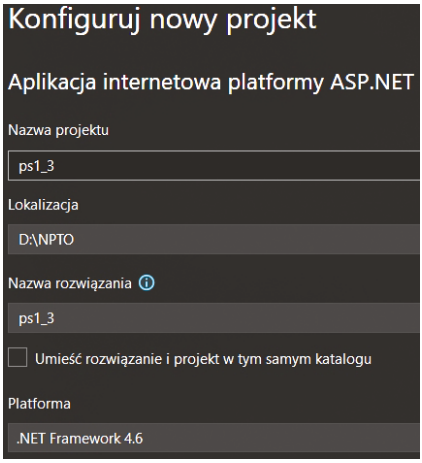
Wybieramy odpowiedni szablon dla projektu – w tym przypadku będzie to „Aplikacja internetowa platformy ASP.NET (.NET Framework)”. Łatwiej znajdziemy właściwy szablon, jeśli wcześniej wybierzemy potrzebny nam język programowania, czyli „C#” (rys. 1.23).



RYSUNEK 1.23. **Wybór odpowiedniego szablonu**

Klikamy przycisk „Dalej” znajdujący się w prawym dolnym rogu.

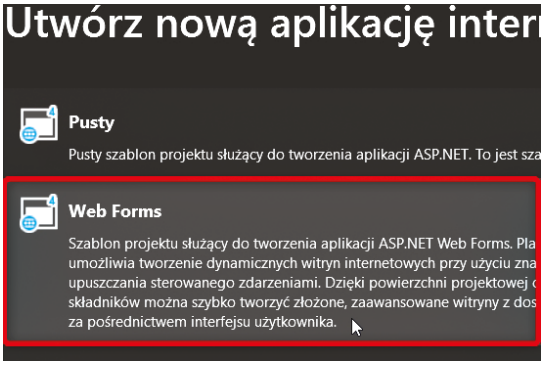
Konfigurujemy projekt, tzn. nadajemy mu nazwę oraz wskazujemy lokalizację projektu na dysku. Możemy zostawić wartości domyślne lub je zmienić (rys. 1.24).



RYSUNEK 1.24. **Konfiguracja projektu**

Klikamy przycisk „Utwórz” znajdujący się w prawym dolnym rogu.

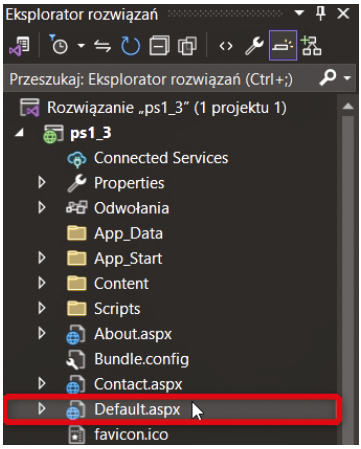
Wybieramy odpowiedni szablon projektu, tzn. „Web Froms” (rys. 1.25).



RYSUNEK 1.25. **Wybór odpowiedniego szablonu**

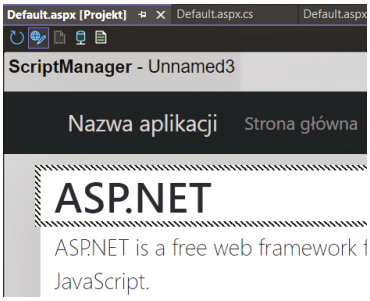
Klikamy przycisk „Utwórz” znajdujący się w prawym dolnym rogu. Zostanie utworzony nowy projekt z przykładową zawartością.

W oknie „Eksplorator rozwiązań” (**Widok → Eksplorator rozwiązań**) znajdującym się po prawej stronie wybieramy (poprzez dwukrotne kliknięcie w nazwę strony) jedną z przykładowych stron internetowych (rys. 1.26).



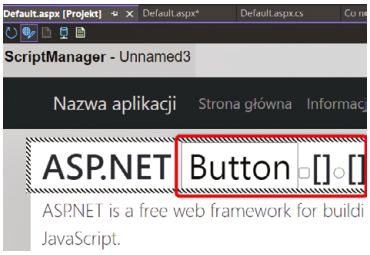
RYSUNEK 1.26. Okno „Eksplorator rozwiązań”

Zmieniamy widok tej strony z „Kod” na „Projekt”, wybierając z menu opcję: **Widok → Projektant** (rys. 1.27).



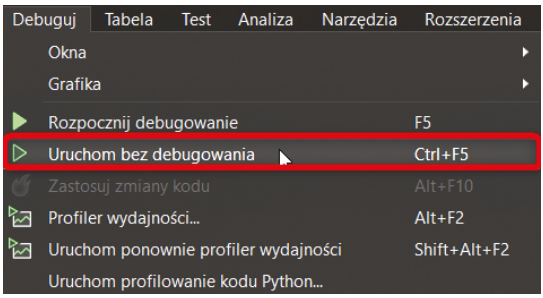
RYSUNEK 1.27. Widok projektu strony „Default”

Dodajemy kilka kontrolek z „Przybornika” (rys. 1.28), np. przycisk (*button*), pole wyboru (*checkbox*) czy przycisk radiowy (*radio button*).



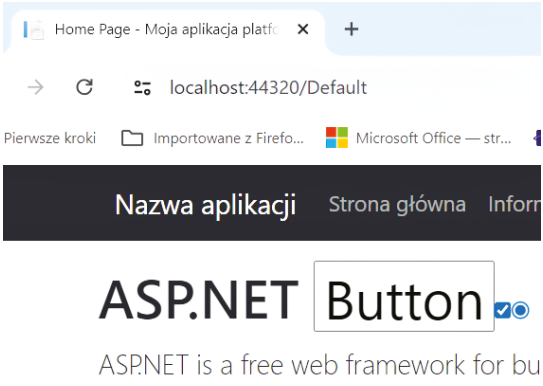
RYSUNEK 1.28. Przykładowa strona z dodanymi kontrolkami

W celu sprawdzenia podglądu bieżącej strony wybieramy z menu: **Debuguj** → **Uruchom bez debugowania** (rys. 1.29).



RYSUNEK 1.29. **Sprawdzanie podglądu bieżącej strony**

W oknie przeglądarki internetowej zostanie otwarta strona z dodanymi przez nas kontrolkami (rys. 1.30).



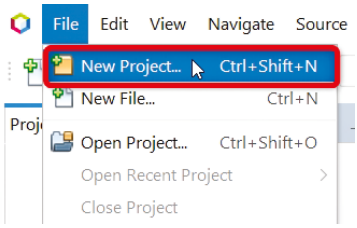
RYSUNEK 1.30. **Podgląd bieżącej strony**

1.2. NetBeans

1.2.1. Zadanie 1 – tworzenie nowego projektu

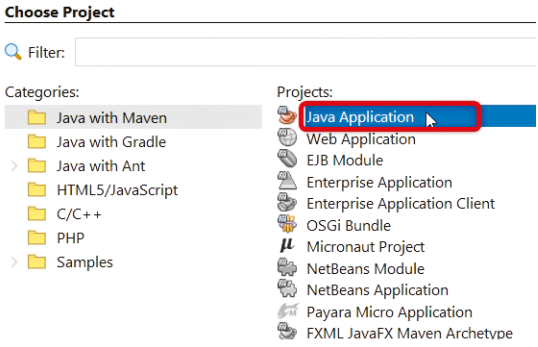
Utwórz aplikację w języku Java, która wypełni 100-elementową tablicę losowymi wartościami, posortuje elementy tablicy w kolejności malejącej, a następnie wypisze wartości najmniejszej i największej liczby na ekranie [5].

W celu utworzenia aplikacji w języku Java rozpoczynamy nowy projekt, wybierając z menu: **File** → **New Project** (rys. 1.31).



RYSUNEK 1.31. **Tworzenie nowego projektu**

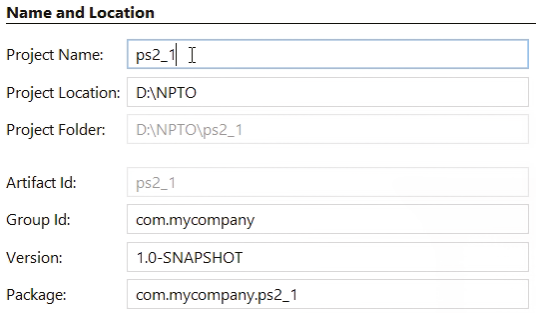
Wybieramy odpowiedni szablon projektu – kategoria: „Java with Maven", projekt: „Java Application" (rys. 1.32).



RYSUNEK 1.32. **Wybór szablonu projektu**

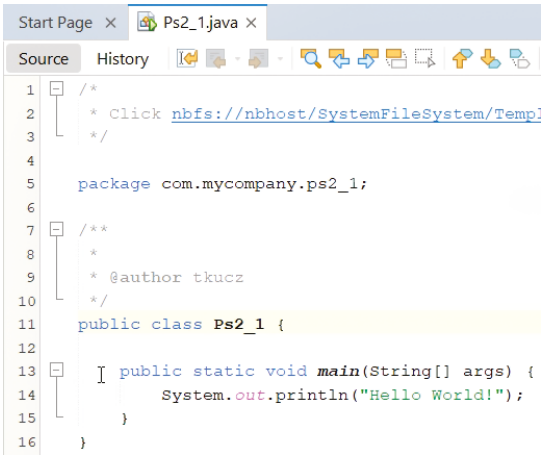
Klikamy przycisk „Next" w celu zatwierdzenia wyboru szablonu projektu.

Nadajemy nazwę dla naszego projektu oraz wskazujemy lokalizację, gdzie zostanie on zapisany na dysku. Możemy wpisać własne wartości lub zostawić domyślne (rys. 1.33).



RYSUNEK 1.33. **Wybór nazwy projektu oraz jego lokalizacji na dysku**

Klikamy przycisk „Finish” w celu zatwierdzenia nazwy projektu oraz jego lokalizacji na dysku. Zostanie utworzony nowy projekt wraz z przykładowym prostym kodem (rys. 1.34).



RYSUNEK 1.34. **Nowy projekt wraz z przykładowym kodem**

Modyfikujemy kod w taki sposób, aby nasza aplikacja spełniała wszystkie wymienione wcześniej założenia (listing 1.2).

LISTING 1.2. Przykładowy kod naszej aplikacji:

```
package com.mycompany.ps2 _ 1;
import java.util.Random;
import java.util.Arrays;
import java.util.Collections;

public class Ps2 _ 1 {
    public static void main(String[] args) {
        Integer[] t;
        Random rand = new Random();
        t = new Integer[100];

        for (int i = 0; i<100; i++)
            t[i] = rand.nextInt(1000);

        Arrays.sort(t,Collections.reverseOrder());
        System.out.println(„Najmniejsza liczba”
        + „ to: „+t[99]+” a najwieksza to: „+t[0]);
    }
}
```

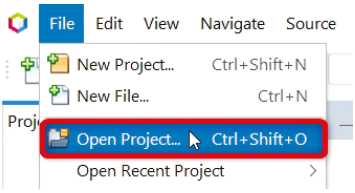
1.2.2. Zadanie 2 – generowanie dokumentacji

Wykonaj poniższe czynności:

1. *Wygeneruj dokumentację Javadoc dla aplikacji* `KalkulatorApp`.
2. *Dodaj w klasie* `Kalkulator` *metodę* `double pierwiastek(double x)` *oraz dołącz do niej komentarz zgodny z Javadoc.*
3. *Wygeneruj ponownie dokumentację* [5].

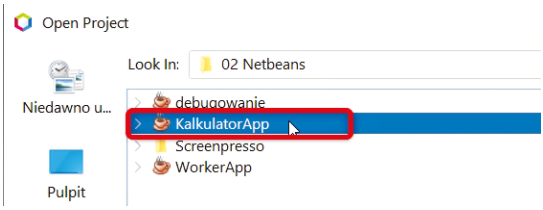
Aplikacja `KalkulatorApp` znajduje się w pliku `ps-netbeans.zip` [5], [6]. Znajdują się tam też dwie inne aplikacje, które będą wykorzystywane w podrozdziałach 1.2.3 i 1.2.4. Są to `WorkerApp` i `Debugowanie`. Możemy jednocześnie otworzyć wszystkie trzy aplikacje w środowisku NetBeans, wybierając z menu: **File → Import Project → From ZIP**. Klikamy przycisk „Browse” znajdujący się obok pola „ZIP File”, a następnie wybieramy plik `ps-netbeans.zip`.

Możemy również otwierać każdą z aplikacji pojedynczo wtedy, kiedy zajdzie taka potrzeba. W takiej sytuacji należy rozpakować archiwum ZIP. Kod wybranej aplikacji wczytujemy, wybierając z menu opcję: **File → Open Project** (rys. 1.35).



RYSUNEK 1.35. **Otwieranie wcześniej utworzonego projektu**

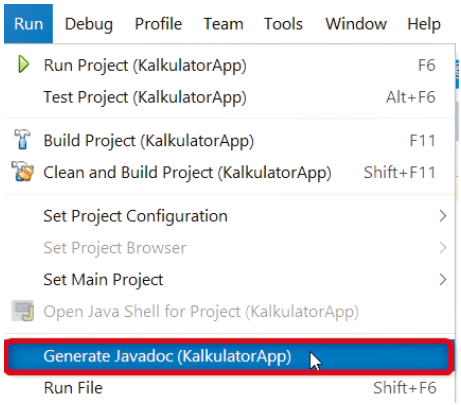
Wskazujemy lokalizację aplikacji `KalkulatorApp` na dysku (rys. 1.36).



RYSUNEK 1.36. **Wskazanie aplikacji** `KalkulatorApp`

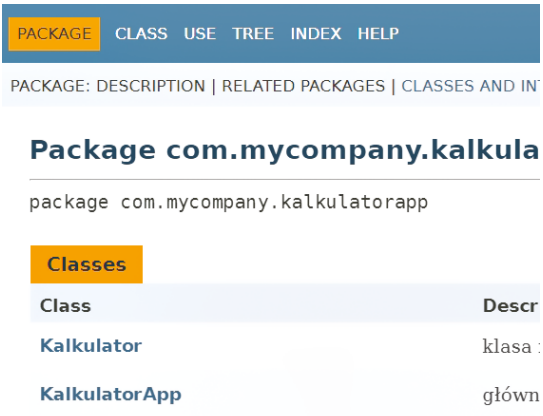
Klikamy przycisk „Open Project" w celu otwarcia wybranego projektu.

Mając już otwarty projekt, generujemy dokumentację Javadoc. W tym celu wybieramy z menu: **Run → Generate Javadoc** (rys. 1.37).



RYSUNEK 1.37. **Generowanie dokumentacji**

W katalogu, w którym znajduje się nasz projekt, zostanie wygenerowana dokumentacja Javadoc. Ścieżka to: `KalkulatorApp\target\site\apidocs\index.html` (rys. 1.38).



RYSUNEK 1.38. **Fragment wygenerowanej dokumentacji**

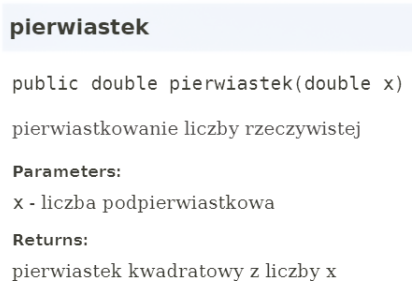
UWAGA: w przypadku wystąpienia błędów w trakcie generowania dokumentacji należy stworzyć nowy projekt składający się z dwóch plików, które będą zawierały kod aplikacji `KalkulatorApp`. W kodzie w obu plikach sprawdzamy, czy nazwa pakietu zgadza się z nazwą pakietu w okienku „Projects" znajdującym się po lewej stronie oraz czy nazwa klasy w kodzie jest taka sama jak nazwa pliku (`*java`) w okienku „Projects". Jeśli nie ma zgodności, należy to poprawić. Przydatna może być również opcja: **Refactor → Rename**. Problem ten możemy także rozwiązać, zmieniając w NetBeans wersję JDK na np. JDK 8 (1.8). Użyteczne mogą tutaj okazać się ponadto opcja **Run → Set Project Configuration → Customize → Compile → Java Platform** oraz ewentualnie **Tools → Java Platforms → Add Platform**.

Dodajemy w klasie `Kalkulator` metodę `double pierwiastek(double x)` oraz dołączamy do niej komentarz zgodny z Javadoc (rys. 1.39).

```
/**
 * pierwiastkowanie liczby rzeczywistej
 * @param x liczba podpierwiastkowa
 * @return pierwiastek kwadratowy z liczby x
 */
public double pierwiastek(double x) {
    return Math.sqrt(x);
}
```

RYSUNEK 1.39. **Metoda `pierwiastek` wraz z komentarzem Javadoc**

Generujemy ponownie dokumentację Javadoc (**Run → Generate Javadoc**) i szukamy w niej metody o nazwie `pierwiastek` (rys. 1.40).



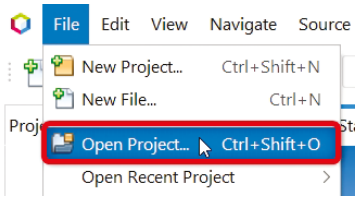
RYSUNEK 1.40. **Fragment dokumentacji zawierający opis metody** `pierwiastek`

1.2.3. Zadanie 3 – profilowanie

Korzystając z profilera, sprawdź, która operacja z klasy `WorkerBean` *w aplikacji* `Worker-App` *jest najbardziej czasochłonna* [5].

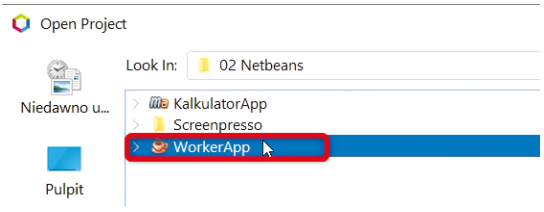
Profilowanie to proces analizowania działania programu w celu identyfikacji miejsc zużywających najwięcej zasobów, takich jak czas procesora lub pamięć.

Aplikacja `WorkerApp` znajduje się w pliku `ps-netbeans.zip` [5], [6]. Należy rozpakować kod aplikacji i wczytać go, wybierając z menu opcję: **File → Open Project** (rys. 1.41).



RYSUNEK 1.41. **Otwieranie wcześniej utworzonego projektu**

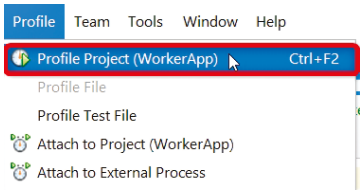
Wskazujemy lokalizację aplikacji `WorkerApp` na dysku (rys. 1.42).



RYSUNEK 1.42. **Wskazanie aplikacji** `WorkerApp`

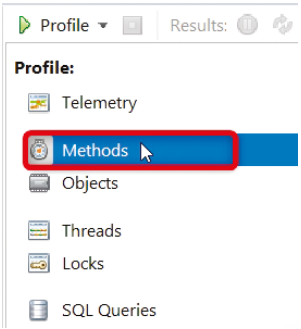
Klikamy przycisk „Open Project” w celu otwarcia wybranego projektu.

Mając już otwarty projekt, wykonujemy profilowanie. W tym celu wybieramy z menu: **Profile** → **Profile Project** (rys. 1.43).



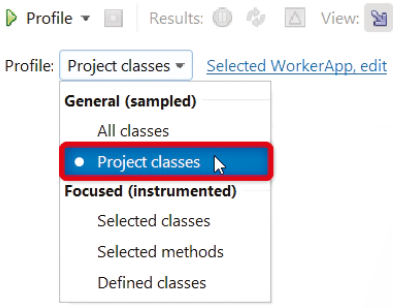
RYSUNEK 1.43. **Profilowanie**

Z rozwijanego menu wybieramy profilowanie metod: **Profile** → **Methods** (rys. 1.44).



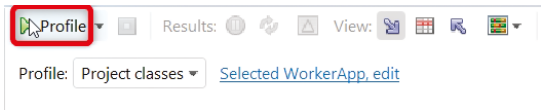
RYSUNEK 1.44. **Wybór profilowania metod**

Wybieramy opcję profilowania klas projektu: **Profile** → **Project classes** (rys. 1.45).



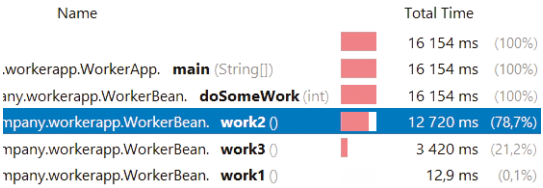
RYSUNEK 1.45. **Wybór profilowania klas projektu**

W celu rozpoczęcia procesu profilowania klikamy przycisk z zieloną strzałką i napisem „Profile” (rys. 1.46).



RYSUNEK 1.46. **Rozpoczęcie procesu profilowania**

Na podstawie otrzymanych wyników możemy stwierdzić, iż najbardziej czasochłonna jest operacja `work2` (rys. 1.47).



RYSUNEK 1.47. **Wyniki profilowania**

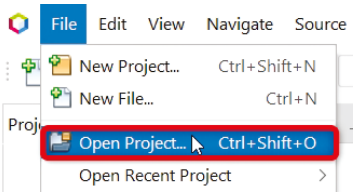
UWAGA: w przypadku wystąpienia błędów w trakcie profilowania należy stworzyć nowy projekt składający się z dwóch plików, które będą zawierały kod aplikacji `WorkerApp`.

W celu zaobserwowania w wynikach profilowania wszystkich trzech operacji może zaistnieć potrzeba zwiększenia w kodzie liczby iteracji dla każdej z trzech metod, np. z 1000 do 100 000.

1.2.4. Zadanie 4 – debugowanie

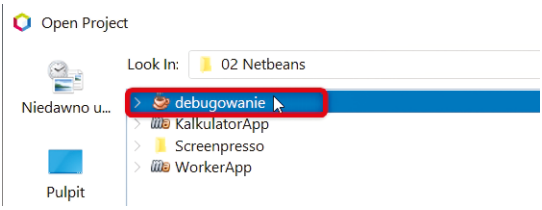
Korzystając z debugera, popraw błędy w kodzie aplikacji `Debugowanie`*. Metody* `zadanie1()`*,* `zadanie2()` *i* `zadanie3()` *powinny na wyjściu wypisywać komunikaty o dokładnie takiej treści, jak określono przed deklaracją każdej z metod* [5].

Aplikacja `Debugowanie` znajduje się w pliku `ps-netbeans.zip` [5], [6]. Należy rozpakować kod aplikacji i wczytać go, wybierając z menu opcję: **File → Open Project** (rys. 1.48).



RYSUNEK 1.48. **Otwieranie wcześniej utworzonego projektu**

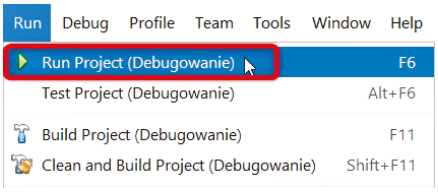
Wskazujemy lokalizację aplikacji `Debugowanie` na dysku (rys. 1.49).



RYSUNEK 1.49. **Wskazanie aplikacji** `Debugowanie`

Klikamy przycisk „Open Project" w celu otwarcia wybranego projektu.

Mając już otwarty projekt, kompilujemy go i uruchamiamy. W tym celu wybieramy z menu: **Run → Run Project** (rys. 1.50).



RYSUNEK 1.50. **Kompilacja i uruchomienie aplikacji**

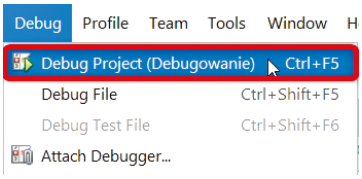
Na dole w okienku konsoli otrzymujemy komunikat:

Exception in thread „main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4

at debugowanie.Debugowanie.zadanie1(Debugowanie.java:26)

at debugowanie.Debugowanie.main(Debugowanie.java:9)

Komunikat informuje o błędzie przekroczenia indeksu tablicy. Przyjrzymy się temu bliżej, korzystając z debugera. W linii 26 wstawiamy punkt przerwania (klikamy w numer linii w kodzie – pojawi się wówczas mały czerwony kwadracik) i wybieramy z menu opcję: **Debug → Debug Project** (rys. 1.51).



RYSUNEK 1.51. **Uruchamianie procesu debugowania**

Kontynuujemy proces debugowania, wybierając z menu: **Debug → Continue** (`F5`) i jednocześnie na dole w okienku „Variables" obserwujemy wartości zmiennych. Naprawiamy błąd znajdujący się w 26 linii kodu.

Ponownie wybieramy z menu opcję **Run** → **Run Project** i widzimy na dole w konsoli taki komunikat:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
    at debugowanie.Debugowanie.zadanie1(Debugowanie.java:32)
    at debugowanie.Debugowanie.main(Debugowanie.java:9)

Komunikat ponownie informuje o błędzie przekroczenia indeksu tablicy. W linii 32 wstawiamy punkt przerwania i uruchamiamy proces debugowania. Naprawiamy błąd znajdujący się w 32 linii kodu.

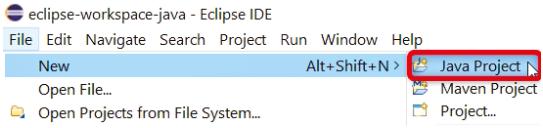
Metoda `zadanie1` nie zawiera już żadnych błędów. Teraz, korzystając z debugera w podobny sposób jak opisany powyżej, należy poprawić błędy w metodach `zadanie2` i `zadanie3`.

1.3. Eclipse

1.3.1. Zadanie 1 – tworzenie nowego projektu

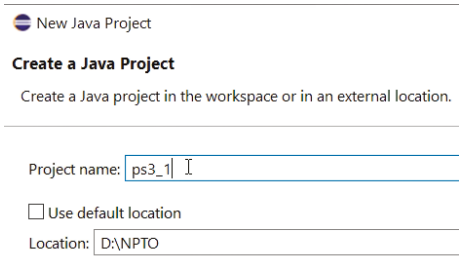
Stwórz aplikację w języku Java.

Tworzymy nowy projekt, wybierając z menu: **File** → **New** → **Java Project** (rys. 1.52).



RYSUNEK 1.52. **Tworzenie nowego projektu**

Wybieramy nazwę dla projektu oraz wskazujemy jego lokalizację na dysku. Możemy wpisać własne wartości lub pozostawić domyślne (rys. 1.53).



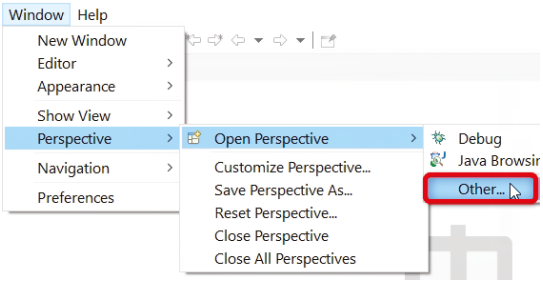
RYSUNEK 1.53. **Wybór nazwy projektu oraz jego lokalizacji na dysku**

Zatwierdzamy wybór, klikając przycisk „Finish" znajdujący się na dole okienka.

1.3.2. Zadanie 2 – przejście do widoku Java (defaults)

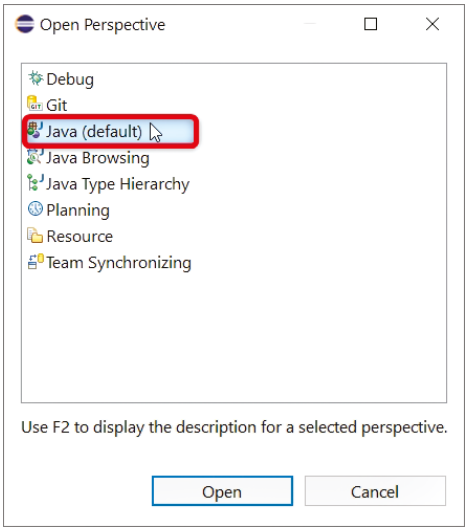
W nowo utworzonym projekcie ustaw widok „Java (default)”.

W celu ustawienia widoku „Java (default)” wybieramy z menu: **Window** → **Perspective** → **Open Perspective** → **Other** (rys. 1.54).



RYSUNEK 1.54. **Ustawianie widoku „Java (default)”**

W oknie „Open Perspective” wybieramy widok „Java (dafault)” (rys. 1.55).



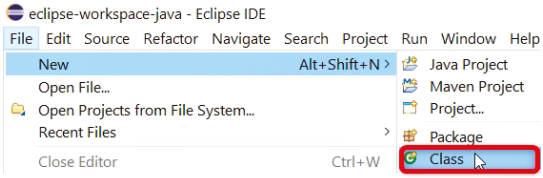
RYSUNEK 1.55. **Wybór widoku „Java (default)”**

Zatwierdzamy nasz wybór poprzez kliknięcie przycisku „Open”.

1.3.3. Zadanie 3 – dodawanie klasy

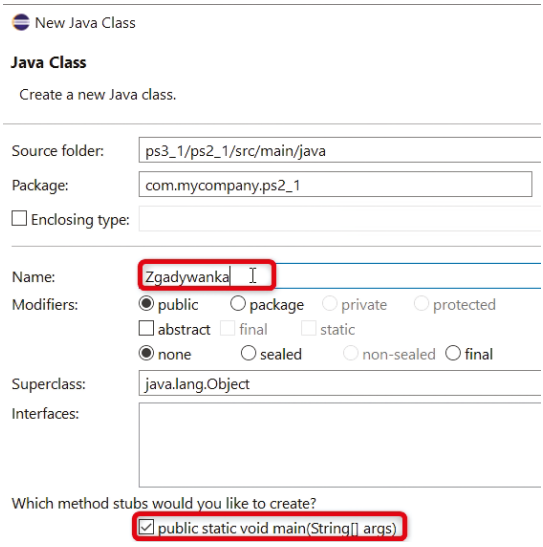
Utwórz w nowo założonym projekcie klasę zawierającą metodę startową `main`.

Dodajemy klasę w nowo założonym projekcie, wybierając z menu opcję: **File** → **New** → **Class** (rys. 1.56).



RYSUNEK 1.56. **Dodawanie klasy**

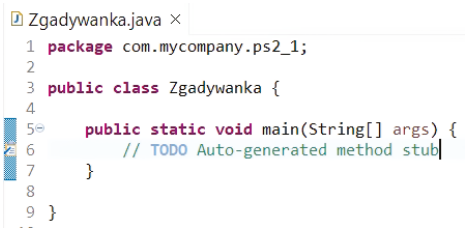
Nadajemy nazwę dla nowo tworzonej klasy oraz zaznaczamy opcję dodania do klasy metody startowej `main` (rys. 1.57).



RYSUNEK 1.57. **Konfiguracja nowej klasy**

Zatwierdzamy wszystko, klikając przycisk „Finish”.

Została utworzona nowa klasa o nazwie `Zgadywanka` (rys. 1.58).



RYSUNEK 1.58. **Kod nowo utworzonej klasy**

1.3.4. Zadanie 4 – uzupełnienie projektu prostym kodem

Uzupełnij projekt prostym kodem generującym liczbę od 0 do 100, który umożliwi następnie użytkownikowi odgadnięcie wylosowanej liczby [5].

LISTING 1.3. Przykładowy kod:

```
package com.mycompany.ps2 _ 1;

import java.util.Random;
import java.util.Scanner;

public class Zgadywanka {
      public static void main(String[] args) {

            Scanner scanner = new Scanner( System.in );
            Random rand = new Random();
            int random = rand.nextInt(100);
            boolean wynik = false;
            int ile _ prob=1;

            while(wynik==false)
            {
                System.out.print(„Wprowadź liczbę z zakresu
              0-100: “);
                int in = scanner.nextInt();
                if(in==random)
                {
                    System.out.println(„Znalazłeś szukaną
                  liczbę!!!”);
                    wynik=true;
                }
```

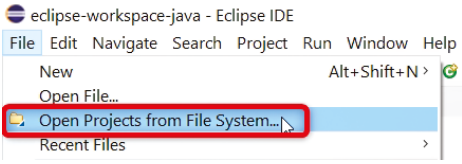
```
                else if (in>random)
                {
                    System.out.println(„Podałeś większą
                  liczbę.”);
                    wynik=false;
                    ile _ prob++;
                }
                else if (in<random)
                {
                    System.out.println(„Podałeś mniejszą
                  liczbę.”);
                    wynik=false;
                    ile _ prob++;
                }
            }
            System.out.println(„Zgadłeś za „+ile _ prob+”
          razem.”);
      }
}
```

1.3.5. Zadanie 5 – refaktoryzacja

Wykonaj poniższe czynności:

1. *W projekcie zastąp powtarzające się fragmenty kodu przez wydzielenie ich do odrębnej metody.*
2. *Zmień nazwę utworzonej metody i wszystkich odwołań do niej* [5]

Sprawdzamy, jaki mamy obszar roboczy (*workspace*) Eclipse poprzez wybranie: **File → Switch Workspace → Other**. W obszarze roboczym Eclipse tworzymy katalog o nazwie `HelloWorld` i kopiujemy do niego całą rozpakowaną zawartość projektu `Hello World` (plik `HelloWorld.zip`) [5], [6]. W celu otwarcia projektu w Eclipse wybieramy z menu: **File → Open Projects from File System or Archive** (rys. 1.59).



RYSUNEK 1.59. **Wybór otwarcia projektu znajdującego się w określonym katalogu**

W okienku, które się pojawi, klikamy przycisk „Directory” znajdujący się na górze tego okna po prawej stronie i wskazujemy lokalizację projektu `Hello World`. Nasz wybór zatwierdzamy, klikając przycisk „Wybierz folder”. W ostatnim kroku klikamy przycisk „Finish”. Otwarty projekt powinien znajdować się po lewej stronie w okienku „Project Explorer”.

Inny sposób otwarcia projektu `Hello World` polega na wyborze z menu: **File** → **Import** → **General** → **Existing Projects into Workspace** → **Next**. Obok pola „Select root directory” wciskamy przycisk „Browse” i wybieramy katalog z projektem `Hello World`. Następnie klikamy: **Wybierz folder** → **Finish**.

Zaznaczamy dowolny powtarzający się fragment kodu, który będziemy chcieli zastąpić za pomocą nowej metody (1.60).

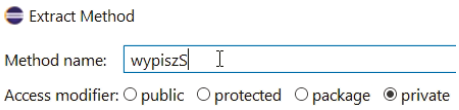
```
5      public int dodaj( int a, int b ){
6          int c = a + b;
7          System.out.println( "Sukces" );
8          return c;
9      }
10
11     public int odejmij( int a, int b ){
12         int c = a - b;
13         System.out.println( "Sukces" );
14         return c;
15     }
```

RYSUNEK 1.60. **Zaznaczenie powtarzającego się fragmentu kodu**

Wybieramy z menu: **Refactor** → **Extract Method** (1.61).

RYSUNEK 1.61. **Wybór z menu opcji „Extract Method”**

Wprowadzamy nazwę nowej metody (rys. 1.62).



RYSUNEK 1.62. **Wprowadzenie nazwy nowej metody**

Zatwierdzamy, klikając przycisk „OK”.

Wcześniej zaznaczony powtarzający się fragment kodu został zastąpiony nową metodą o nazwie `wypiszS` (rys. 1.63).

```
5    public int dodaj( int a, int b ){
6        int c = a + b;
7        wypiszS();
8        return c;
9    }
10
11   private void wypiszS() {
12       System.out.println( "Sukces" );
13   }
```

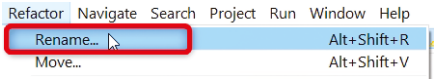
RYSUNEK 1.63. **Dodanie nowej metody o nazwie `wypiszS`**

W celu zmiany nazwy utworzonej metody i wszystkich odwołań do niej zaznaczamy tę nazwę w kodzie (rys. 1.64).

```
11   private void wypiszS() {
12       System.out.println( "Sukces" );
13   }
```

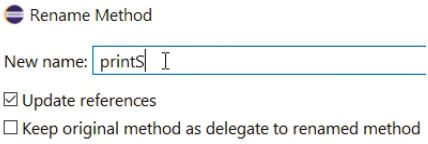
RYSUNEK 1.64. **Zaznaczenie nazwy metody**

Wybieramy z menu opcję: **Refactor** → **Rename** (rys. 1.65).



RYSUNEK 1.65. **Wybór opcji „Rename"**

Wpisujemy nową nazwę metody: `wypiszS` -> `printS` (rys. 1.66).



RYSUNEK 1.66. **Wpisywanie nowej nazwy metody**

Zatwierdzamy, klikając przycisk „OK".
W efekcie zmieniona została nazwa metody i wszystkich odwołań do niej (rys. 1.67).

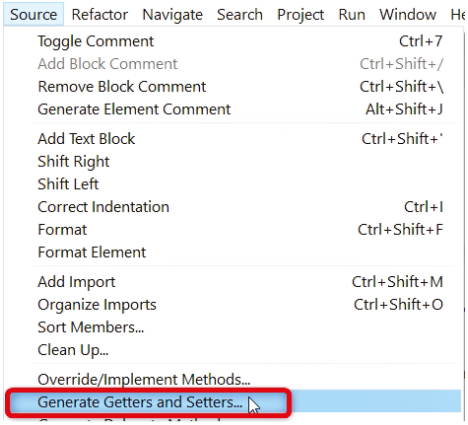
```
5    public int dodaj( int a, int b ){
6        int c = a + b;
7        printS();
8        return c;
9    }
10
11   private void printS() {
12       System.out.println( "Sukces" );
13   }
```

RYSUNEK 1.67. **Kod po zmianie nazwy wybranej metody**

1.3.6. Zadanie 6 – generowanie kodu

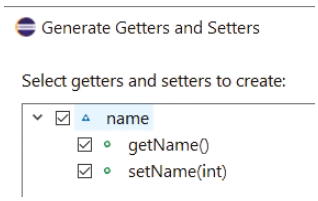
Wykonaj poniższe czynności:
1. *Dodaj metody dostępowe (gettery i settery) do atrybutu* `name` *klasy* `Projektor`.
2. *Napraw wcięcia w kodzie projektu* [5].

W tym zadaniu również korzystamy z projektu `Hello World`. W celu dodania metod dostępowych do atrybutu `name` klasy `Projektor` należy wybrać z menu opcję: **Source → Generate Getters and Setters** (rys. 1.68).



RYSUNEK 1.68. **Dodawanie metod dostępowych**

Zaznaczamy pole wyboru obok atrybutu `name` (rys. 1.69).



RYSUNEK 1.69. **Wybór atrybutu** `name`

Klikamy przycisk „Generate” znajdujący się na dole okienka.

W kodzie projektu pojawią się wygenerowane metody dostępowe (rys. 1.70).

```
11    public int getName() {
12        return name;
13    }
14
15    public void setName(int name) {
16        this.name = name;
17    }
```

RYSUNEK 1.70. **Metody dostępowe dodane do atrybutu** `name`

Robimy kilka przypadkowych wcięć w kodzie projektu (rys. 1.71).

```
19    private void printS() {
20 System.out.println( "Sukces" );
21    }
22
23public int odejmij( int a, int b ){
24    int c = a - b;
25                printS();
26        return c;
27    }
```

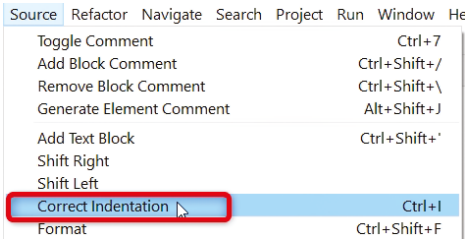
RYSUNEK 1.71. **Przypadkowe wcięcia w kodzie projektu**

Zaznaczamy fragment kodu, w którym wcięcia mają być naprawione (rys. 1.72).

```
18
19    private void printS() {
20 System.out.println( "Sukces" );
21    }
22
23public int odejmij( int a, int b ){
24    int c = a - b;
25                printS();
26        return c;
27    }
28
```

RYSUNEK 1.72. **Zaznaczenie fragmentu kodu, w którym należy naprawić wcięcia**

W celu naprawy wcięć w kodzie projektu wybieramy z menu opcję: **Source → Correct Indentation** (rys. 1.73).



RYSUNEK 1.73. **Wybór opcji „Correct indentation"**

Teraz widzimy, że wcięcia zostały naprawione (rys. 1.74).

```
19      private void printS() {
20          System.out.println( "Sukces" );
21      }
22
23      public int odejmij( int a, int b ){
24          int c = a - b;
25          printS();
26          return c;
27      }
```

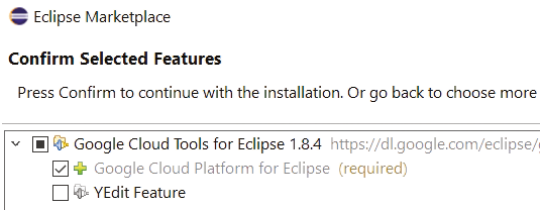
RYSUNEK 1.74. **Kod po zastosowaniu opcji naprawy wcięć**

1.3.7. Zadanie 7 – instalowanie wtyczek

Zainstaluj wtyczkę Google (Cloud Tools for Eclipse).

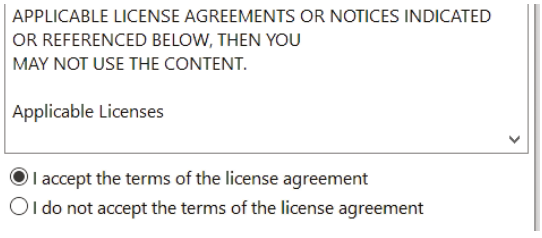
Na stronie `https://cloud.google.com/eclipse/docs/quickstart` w sekcji „Installing Cloud Tools for Eclipse" wybieramy jedną z dwóch metod instalacji wtyczki. Jeśli wybierzemy pierwszą, wystarczy tylko nacisnąć określony przycisk, przytrzymać go i przeciągnąć do uruchomionego środowiska Eclipse.

Rozpocznie się proces instalacji wtyczki (rys. 1.75).



RYSUNEK 1.75. **Instalacja wtyczki Cloud Tools for Eclipse**

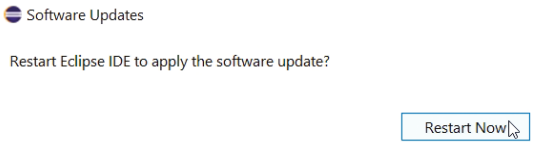
Instalację zatwierdzamy kliknięciem przycisku „Confirm".



RYSUNEK 1.76. **Tekst licencji**

Po przeczytaniu treści licencji akceptujemy ją (rys. 1.76) i klikamy przycisk „Finish".

Gdy wtyczka zostanie już zainstalowana, pojawi się okienko z propozycją restartu Eclipse. Klikamy przycisk „Restart Now” (rys. 1.77).



RYSUNEK 1.77. **Propozycja restartu Eclipse**

Od tego momentu możemy już korzystać z zainstalowanej wtyczki.

1.4. MS Visual Studio Code

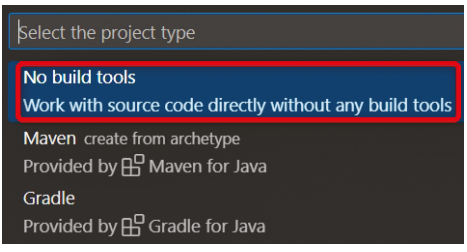
1.4.1. Zadanie 1 – tworzenie nowego projektu

Stwórz aplikację w języku Java, która wypełni 10-elementową tablicę losowymi wartościami z przedziału [0,1000] oraz wypisze wszystkie liczby parzyste na ekranie [5].

Sprawdzamy, czy mamy zainstalowaną obsługę programowania w języku Java. W tym celu na pasku po lewej stronie wybieramy „Explorer” (`Ctrl + Shift + E`). Powinien być widoczny przycisk „Create Java Project”. Ewentualnie sprawdzenia możemy też dokonać, wybierając z menu: **View → Command Palette**. W tym przypadku należy wpisać komendę: `Create Java Project`.

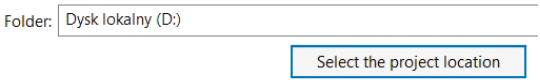
Jeśli nie mamy zainstalowanej obsługi programowania w języku Java, należy zainstalować rozszerzenie Extension Pack for Java. W tym celu na pasku po lewej stronie wybieramy „Extensions” (`Ctrl + Shift + X`) i wyszukujemy wspomniane wcześniej rozszerzenie, a następnie klikamy przycisk „Install”.

Tworzymy nowy projekt w języku Java. Na pasku po lewej stronie wybieramy „Explorer” i klikamy przycisk „Create Java Project”. Następnie wybieramy typ projektu „No build tools” (rys. 1.78).



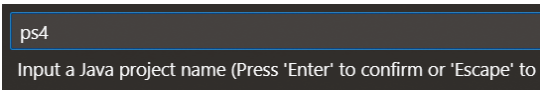
RYSUNEK 1.78. **Wybór typu projektu**

Wskazujemy lokalizację dla projektu na dysku i zatwierdzamy, klikając przycisk „Select the project location” (rys. 1.79).



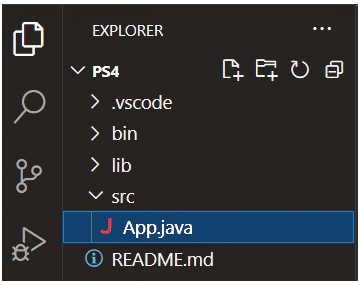
RYSUNEK 1.79. **Wybór lokalizacji dla projektu**

Wprowadzamy nazwę projektu i zatwierdzamy, wciskając enter (rys. 1.80).

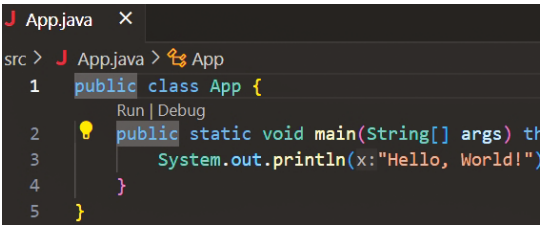


RYSUNEK 1.80. **Wybór nazwy projektu**

Został utworzony nowy projekt z przykładowym prostym kodem (rys. 1.81 i 1.82).



RYSUNEK 1.81. **Widok projektu w oknie „Explorer”**



RYSUNEK 1.82. **Kod utworzonego projektu**

Dostosowujemy kod do naszych potrzeb, zgodnie z treścią zadania: „Stwórz aplikację […], która wypełni 10-elementową tablicę wartościami losowymi z przedziału [0,1000] oraz wypisze wszystkie liczby parzyste na ekranie”. Przykładowy kod takiej aplikacji przedstawiono na listingu 1.4.

LISTING 1.4. Przykładowy kod aplikacji:

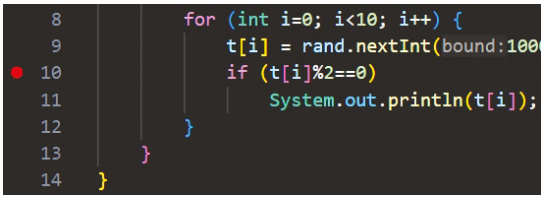
```
import java.util.Random;

public class App {
    public static void main(String[] args) throws Exception {
        Integer[] t = new Integer[10];
        Random rand = new Random();

        for (int i=0; i<10; i++) {
            t[i] = rand.nextInt(1000);
            if (t[i]%2==0)
                System.out.println(t[i]);
        }
    }
}
```

1.4.2. Zadanie 2 – debugowanie

Zapoznaj się z działaniem debugera:

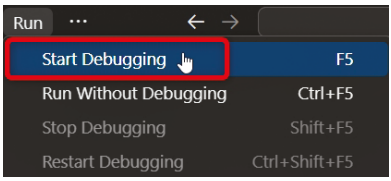
1. *Wstaw punkt przerwania.*
2. *Uruchom debugowanie.*
3. *Obserwuj wartości zmiennych w oknie po lewej stronie* [5].

Wstawiamy punkt przerwania widoczny w postaci czerwonej kropki po lewej stronie (rys. 1.83).



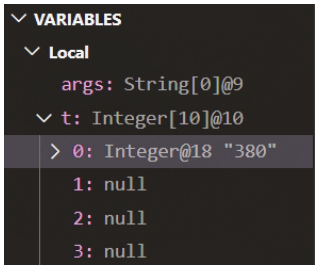
RYSUNEK 1.83. **Wstawianie punktu przerwania**

Uruchamiamy proces debugowania, np. poprzez wybranie z menu opcji: **Run** → **Start Debugging** (rys. 1.84).



RYSUNEK 1.84. **Uruchamianie procesu debugowania**

Obserwujemy wartości zmiennych w oknie „Variables" znajdującym się po lewej stronie (rys. 1.85).



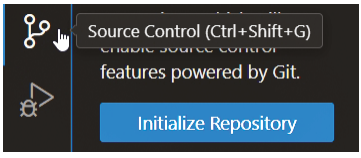
RYSUNEK 1.85. **Okno „Variables"**

Kontynuujemy proces debugowania, wybierając z menu: **Run** → **Continue** (`F5`).

1.4.3. Zadanie 3 – narzędzie kontroli wersji Git

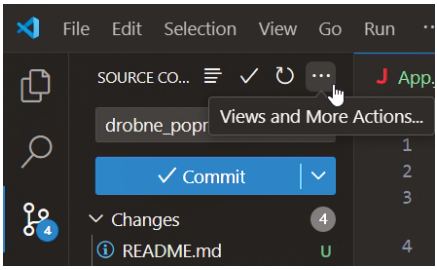
Skorzystaj z narzędzia kontroli wersji Git:
1. *Stwórz lokalne repozytorium.*
2. *Zatwierdź zmiany lokalnie.*
3. *Utwórz nową lokalną gałąź repozytorium* [5].

Tworzymy lokalne repozytorium poprzez wybranie na pasku po lewej stronie widoku „Source Control" (`Ctrl + Shift + G`), a następnie kliknięcie przycisku „Initialize Repository" (rys. 1.86).



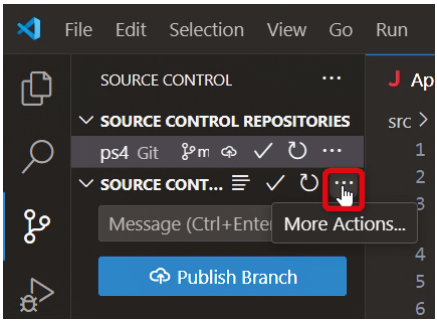
RYSUNEK 1.86. **Tworzenie lokalnego repozytorium**

Zatwierdzamy zmiany lokalnie, korzystając z widoku „Source Control”. Wpisujemy opis wersji kodu i klikamy przycisk „Commit”. Ewentualnie możemy wybrać także opcję: **Views and More Actions** → **Commit** → **Commit** (rys. 1.87).



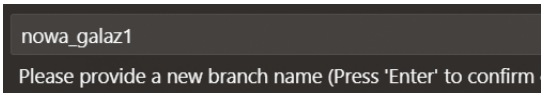
RYSUNEK 1.87. **Zatwierdzanie zmian lokalnie**

Tworzymy nową lokalną gałąź repozytorium, wybierając opcję: **More Actions** → **Branch** → **Create Branch** (rys. 1.88).



RYSUNEK 1.88. **Tworzenie nowej lokalnej gałęzi repozytorium**

Wprowadzamy nazwę dla nowej gałęzi i wciskamy enter (rys. 1.89).



RYSUNEK 1.89. **Wprowadzanie nazwy nowej gałęzi**

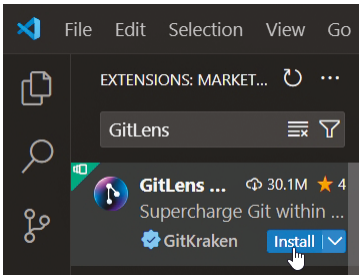
1.4.4. Zadanie 4 – instalowanie rozszerzeń

Wykonaj poniższe czynności:

1. *Zainstaluj rozszerzenia (dodatki): GitLens – Git supercharged, Javadoc Tools.*
2. *Przetestuj działanie zainstalowanych rozszerzeń* [5].

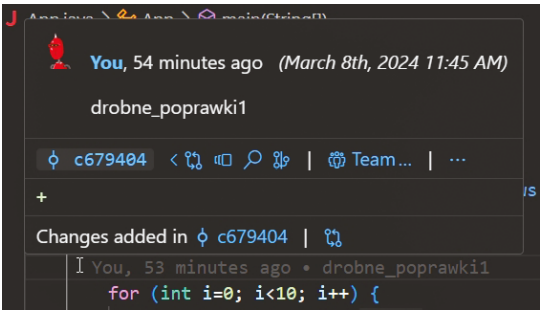
Dodatek GitLens – Git supercharged służy do wzbogacenia Visual Studio Code o zaawansowane funkcje pracy z Git, takie jak szczegółowa historia zmian, adnotacje autora i łatwiejsza analiza repozytorium.

Na pasku po lewej stronie wybieramy widok „Extensions” (`Ctrl + Shift + X`). Wyszukujemy rozszerzenie GitLens – Git supercharged. Klikamy przycisk „Install”, dzięki czemu dodatek zostanie zainstalowany (rys. 1.90).



RYSUNEK 1.90. **Instalacja rozszerzenia GitLens – Git supercharged**

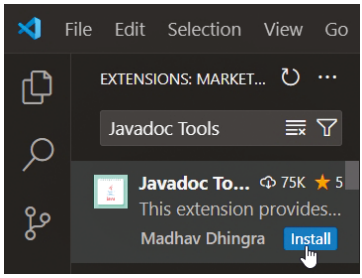
Testujemy działanie zainstalowanego dodatku GitLens – Git supercharged (rys. 1.91). Sprawdzamy, czy podczas pracy z repozytorium bezpośrednio w interfejsie Visual Studio Code prezentowane są historia zmian, adnotacje autora i inne informacje Git.



RYSUNEK 1.91. **Testowanie działania rozszerzenia GitLens – Git supercharged**

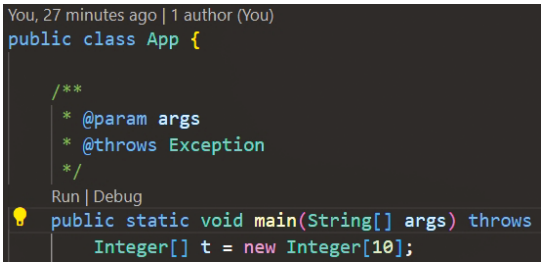
Dodatek Javadoc Tools służy do automatycznego generowania, uzupełniania i formatowania komentarzy Javadoc w kodzie Java w Visual Studio Code.

Na pasku po lewej stronie wybieramy widok „Extensions” (`Ctrl + Shift + X`). Wyszukujemy rozszerzenie Javadoc Tools. Klikamy przycisk „Install”, dzięki czemu dodatek zostanie zainstalowany (rys. 1.92).



RYSUNEK 1.92. Instalacja rozszerzenia Javadoc Tools

Testujemy działanie zainstalowanego dodatku Javadoc Tools. W widoku „Explorer” wybieramy plik, dla którego chcemy wygenerować komentarze, a następnie klikamy **Prawy przycisk myszy → Generate Javadoc Comments**. W efekcie otrzymujemy automatycznie wygenerowane komentarze (rys. 1.93).



RYSUNEK 1.93. Testowanie działania rozszerzenia Javadoc Tools

2. Systemy kontroli wersji – Git

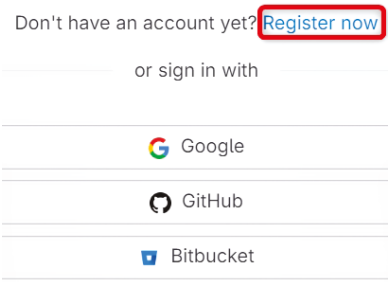
2.1. Zadanie 1 – rejestracja w serwisie GitLab

Zarejestruj się w serwisie GitLab: `https://gitlab.com`.

Przechodzimy do serwisu GitLab (`https://gitlab.com`). Z menu na górze po prawej stronie wybieramy opcję „Sign in” (rys. 2.1).

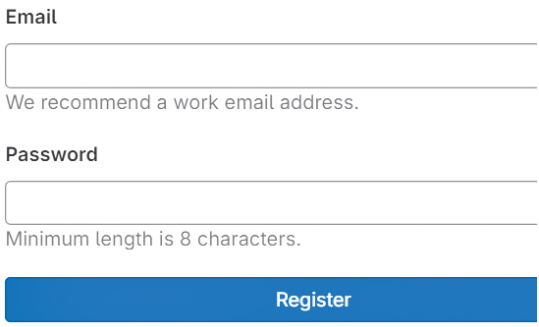
RYSUNEK 2.1. **Wybór opcji „Sign in”**

Poniżej formularza logowania wybieramy opcję rejestracji w serwisie GitLab, klikając „Register now” (rys. 2.2). Możemy też ewentualnie od razu zalogować się do serwisu danymi jednego z wcześniej utworzonych kont, np. Google, GitHub czy Bitbucket.



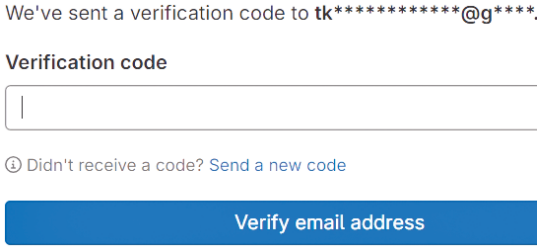
RYSUNEK 2.2. **Wybór opcji rejestracji w serwisie GitLab**

Wypełniamy formularz rejestracyjny, podając: imię, nazwisko, nazwę użytkownika, e-mail oraz hasło. Zatwierdzamy wszystko, klikając przycisk „Register” (rys. 2.3).



RYSUNEK 2.3. **Fragment formularza rejestracyjnego**

Na podany podczas rejestracji adres e-mail zostanie wysłany kod weryfikacyjny, który należy wprowadzić w odpowiednim polu i zatwierdzić poprzez kliknięcie przycisku „Verify email address” (rys. 2.4).



RYSUNEK 2.4. **Weryfikacja adresu e-mail**

Po poprawnej weryfikacji adresu e-mail możemy już cieszyć się z założonego konta w serwisie GitLab. Zostaniemy jeszcze poproszeni o wypełnienie krótkiego formularza powitalnego (rys. 2.5). Wypełniamy ten formularz i zatwierdzamy podane dane, klikając przycisk „Continue”.

RYSUNEK 2.5. **Fragment formularza powitalnego**

2.2. Zadanie 2 – przygotowanie Gita do działania

Przygotuj Gita do działania:

1. *W przypadku korzystania z narzędzia Git for Windows najpierw usuń z systemu Windows zapisane dane uwierzytelniające dotyczące serwisu* `gitlab.com`.
2. *Skonfiguruj swoje ustawienia Gita* [5].

Jeśli będziemy korzystać z narzędzia Git for Windows, usuwamy z systemu Windows (korzystając z panelu sterowania) zapisane dane uwierzytelniające dotyczące serwisu GitLab. W tym celu wybieramy następujące opcje: **Panel sterowania** → **Konta użytkowników** → **Menedżer poświadczeń**. Jeśli wśród poświadczeń znajdziemy wpis dotyczący serwisu GitLab, usuwamy go – w przeciwnym razie nic nie robimy.

Konfigurujemy swoje ustawienia Gita. W tym celu uruchamiamy wiersz polecenia i wprowadzamy trzy komendy (rys. 2.6–2.8).

```
D:\NPTO>git config --global user.email "j.kowalski@gmail.com"
```

RYSUNEK 2.6. Konfiguracja ustawień – e-mail

```
D:\NPTO>git config --global user.name "Jan Kowalski"
```

RYSUNEK 2.7. Konfiguracja ustawień – imię i nazwisko

```
D:\NPTO>git config --system --unset credential.helper
```

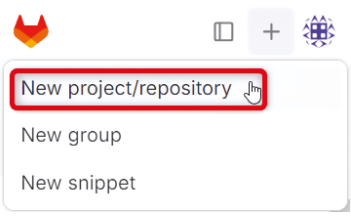
RYSUNEK 2.8. Konfiguracja ustawień – zapisywanie danych uwierzytelniających

W przypadku komendy, która nadpisuje domyślny sposób zapisywania danych uwierzytelniających przez Gita, wykorzystujący poświadczenia systemu Windows, należy liczyć się z tym, że prawdopodobnie pojawi się taki komunikat: „error: could not lock config file C:/Program Files/Git/etc/gitconfig: Permission denied". Należy wtedy jeszcze raz uruchomić wiersz polecenia, wybierając przy tym opcję „Uruchom jako administrator" i ponownie wpisać wspomnianą komendę, a następnie zamknąć okno wiersza polecenia.

2.3. Zadanie 3 – tworzenie projektu w serwisie GitLab

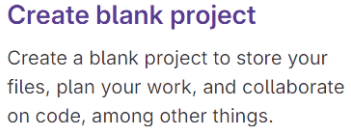
Utwórz nowy projekt w serwisie GitLab.

Logujemy się do serwisu GitLab, a następnie na górze po lewej stronie klikamy przycisk ze znakiem „+". Z rozwijanego menu wybieramy opcję „New project/repository" (rys. 2.9).



RYSUNEK 2.9. **Tworzenie własnego projektu w serwisie GitLab**

Wybieramy opcję „Create blank project", czyli utworzenie pustego projektu (rys. 2.10).



RYSUNEK 2.10. **Opcja „Create blank project"**

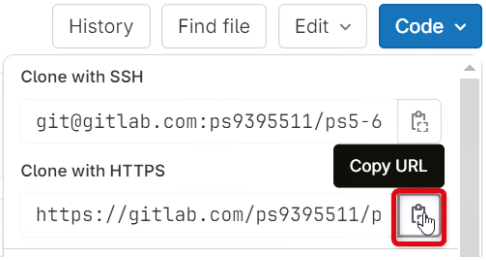
W formularzu, który się pojawi, w polu „Project name" wpisujemy nazwę projektu i klikamy przycisk „Create project" znajdujący się na samym dole formularza. Tym samym projekt zostaje utworzony.

2.4. Zadanie 4 – tworzenie lokalnej kopii zdalnego repozytorium

Utwórz lokalną kopię zdalnego repozytorium na swoim komputerze.

Będąc zalogowanymi w serwisie GitLab, wybieramy nasz projekt (**Your work → Projects → nazwa_grupy/nazwa_projektu**). Następnie klikamy przycisk z napisem „Code" i z rozwijanego menu wybieramy opcję „Clone with HTTPS".

Klikamy przycisk „Copy URL", dzięki czemu link prowadzący do repozytorium zostaje skopiowany do schowka (rys. 2.11).



RYSUNEK 2.11. **Wybór opcji „Copy URL"**

Mając już w schowku link prowadzący do repozytorium, możemy utworzyć na swoim komputerze lokalną kopię zdalnego repozytorium. Uruchamiamy wiersz polecenia i wpisujemy komendę `git clone` oraz wklejamy (`Ctrl + V`) link prowadzący do repozytorium (rys. 2.12).

```
D:\NPTO>git clone https://gitlab.com/ps9395511/ps5-6.git
Cloning into 'ps5-6'...
Username for 'https://gitlab.com': jkowalski25
Password for 'https://jkowalski25@gitlab.com':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

RYSUNEK 2.12. **Tworzenie lokalnej kopii zdalnego repozytorium**

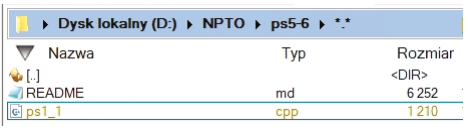
Ścieżka do lokalnego repozytorium w tym przypadku będzie wyglądała następująco: `D:\NPTO\ps5-6`.

UWAGA: w związku z tym, że przy tworzeniu projektu w serwisie GitLab pozostawiono domyślny poziom dostępu (**Visibility Level → Private**), w sytuacji użycia komendy `git clone` pojawi się prośba o podanie nazwy użytkownika serwisu GitLab i hasła. Oczywiście użytkownik ten powinien być widoczny w „Project members" (**Manage → Members**).

2.5. Zadanie 5 – wypełnianie katalogu lokalnego repozytorium plikami

Skopiuj do katalogu lokalnego repozytorium pliki źródłowe dowolnej aplikacji (np. pliki aplikacji konsolowej z podrozdziału 1.1.1).

Kopiujemy do katalogu lokalnego repozytorium przykładowy plik źródłowy projektu (rys. 2.13).



RYSUNEK 2.13. **Dodawanie do katalogu roboczego pliku źródłowego projektu**

2.6. Zadanie 6 – sprawdzenie statusu lokalnego repozytorium

Sprawdź status lokalnego repozytorium.

W wierszu polecenia, będąc w katalogu z repozytorium (`D:\NPTO\ps5-6`), używamy komendy `git status` (rys. 2.14).

```
D:\NPTO\ps5-6>git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will
        ps1_1.cpp

nothing added to commit but untracked files present
```

RYSUNEK 2.14. **Sprawdzanie statusu lokalnego repozytorium**

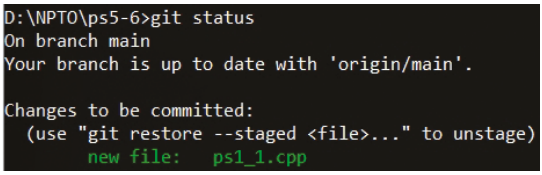
2.7. Zadanie 7 – dodanie do obszaru indeksu plików z katalogu roboczego

Dodaj wszystkie nowe oraz zmodyfikowane pliki z katalogu roboczego do obszaru indeksu (staging area), aby przygotować je do zatwierdzenia w repozytorium lokalnym.

Używamy komendy `git add .`, która dodaje wszystkie nowe bądź zmodyfikowane pliki do obszaru indeksu (rys. 2.15).

```
D:\NPTO\ps5-6>git add .
```

RYSUNEK 2.15. **Dodawanie do obszaru indeksu nowego pliku**

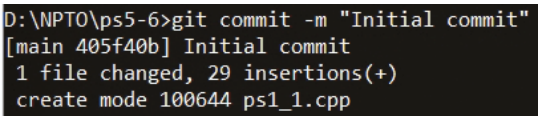


RYSUNEK 2.16. Status lokalnego repozytorium

2.8. Zadanie 8 – zatwierdzanie zmian lokalnie

Zatwierdź zmiany (*commit*) *lokalnie na konkretnej gałęzi (branch), załączając opis danej wersji kodu.*

Używamy komendy `git commit -m [opis wersji kodu]` (rys. 2.17).

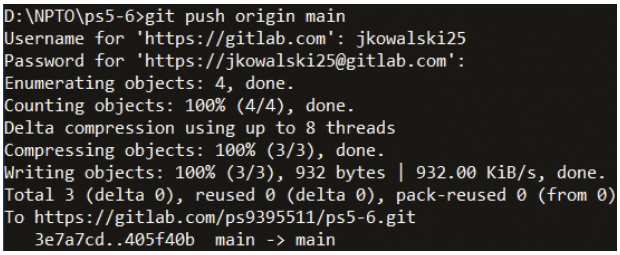


RYSUNEK 2.17. Zatwierdzanie zmian lokalnie

2.9. Zadanie 9 – przeniesienie zmian do repozytorium zdalnego

Przenieś lokalnie zatwierdzone zmiany do repozytorium zdalnego (*push*).

Używamy komendy `git push origin main` (rys. 2.18).



RYSUNEK 2.18. Przenoszenie zmian do repozytorium zdalnego

2.10. Zadanie 10 – wybór projektu do pracy w grupie

(do wykonania w grupie) *Wykonaj poniższe czynności:*

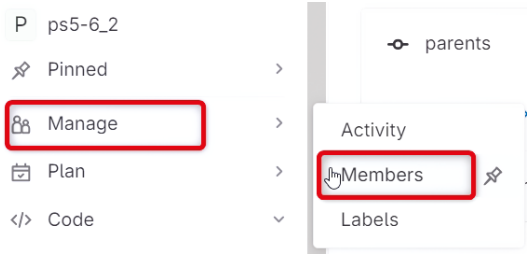
1. *Pracując w grupie dwu- lub trzyosobowej, wybierz wraz z innymi osobami jeden projekt, który zostanie wykorzystany do realizacji kolejnych zadań.*
2. *Jeśli wybranym projektem jest ten, którego jesteś autorem/autorką, kontynuuj pracę, korzystając z już istniejącego lokalnego repozytorium Git.*

Jeśli wybranym projektem jest projekt innego członka grupy, utwórz jego lokalną kopię poprzez sklonowanie odpowiedniego repozytorium zdalnego.

Celem zadania jest zapewnienie, że każdy członek grupy będzie posiadał lokalne repozytorium Git wybranego projektu gotowe do dalszej pracy.

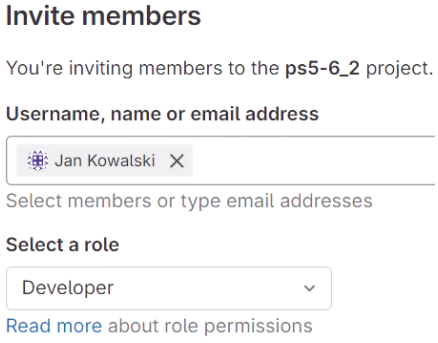
Zakładamy przypadek, w którym do realizacji zadań grupowych zostaje wybrany projekt innej osoby z grupy. W związku z tym musimy utworzyć lokalną kopię wybranego repozytorium zdalnego.

Osoba, której projekt chcemy wykorzystać, powinna dodać nas w GitLabie jako członka projektu poprzez: **Manage** → **Members** (rys. 2.19).



RYSUNEK 2.19. **Dodawanie nowego członka projektu**

Osoba zapraszająca powinna kliknąć przycisk „Invite members" znajdujący się na górze po prawej stronie i wypełnić krótki formularz zaproszenia do projektu (rys. 2.20).



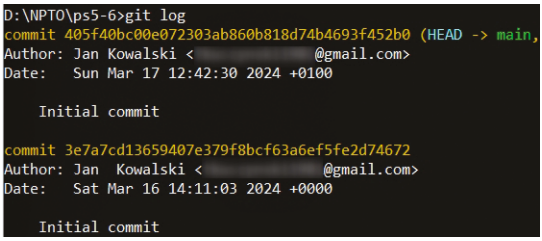
RYSUNEK 2.20. **Formularz zaproszenia do projektu**

Zapraszający zatwierdza przyjęcie do projektu, klikając przycisk „Invite". Na nasz adres e-mail powinna przyjść informacja o uzyskaniu dostępu do projektu innej osoby z grupy. Od tego momentu po zalogowaniu się do serwisu GitLab mamy dostęp do projektu utworzonego przez inną osobę. Możemy skopiować do schowka link prowadzący do repozytorium i użyć komendy `git clone URL_PROWADZĄCY_DO_REPOZYTORIUM`.

2.11. Zadanie 11 – sprawdzenie historii wersji kodu

Sprawdź historię wersji kodu.

Używamy komendy `git log` (rys. 2.21).

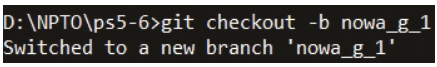


RYSUNEK 2.21. **Sprawdzanie historii wersji kodu**

2.12. Zadanie 12 – tworzenie nowej lokalnej gałęzi repozytorium

Utwórz nową gałąź w lokalnym repozytorium.

Używamy komendy `git checkout -b nazwa_galezi` (rys. 2.22).



RYSUNEK 2.22. **Tworzenie nowej lokalnej gałęzi repozytorium**

2.13. Zadanie 13 – ponowne przeniesienie zmian do zdalnego repozytorium

Wykonaj poniższe czynności:

1. *Zmodyfikuj wybrany plik z repozytorium lokalnego.*
2. *Zatwierdź zmiany w repozytorium.*

3. *Przenieś zmiany do zdalnego repozytorium na gałąź o tej samej nazwie co lokalna gałąź* [5].

W repozytorium lokalnym (`D:\NPTO\ps5-6`) modyfikujemy plik `ps1 _ 1.cpp`. Zmieniamy liczbę iteracji w pętli `for` z 10 na 20 (rys. 2.23).

```
int main()
{
    for (int a = 0; a < 20; a++) {
        if (a % 2 == 0)
            cout << "parzysta\n";
        else
            cout << "nieparzysta\n";
    }

    return 0;
}
```

RYSUNEK 2.23. **Modyfikacja pliku z repozytorium lokalnego**

Zanim zatwierdzimy zmiany, musimy użyć komendy `git add .`. Następnie zatwierdzamy zmiany, używając komendy `git commit -m [opis]` (rys. 2.24).

```
D:\NPTO\ps5-6>git add .

D:\NPTO\ps5-6>git commit -m "zwiekszenie liczby iteracji"
[nowa_g_1 f8fa5c8] zwiekszenie liczby iteracji
 1 file changed, 1 insertion(+), 1 deletion(-)
```

RYSUNEK 2.24. **Zatwierdzanie zmian**

W celu przeniesienia zmian do zdalnego repozytorium na gałąź o tej samej nazwie co gałąź lokalna użyjemy komendy `git push origin nazwa _ galezi` (rys. 2.25).

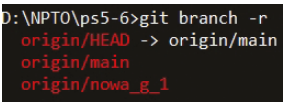
```
D:\NPTO\ps5-6>git push origin nowa_g_1
Username for 'https://gitlab.com': jkowalski25
Password for 'https://jkowalski25@gitlab.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 327 bytes | 327.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
```

RYSUNEK 2.25. **Przenoszenie zmian do zdalnego repozytorium**

2.14. Zadanie 14 – wypisanie zdalnych gałęzi w repozytorium

Wyświetl aktualną listę zdalnych gałęzi repozytorium.

Używamy komendy `git branch -r` (rys. 2.26).

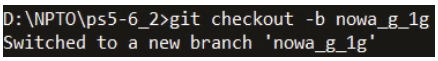


RYSUNEK 2.26. **Wyświetlanie zdalnych gałęzi w repozytorium**

2.15. Zadanie 15 – pobieranie zdalnej gałęzi i próba połączenia jej z lokalną

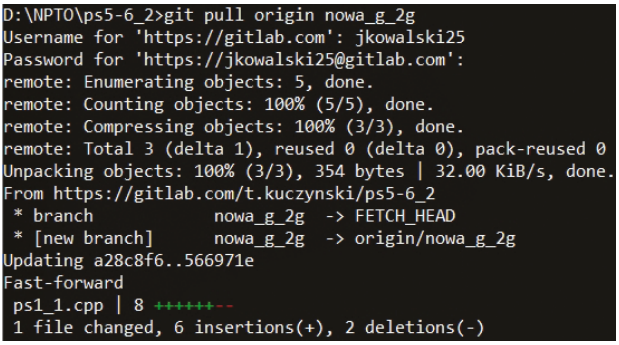
(do wykonania w grupie) *Utwórz w swoim repozytorium Git nową lokalną gałąź, która będzie odpowiadać gałęzi utworzonej przez innego członka grupy i udostępnionej w repozytorium zdalnym. Celem zadania jest utworzenie lokalnej gałęzi śledzącej wybraną gałąź zdalną innego członka zespołu.*

Przełączamy się na projekt grupowy (`D:\NPTO\ps5-6 _ 2`) i tworzymy nową lokalną gałąź (rys. 2.27).



RYSUNEK 2.27. **Tworzenie nowej lokalnej gałęzi**

Pobieramy ze zdalnego repozytorium zawartość wybranej gałęzi utworzonej przez inną osobę z grupy i przenosimy ją do nowej lokalnej gałęzi. Użyteczną komendą będzie tutaj `git pull origin nazwa _ zdalnej _ galezi` – pobiera zdalną gałąź i próbuje ją połączyć z gałęzią lokalną, czyli w praktyce wywołuje polecenia `git fetch` i `git merge` (rys. 2.28).

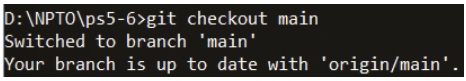


RYSUNEK 2.28. **Pobieranie zdalnej gałęzi i próba połączenia jej z gałęzią lokalną**

2.16. Zadanie 16 – scalanie gałęzi repozytorium

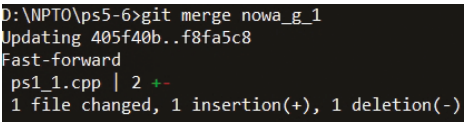
Włącz do lokalnej gałęzi, w której jesteś, zmiany z innej lokalnej gałęzi (*merge*). *Zwróć uwagę na różne sytuacje, w których scalanie gałęzi nie jest możliwe! Uważaj też na możliwe do wystąpienia konflikty* [5]*!*

Przełączamy się na gałąź, która ma otrzymać zmiany z innej gałęzi (rys. 2.29).



RYSUNEK 2.29. **Przełączanie się na gałąź, która ma otrzymać zmiany**

Włączamy do lokalnej gałęzi, w której jesteśmy, zmiany z innej lokalnej gałęzi. Użyteczna będzie tutaj komenda `git merge nazwa _ galezi` – dołącza zmiany z innej gałęzi. Należy jednak najpierw być na gałęzi, która ma otrzymać zmiany, a dopiero potem użyć podanej komendy (rys. 2.30).



RYSUNEK 2.30. **Włączanie zmian z innej lokalnej gałęzi**

2.17. Zadanie 17 – tworzenie sytuacji konfliktowej

Stwórz sytuację konfliktową (jeżeli nie wystąpiła w poprzednim zadaniu), a następnie postaraj się ją rozwiązać. Wykonaj poniższe czynności:

1. *W utworzonych wcześniej dwóch gałęziach lokalnych o tej samej zawartości zmodyfikuj ten sam plik, tę samą linię w pliku, ale w inny sposób (czyli ta sama linia będzie wyglądała w tych gałęziach zupełnie inaczej).*
2. *Zatwierdź zmiany.*
3. *Podejmij próbę scalenia zmian z jednej gałęzi z drugą.*
4. *Rozwiąż sytuację konfliktową* [5].

Będąc na gałęzi `main`, w pliku źródłowym modyfikujemy np. liczbę iteracji w pętli `for` (zmieniamy ją na 30). Zapisujemy zmiany. Stosujemy komendy `git add` oraz `git commit` (rys. 2.31).

```
D:\NPTO\ps5-6>git add .

D:\NPTO\ps5-6>git commit -m "liczba iteracji = 30"
[main 26a68fa] liczba iteracji = 30
 1 file changed, 2 insertions(+), 2 deletions(-)
```

RYSUNEK 2.31. **Zastosowanie komend na gałęzi** `main`

Przełączamy się na gałąź `nowa _ g _ 1` (`git checkout nowa _ g _ 1`). W tym samym pliku, w tej samej linii modyfikujemy ponownie liczbę iteracji w pętli `for` (zmieniamy ją na 50). Zapisujemy zmiany. Stosujemy komendy `git add` i `git commit` (rys. 2.32).

```
D:\NPTO\ps5-6>git add .

D:\NPTO\ps5-6>git commit -m "liczba iteracji = 50"
[nowa_g_1 5d0246e] liczba iteracji = 50
 1 file changed, 1 insertion(+), 1 deletion(-)
```

RYSUNEK 2.32. **Zastosowanie komend na gałęzi** `nowa_g_1`

Przełączamy się na gałąź `main` (`git checkout main`), gdyż to do niej będziemy chcieli włączyć zmiany z gałęzi `nowa _ g _ 1`. Stosujemy komendę `git merge`. W efekcie pojawia się sytuacja konfliktowa (rys. 2.33).

```
D:\NPTO\ps5-6>git merge nowa_g_1
Auto-merging ps1_1.cpp
CONFLICT (content): Merge conflict in ps1_1.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

RYSUNEK 2.33. **Sytuacja konfliktowa**

Komendy Git ułatwiające rozwiązywanie konfliktów to: `git status`, `git log --merge`, `git diff`, `git merge --abort` oraz `git reset`.

W pliku objętym konfliktem Git umieszcza znaczniki konfliktu, np.:

`<<<<<<< HEAD`

Treść z bieżącej gałęzi.

`=======`

Treść z gałęzi, z którą następuje scalenie.

`>>>>>>> nazwa_galezi`

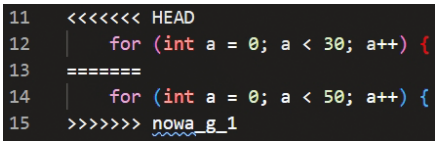
Wyjaśnienie znaczników:

`<<<<<<< HEAD` – zmiany z aktualnej gałęzi;

`=======` – separator;

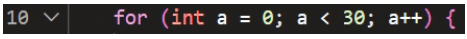
`>>>>>>>` ... – zmiany z drugiej gałęzi.

W celu ręcznego rozwiązania konfliktu należy otworzyć plik w edytorze tekstu lub IDE (rys. 2.34).



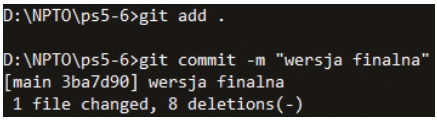
RYSUNEK 2.34. **Plik otwarty w Visual Studio Code**

W kolejnym kroku decydujemy, które zmiany chcemy zachować (rys. 2.35). Usuwamy wszystkie znaczniki konfliktu (`<<<<<<<`, `=======`, `>>>>>>>`) i zapisujemy plik.



RYSUNEK 2.35. **Zmiany do zachowania**

Po poprawieniu pliku dodajemy go do obszaru indeksu i zatwierdzamy zmiany (rys. 2.36).



RYSUNEK 2.36. **Dodawanie pliku do obszaru indeksu oraz zatwierdzanie zmian**

Sytuacja konfliktowa została rozwiązana.

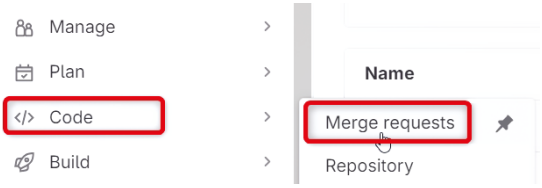
2.18. Zadanie 18 – nadanie prośby o włączenie zmian z jednej gałęzi do drugiej

(do wykonania w grupie) *Wykonaj poniższe czynności:*

1. *Na GitLabie utwórz żądanie scalenia (Pull Request/Merge Request), czyli nadaj prośbę o włączenie zmian z jednej gałęzi zdalnej do drugiej (ustal osobę, do której to żądanie będzie skierowane).*
2. *Postaraj się odpowiedzieć na prośbę nadaną przez inną osobę z grupy (poprzez zaakceptowanie tej prośby).*

Uwaga! Czasami wykonanie żądania scalenia może być niemożliwe, jeśli nie da się zmian automatycznie scalić [5].

Logujemy się do serwisu GitLab. Wybieramy nasz projekt grupowy (ps5-6_2). Z menu znajdującego się po lewej stronie wybieramy: **Code → Merge requests** (rys. 2.37).



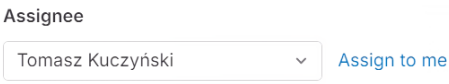
RYSUNEK 2.37. **Wybór opcji „Merge requests"**

W oknie głównym klikamy przycisk „New merge request". Wybieramy gałąź źródłową „Source branch" oraz docelową „Target branch" (rys. 2.38 i 2.39).

RYSUNEK 2.38. **Wybór gałęzi źródłowej**

RYSUNEK 2.39. **Wybór gałęzi docelowej**

Klikamy przycisk „Compare branches and continue”, a następnie wypełniamy formularz dotyczący nowego żądania scalenia. Ustalamy w nim m.in. osobę, do której będzie skierowana prośba o włączenie zmian z jednej gałęzi zdalnej do drugiej (rys. 2.40).



RYSUNEK 2.40. **Wybór osoby, do której będzie kierowana prośba**

Po wypełnieniu formularza klikamy przycisk „Create merge request”.

Chcąc odpowiedzieć na prośbę nadaną przez inną osobę z grupy (poprzez zaakceptowanie tej prośby), należy z menu znajdującego się po lewej stronie wybrać opcję: **Merge requests** → **Assigned** (2.41).

RYSUNEK 2.41. **Sprawdzanie przypisanych do nas próśb**

Klikamy w przypisane do nas żądanie, a następnie w przyciski „Approve” i „Merge”. Tym sposobem prośba zostaje zaakceptowana, a zmiany z gałęzi `nowa_g_2g` zostają włączone do gałęzi `main` (rys. 2.42).

RYSUNEK 2.42. **Zaakceptowanie prośby**

3. Dynamiczne testowanie aplikacji (pamięć)

3.1. Zadanie 1 – Valgrind i Memcheck (Linux)

Używając Valgrinda (i ewentualnie debuggera), znajdź i popraw błędy w obu programach opisanych poniżej oraz zlikwiduj w nich przyczyny wycieków pamięci.

Program `toupper` *zamienia słowo podane na jego wejściu na to samo słowo zapisane wielkimi literami, np.* `toupper abc` *=> ABC,* `toupper Hello` *=> HELLO.*

Program `bucket_sort` *jako argument pobiera liczbę naturalną N, generuje N losowych liczb rzeczywistych i sortuje je z zastosowaniem algorytmu sortowania kubełkowego. Przykład wywołania:* `bucket_sort 100` [5].

Valgrind jest zestawem narzędzi do dynamicznej analizy programów, natomiast Memcheck jest jednym z narzędzi dostępnych w ramach Valgrinda, przeznaczonym do wykrywania błędów związanych z zarządzaniem pamięcią.

Pracując w systemie Linux, kompilujemy program `toupper` (plik `ps-debugging.zip`) [5], [6]. Wpisujemy komendę `g++ -g toupper.cpp -o toupper`.

Uruchamiamy program pod kontrolą Valgrinda. Wpisujemy komendę `valgrind --leak-check=yes ./toupper Hello`.

Widzimy, że pamięć została zaalokowana dwa razy, ale tylko jeden raz zwolniona. Występuje więc utrata sześciu bajtów pamięci (rys. 3.1).

```
HEAP SUMMARY:
    in use at exit: 6 bytes in 1 blocks
  total heap usage: 2 allocs, 1 frees, 1,030 

6 bytes in 1 blocks are possibly lost in loss
   at 0x483B7F3: malloc (in /usr/lib/x86_64-l
k-amd64-linux.so)
   by 0x48F13BE: strdup (strdup.c:42)
   by 0x1091E4: to_upper(char const*) (touppe
   by 0x10925C: main (toupper.cpp:22)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 6 bytes in 1 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

RYSUNEK 3.1. **Valgrind – wyniki dla programu** `toupper`

Lista błędów, które naprawiono (kolor zielony – nowa linia kodu):

1. Brak biblioteki `stdlib.h` (wykorzystanie funkcji `free()`).

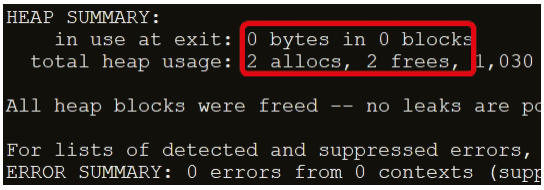
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
```

2. Brak wskaźnika pomocniczego na zmienną `upper` oraz brak zwolnienia pamięci.

```
upper = to_upper(argv[1]);
char *tmp = upper;

while (*upper)
  putchar (*(upper++));
puts ("");

free(tmp);
```

Po naprawieniu błędów w kodzie programu jeszcze raz kompilujemy program, a następnie uruchamiamy go pod kontrolą Valgrinda. W tym celu wpisujemy następujące komendy:

```
g++ -g toupper.cpp -o toupper
valgrind --leak-check=yes ./toupper Hello.
```



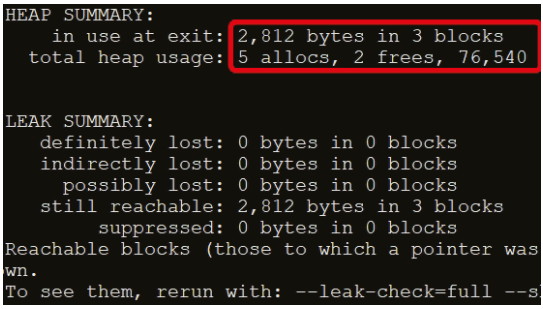
RYSUNEK 3.2. **Valgrind – wyniki dla programu** `toupper`

Teraz możemy zauważyć, że obu alokacjom pamięci odpowiadają dwa zwolnienia pamięci. Żadne bajty pamięci nie są więc tracone. Zlikwidowano przyczyny wycieków pamięci (rys. 3.2).

Pracując w systemie Linux, kompilujemy program `bucket_sort` (plik `ps-debugging.zip`) [5], [6]. Wpisujemy komendę `g++ -g bucket_sort.cpp -o bucket_sort`.

Uruchamiamy program pod kontrolą Valgrinda. Wpisujemy komendę `valgrind --leak-check=yes ./bucket_sort 100`.

Zauważmy, że pamięć została zaalokowana pięć razy, ale tylko dwa razy jest zwalniana. Występuje więc utrata 2812 bajtów pamięci (rys. 3.3).



RYSUNEK 3.3. **Valgrind – wyniki dla programu** `bucket_sort`

Lista błędów, które naprawiono (kolor czerwony – wersja błędna, kolor zielony – wersja poprawiona):

1. Nieprawidłowy rozmiar tablicy (linia 25).

```
K = new int[N-1];    K = new int[N+1];
```

2. Wyjście poza zakres liczb, jakie mogą być wygenerowane przez funkcję `rand()` (linia 35).

```
d[i] = WMIN+(double)rand() / (double)(RAND_MAX+1) * (WMAX - WMIN);
d[i] = WMIN+(double)rand() / (double)(RAND_MAX) * (WMAX - WMIN);
```

3. Użycie niezainicjalizowanej zmiennej (linia 46).

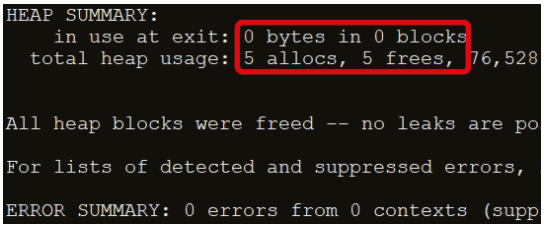
```
// ine  = 1;          ine  = 1;
```

4. Nieprawidłowe oraz niekompletne zwolnienie pamięci (linia 93).

```
free(L);             delete[] L;
                     delete[] d;
                     delete[] K;
```

Po naprawieniu błędów w kodzie programu powtórnie kompilujemy program oraz uruchamiamy go pod kontrolą Valgrinda. W tym celu wprowadzamy następujące komendy:

```
g++ -g bucket_sort.cpp -o bucket_sort
valgrind --leak-check=yes ./bucket_sort 100.
```



RYSUNEK 3.4. **Valgrind – wyniki dla programu** `bucket_sort`

Widzimy, że teraz pięciu alokacjom pamięci odpowiada pięć zwolnień pamięci. Żadne bajty pamięci nie są więc już tracone. Przyczyny wycieków pamięci zostały zlikwidowane (rys. 3.4).

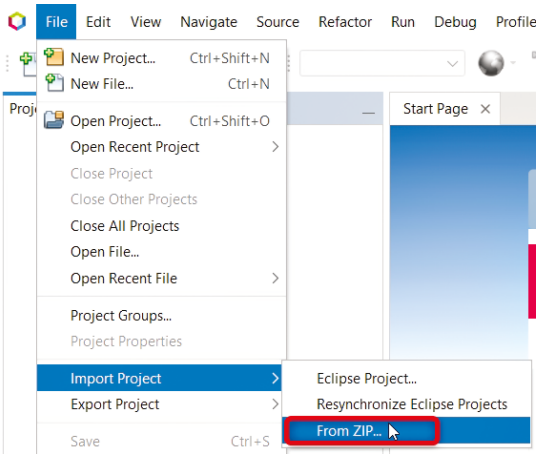
3.2. Zadanie 2 – NetBeans + Memory Profiler (Windows)

Wykonaj poniższe czynności:

1. *Otwórz aplikację* `stos _ memory _ prof` *w NetBeans.*
2. *Uruchom i przetestuj aplikację bez użycia profilera. Czy widzisz jakiś problem z aplikacją?*
3. *Uruchom aplikację z użyciem profilera pamięci:*
   - *podczas testowania aplikacji używaj opcji wymuszającej uruchomienie odśmiecacza pamięci (Garbage Collector);*
   - *przeanalizuj użycie pamięci w oknie „VM Telemetry Overview";*
   - *przeanalizuj zrzuty pamięci w różnych momentach działania aplikacji i porównaj, jakie obiekty z tej listy nie powinny się już tam znajdować po wyczyszczeniu stosu przy użyciu interfejsu graficznego użytkownika (GUI) oraz pamięci za pomocą odśmiecacza pamięci;*
   - *sprawdź, co zajmuje najwięcej zasobów w pamięci (okienko „Inspect" po prawej stronie w widoku zrzutu pamięci).*
4. *Popraw błąd w aplikacji i wykonaj profilowanie pamięci jak w punkcie 3.*

Jak nazywa się błąd, który został poprawiony? Jaka jest różnica w porównaniu z kodem niezarządzanym [5]?

Otwieramy aplikację `stos _ memory _ prof` (plik `stos _ memory _ prof.zip`) [5], [6] w NetBeans. W tym celu wybieramy z menu: **File** → **Import Project** → **From ZIP** (rys. 3.5).

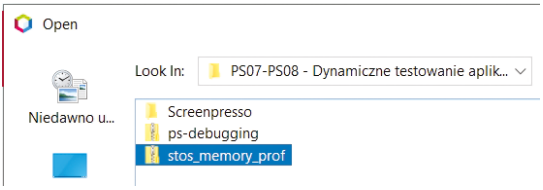


RYSUNEK 3.5. **Otwieranie aplikacji** `stos_memory_prof` **(1)**

Następnie obok pola „ZIP File" klikamy przycisk „Browse" (rys. 3.6).

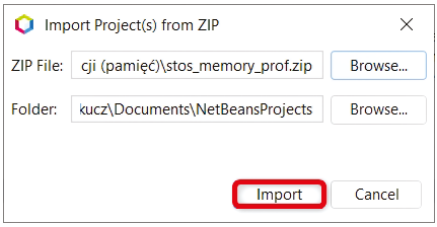
RYSUNEK 3.6. **Otwieranie aplikacji** `stos_memory_prof` **(2)**

Wskazujemy lokalizację pliku `stos _ memory _ prof.zip` na dysku (rys. 3.7) i klikamy przycisk „Open".



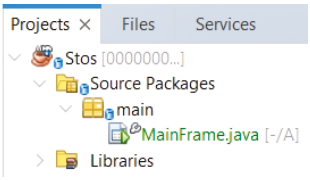
RYSUNEK 3.7. **Otwieranie aplikacji** `stos_memory_prof` **(3)**

Wybór zatwierdzamy, klikając przycisk „Import” (rys. 3.8).



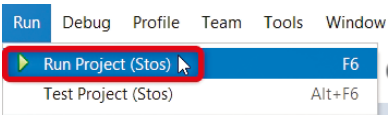
RYSUNEK 3.8. **Otwieranie aplikacji** `stos_memory_prof` **(4)**

W widoku „Projects” widzimy otwarty już projekt `Stos` (rys. 3.9).



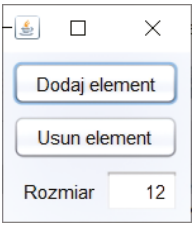
RYSUNEK 3.9. **Otwarta aplikacja** `stos_memory_prof`

Uruchamiamy aplikację bez użycia profilera: **Run → Run Project** (rys. 3.10 i 3.11).



RYSUNEK 3.10. **Uruchamianie aplikacji**

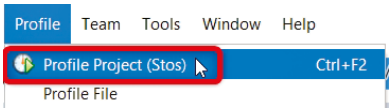
UWAGA: w przypadku pojawienia się błędów podczas uruchamiania aplikacji należy utworzyć nowy projekt w NetBeans i skopiować do niego zawartość kodu źródłowego z aplikacji `Stos`.



RYSUNEK 3.11. **Uruchomiona aplikacja**

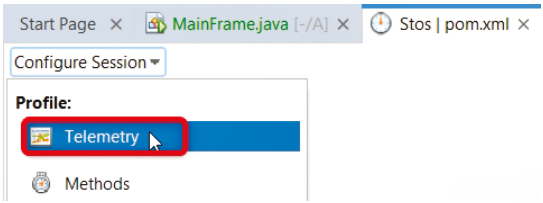
Podczas testów aplikacji bez użycia profilera nie widzimy żadnych problemów.

Uruchamiamy aplikację z użyciem profilera. W tym celu wybieramy z menu: **Profile** → **Profile Project** (rys. 3.12).



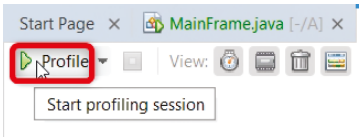
RYSUNEK 3.12. **Wybór opcji profilowania**

Wybieramy opcję profilowania pamięci: **Configure Session** → **Telemetry** (rys. 3.13).

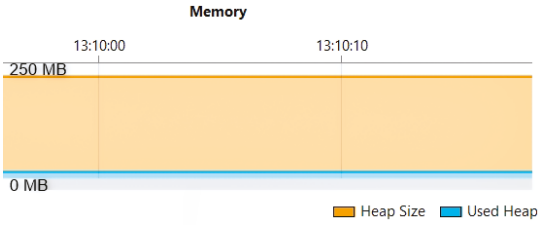


RYSUNEK 3.13. **Wybór opcji profilowania pamięci**

Klikamy przycisk „Profile" w celu rozpoczęcia procesu profilowania pamięci (rys. 3.14).

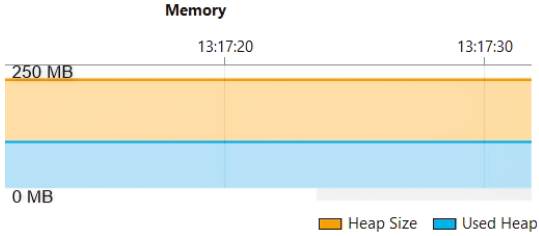


RYSUNEK 3.14. **Uruchamianie procesu profilowania**



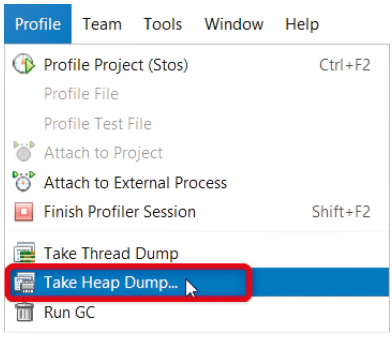
RYSUNEK 3.15. **Pamięć przed dodaniem elementów**

Poprzez użycie interfejsu graficznego użytkownika dodajemy 100 elementów na stos.



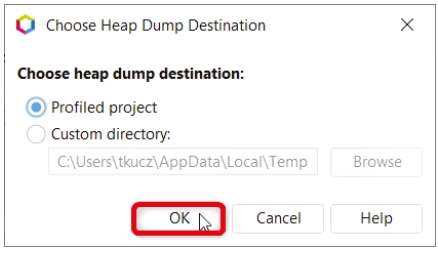
RYSUNEK 3.16. **Pamięć po dodaniu 100 elementów na stos**

W celu przeanalizowania zrzutów pamięci w różnych momentach działania aplikacji wybieramy z menu opcję: **Profile → Take Heap Dump** (rys. 3.17).



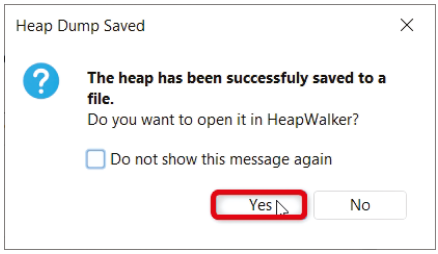
RYSUNEK 3.17. **Wybór opcji „Take Heap Dump"**

Wybieramy miejsce docelowe zrzutu pamięci i zatwierdzamy, klikając przycisk „OK" (rys. 3.18).



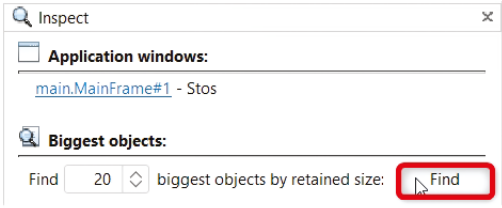
RYSUNEK 3.18. **Wybór miejsca docelowego zrzutu pamięci**

Wyrażamy zgodę na otwarcie zrzutu pamięci w narzędziu HeapWalker poprzez kliknięcie przycisku „Yes” (rys. 3.19).

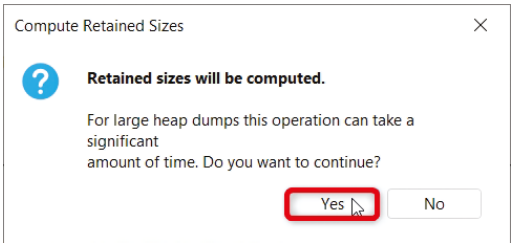


RYSUNEK 3.19. **Wybór otwarcia zrzutu pamięci w HeapWalker**

Sprawdzamy, co zajmuje najwięcej zasobów w pamięci przed wyczyszczeniem stosu przy użyciu interfejsu graficznego użytkownika oraz pamięci za pomocą odśmiecacza pamięci. W okienku „Inspect” znajdującym się po prawej stronie klikamy przycisk „Find” (rys. 3.20).



RYSUNEK 3.20. **Wyświetlanie listy największych obiektów**



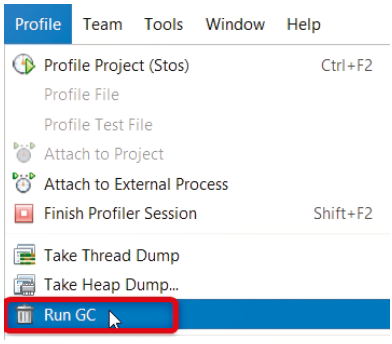
RYSUNEK 3.21. **Kontynuacja wyświetlania listy największych obiektów**

Po kliknięciu przycisku „Yes” (rys. 3.21) zostanie wyświetlona lista największych obiektów w pamięci (rys. 3.22).

Class Name	Retained Size
main.MainFrame#1	80 007 063
main.StosNaTablicy#1	80 005 652
main.Element[]#1	80 005 624
class org.netbeans.lib.profiler.server.ProfilerRuntime	1 200 220
byte[]#1	1 200 024
main.Element#26	800 048
main.Element#27	800 048
main.Element#28	800 048
main.Element#29	800 048
main.Element#30	800 048

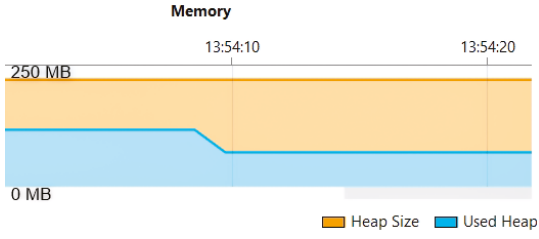
RYSUNEK 3.22. Lista największych obiektów w pamięci (1)

Czyścimy stos przy użyciu interfejsu graficznego użytkownika oraz pamięć za pomocą opcji wymuszającej uruchomienie odśmiecacza pamięci: **Profile → Run GC** (rys. 3.23).



RYSUNEK 3.23. Uruchomienie odśmiecacza pamięci

Widzimy, że pomimo usunięcia elementów ze stosu poprzez interfejs graficzny użytkownika oraz użycia odśmiecacza pamięci pamięć nie jest zwalniana (rys. 3.24).



RYSUNEK 3.24. Pamięć po usunięciu elementów i użyciu odśmiecacza pamięci

Używając opcji **Profile → Take Heap Dump**, również możemy zauważyć, że elementy stosu nie są usuwane z pamięci, gdy program przestaje z nich korzystać (rys. 3.25).

Class Name	Retained Size
main.MainFrame#1	80 007 063
main.StosNaTablicy#1	80 005 652
main.Element[]#1	80 005 624
class org.netbeans.lib.profiler.server.ProfilerRuntime	1 200 220
byte[]#1	1 200 024
main.Element#8	800 048
main.Element#9	800 048
main.Element#10	800 048
main.Element#11	800 048
main.Element#12	800 048

RYSUNEK 3.25. Lista największych obiektów w pamięci (2)

Błąd w kodzie aplikacji znajduje się w metodzie `pop()`. Polega on na pozostawieniu w tablicy referencji do usuniętego elementu, co może prowadzić do wycieku pamięci, ponieważ obiekt ten nie może zostać zwolniony przez odśmiecacz pamięci. Należy poprawić tę metodę (listing 3.1 i 3.2).

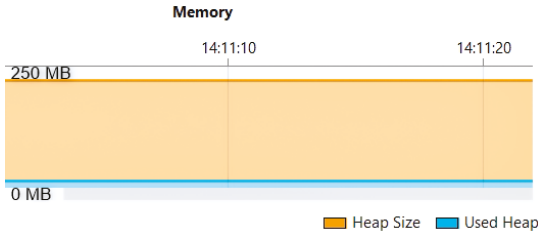
LISTING 3.1. Kod przed poprawą:

```
public Element pop() {
    if (size > 0) {
        return tab[--size];
    }
    else
        return null;
}
```

LISTING 3.2. Kod po poprawie:

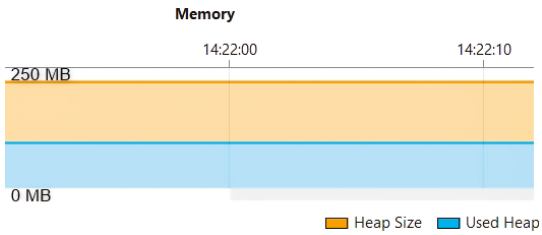
```
public Element pop() {
    if (size == 0) {
        return null;
    }
    Element e = tab[--size];
    tab[size] = null;
    return e;
}
```

Po naprawieniu błędu w aplikacji ponownie wykonujemy profilowanie pamięci: **Profile → Profile Project**.



RYSUNEK 3.26. Pamięć przed dodaniem elementów

Ponownie poprzez użycie interfejsu graficznego użytkownika dodajemy 100 elementów na stos (rys. 3.27).



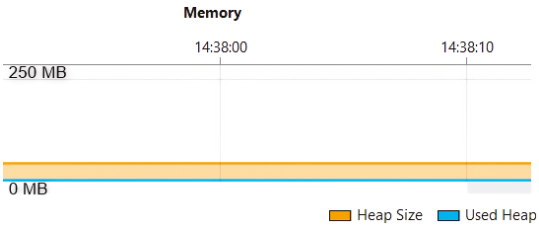
RYSUNEK 3.27. Pamięć po dodaniu 100 elementów na stos

Analizujemy zrzuty pamięci w różnych momentach działania aplikacji: **Profile → Take Heap Dump** (rys. 3.28).

Class Name	Retained Size
main.stos.Stos#1	80 007 063
main.stos.StosNaTablicy#1	80 005 652
main.stos.Element[]#1	80 005 624
class org.netbeans.lib.profiler.server.ProfilerRuntime	1 200 220
byte[]#1	1 200 024
main.stos.Element#4	800 048
main.stos.Element#5	800 048
main.stos.Element#6	800 048
main.stos.Element#7	800 048
main.stos.Element#8	800 048

RYSUNEK 3.28. Lista największych obiektów w pamięci (3)

Ponownie czyścimy stos przy użyciu interfejsu graficznego użytkownika oraz pamięć za pomocą opcji wymuszającej uruchomienie odśmiecacza pamięci (**Profile → Run GC**). Tym razem widzimy, że pamięć została zwolniona (rys. 3.29).



RYSUNEK 3.29. Pamięć po usunięciu elementów i użyciu odśmiecacza pamięci

Ponownie analizujemy zrzut pamięci: **Profile → Take Heap Dump** (rys. 3.30).

Class Name	Retained Size
class org.netbeans.lib.profiler.server.ProfilerRuntime	1 200 220
byte[]#1	1 200 024
main.stos.Stos#1	802 311
main.stos.StosNaTablicy#1	800 900
main.stos.Element[]#1	800 872
main.stos.Element#1	800 048
int[]#2	800 024
class java.time.zone.ZoneRulesProvider	249 210
class sun.util.calendar.ZoneInfoFile	168 303
java.time.zone.TzdbZoneRulesProvider#1	138 083

RYSUNEK 3.30. Lista największych obiektów w pamięci (4)

Widzimy, że elementy stosu są usuwane z pamięci.

4. Profilowanie aplikacji (czas)

4.1. Zadanie 1 – gprof (Linux)

Przeanalizuj program `gprof _ ex.cpp`*. Program na początku wykonuje losowanie elementów tablicy, a następnie elementy tablicy są sortowane metodą wstawiania. Na koniec wykonywane są dodatkowe obliczenia w funkcji* `doSomething()`*, której zawartość nie jest istotna. Wykonaj poniższe czynności:*

1. *Przeprowadź profilowanie programu za pomocą profilu płaskiego (flat profile) i grafu wywołań (call graph).*
2. *Zamień algorytm sortowania z algorytmu przez wstawianie na algorytm quicksort.*
3. *Wykonaj ponownie profilowanie aplikacji.*
4. *Porównaj wyniki z poprzednim profilowaniem* [5].

Celem profilowania aplikacji jest zidentyfikowanie fragmentów kodu o największym wpływie na czas wykonania programu oraz porównanie wydajności różnych rozwiązań algorytmicznych.

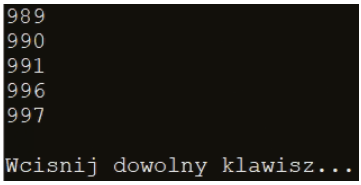
Profil płaski pokazuje, ile czasu program spędza w poszczególnych funkcjach niezależnie od ich wzajemnych wywołań, natomiast graf wywołań przedstawia relacje między funkcjami oraz rozkład czasu wykonania z uwzględnieniem hierarchii wywołań.

Pracując w systemie operacyjnym Linux, kompilujemy program `gprof _ ex.cpp` (plik `ps-profilowanie _ src.zip`) [5], [6] tak, aby dodawał niezbędne informacje dla profilera gprof. Podczas kompilacji oraz linkowania należy podać opcję -pg. Opcja ta spowoduje, że podczas działania aplikacji będą zbierane informacje, które później będziemy mogli analizować.

W celu kompilacji wprowadzamy komendę `g++ -pg gprof _ ex.cpp -o gprof _ ex`.

Uruchamiamy aplikację:.`/gprof _ ex`.

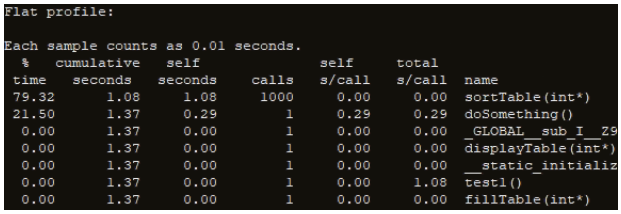
Po uruchomieniu aplikacji widzimy, że działa ona zgodnie z założeniami, tzn. losuje 1000 liczb z zakresu 0–999, sortuje je i wypisuje na ekranie (rys. 4.1). Wyniki (zebrane informacje dla profilera) zostały zapisane do pliku gmon.out.



RYSUNEK 4.1. **Fragment wyników z aplikacji**

Uruchamiamy profilera gprof w celu przeprowadzenia analizy otrzymanych wyników z pliku `gmon.out`. Wykonujemy profilowanie za pomocą profilu płaskiego (rys. 4.2):

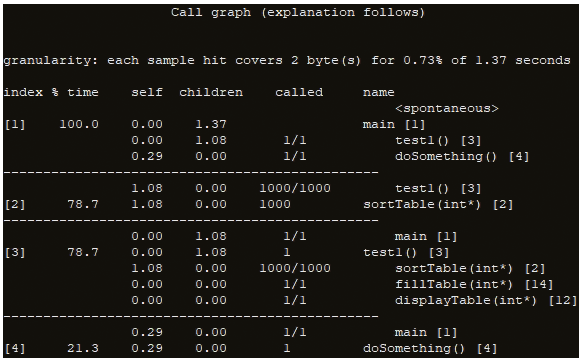
```
gprof -p ./gprof _ ex gmon.out.
```



RYSUNEK 4.2. **Profil płaski – sortowanie przez wstawianie**

Wykonujemy profilowanie za pomocą grafu wywołań (rys. 4.3):

```
gprof -q ./gprof _ ex gmon.out.
```



RYSUNEK 4.3. **Graf wywołań – sortowanie przez wstawianie**

Zmieniamy algorytm sortowania z algorytmu przez wstawianie na algorytm quicksort. W tym celu w metodzie `sortTable()` należy zakomentować kod dotyczący pierwszego przypadku (*insertion sort*) i odkomentować kod dotyczący drugiego przypadku (*quicksort*).

Po zmianie algorytmu na quicksort ponownie kompilujemy aplikację: `g++ -pg gprof_ex.cpp -o gprof_ex`.

Uruchamiamy aplikację: `./gprof_ex`.

Wykonujemy profilowanie za pomocą profilu płaskiego (rys. 4.4): `gprof -p ./gprof_ex gmon.out`.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 97.58      0.29     0.29        1   292.74   292.74  doSomething()
  3.36      0.30     0.01                             compareInt(void co
  0.00      0.30     0.00     1000     0.00     0.00  sortTable(int*)
  0.00      0.30     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z9
  0.00      0.30     0.00        1     0.00     0.00  displayTable(int*)
  0.00      0.30     0.00        1     0.00     0.00  __static_initializ
  0.00      0.30     0.00        1     0.00     0.00  test1()
  0.00      0.30     0.00        1     0.00     0.00  fillTable(int*)
```

RYSUNEK 4.4. **Profil płaski – sortowanie *quicksort***

Wykonujemy profilowanie za pomocą grafu wywołań (rys. 4.5): `gprof -q ./gprof_ex gmon.out`.

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 3.30% of 0.30 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]     96.7    0.00    0.29                 main [1]
                0.29    0.00       1/1           doSomething() [2]
                0.00    0.00       1/1           test1() [14]
-----------------------------------------------
                0.29    0.00       1/1           main [1]
[2]     96.7    0.29    0.00       1         doSomething() [2]
-----------------------------------------------
                                               <spontaneous>
[3]      3.3    0.01    0.00                 compareInt(void const*, vo
-----------------------------------------------
                0.00    0.00    1000/1000        test1() [14]
[10]     0.0    0.00    0.00    1000         sortTable(int*) [10]
```

RYSUNEK 4.5. **Graf wywołań – sortowanie *quicksort***

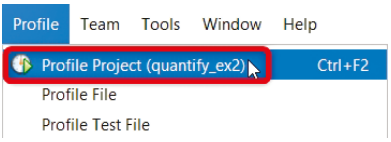
Po porównaniu wyników możemy stwierdzić, że zmiana algorytmu sortowania pozwoliła znacząco skrócić czas potrzebny na sortowanie tablicy.

4.2. Zadanie 2 – NetBeans (Windows)

Korzystając z NetBeans, wskaż w pliku `quantify _ ex2.java` *wydajniejsze metody w każdej parze metod:* `test1 _ 1()-test1 _ 2()`, `test2 _ 1()-test2 _ 2()`, `test3 _ 1()-test3 _ 2()` *i* `test4 _ 1()-test4 _ 2()` [5].

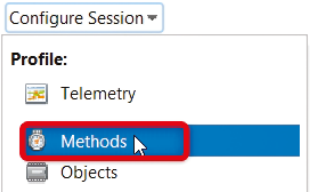
Tworzymy nowy projekt w środowisku NetBeans (**File → New Project → Java with Maven → Java Application**) i wklejamy do niego zawartość pliku `quantify _ ex2.java` (plik `ps-profilowanie _ src.zip`) [5], [6].

Wykonujemy profilowanie poprzez wybranie z menu opcji: **Profile → Profile Project** (rys. 4.6).



RYSUNEK 4.6. **Wybór opcji profilowania**

Wybieramy opcję profilowania metod: **Configure Session → Methods** (rys. 4.7).

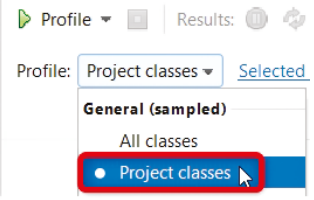


RYSUNEK 4.7. **Wybór opcji profilowania metod**

Po prawej stronie u góry wybieramy opcję dodatkowych ustawień, czyli „Settings" (rys. 4.8).

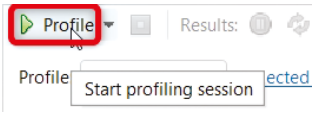
RYSUNEK 4.8. **Wybór opcji dodatkowych ustawień**

W dodatkowych ustawieniach wybieramy opcję profilowania tylko klas projektu: **Profile → Project classes** (rys. 4.9).



RYSUNEK 4.9. **Wybór opcji profilowania tylko klas projektu**

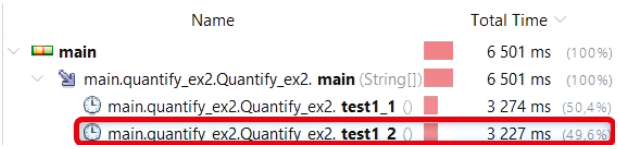
Uruchamiamy proces profilowania (rys. 4.10).



RYSUNEK 4.10. **Uruchomienie procesu profilowania**

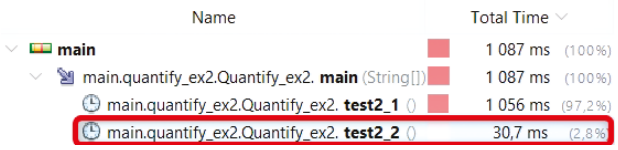
UWAGA: jeżeli metody nie są widoczne w wynikach profilowania, należy umieścić ich kod w pętli wykonywanej odpowiednio wiele razy. Liczbę iteracji można spróbować dobrać eksperymentalnie, tzn. najpierw dodać przykładowo 10 iteracji, a następnie, jeśli to okaże się niewystarczające, zwiększyć do 100, 1000 itd.

Wyniki profilowania pokazano na rysunkach 4.11–4.14.



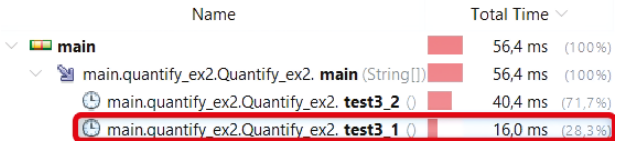
RYSUNEK 4.11. **Porównanie wydajności metod** `test1_1` **i** `test1_2`

W parze metod `test1 _ 1` i `test1 _ 2` wydajniejsza okazała się metoda **`test1 _ 2`**.



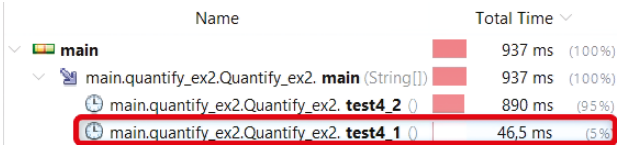
RYSUNEK 4.12. **Porównanie wydajności metod** `test2_1` **i** `test2_2`

W parze metod `test2 _ 1` i `test2 _ 2` wydajniejsza okazała się metoda **`test2 _ 2`**.



RYSUNEK 4.13. **Porównanie wydajności metod** `test3_1` **i** `test3_2`

W parze metod `test3 _ 1` i `test3 _ 2` wydajniejsza okazała się metoda **`test3 _ 1`**.



RYSUNEK 4.14. **Porównanie wydajności metod** `test4_1` **i** `test4_2`

W parze metod `test4 _ 1` i `test4 _ 2` wydajniejsza okazała się metoda **`test4 _ 1`**.

5. Dokumentowanie kodu – Javadoc

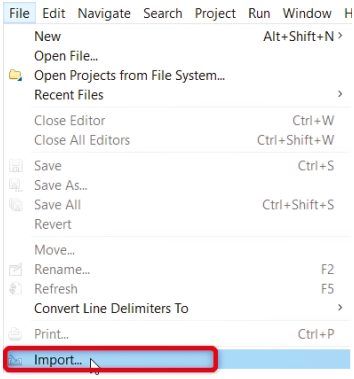
5.1. Zadanie 1 – import projektu

Wykonaj poniższe czynności:
1. *Rozpakuj projekt znajdujący się w katalogu* `Javadoc`.
2. *Przenieś go do obszaru roboczego Eclipse.*
3. *Otwórz projekt w Eclipse.*

Sprawdzamy, jaki mamy ustawiony obszar roboczy w Eclipse, wybierając z menu **File → Switch Workspace → Other**, a następnie umieszczamy tam rozpakowany z pliku `ps-dokumentowanie.zip` [5], [6] katalog `Projekt` (rys. 5.1).

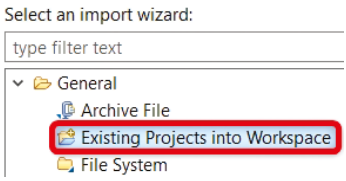
RYSUNEK 5.1. Katalog `Projekt` umieszczony w obszarze roboczym Eclipse

W celu otwarcia projektu w Eclipse wybieramy z menu: **File → Import** (rys. 5.2).



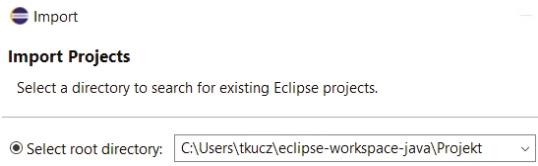
RYSUNEK 5.2. Otwarcie projektu w Eclipse (1)

Wybieramy opcję **General → Existing Project into Workspace** (rys. 5.3) i klikamy „Next”.



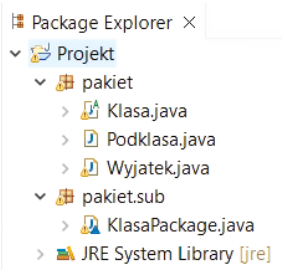
RYSUNEK 5.3. **Otwarcie projektu w Eclipse (2)**

Przy zaznaczonej opcji „Select root directory” wskazujemy lokalizację projektu (rys. 5.4) i zatwierdzamy, klikając przycisk „Finish”.



RYSUNEK 5.4. **Otwarcie projektu w Eclipse (3)**

W okienku „Package Explorer” znajdującym się po lewej stronie powinniśmy mieć widok otwartego projektu (rys. 5.5).



RYSUNEK 5.5. **Widok otwartego projektu w Eclipse**

5.2. Zadanie 2 – dodanie komentarzy

Dodaj własne komentarze:

- *w klasie* ***`Klasa`*** *– dla metody* ***`metoda`****;*
- *w klasie* ***`Klasa`*** *– dla wszystkich atrybutów;*
- *w klasie* ***`KlasaPackage`*** *– dla metody* ***`metodaWyjatkowa`*** [5].

W celu wstawienia komentarza klikamy w kod dotyczący danej metody lub atrybutu i wciskamy klawisze `Shift + Alt + J`. W opisie kodu możemy użyć szerokiej gamy znaczników (rys. 5.6–5.8), takich jak np.:

- **`@param`** – opisuje parametr metody (jego znaczenie i przeznaczenie);
- **`@return`** – informuje o wartości zwracanej przez metodę;
- **`@author`** – wskazuje autora klasy lub metody;
- **`@date`** – określa datę utworzenia lub ostatniej modyfikacji kodu;
- **`@version`** – podaje wersję danej klasy lub elementu kodu.

```
29    /**
30     * zwracanie liczby całkowitej
31     * @param intparm liczba całkowita
32     * @return liczba całkowita
33     */
34    protected Integer metoda(Integer intparm){
35        return intparm;
36    }
```

RYSUNEK 5.6. **Komentarz w klasie `Klasa` dla metody `metoda`**

```
 7    /**
 8     * atrybut przechowuje typ String
 9     */
10    public String stringAtr;
11
12    /**
13     * atrybut przechowuje typ Boolean
14     */
15    protected Boolean boolAtr;
16
17    /**
18     * atrybut przechowuje typ Double
19     */
20    Double doubleAtr;
21
22    /**
23     * atrybut przechowuje typ Integer
24     */
25    private Integer intAtr;
```

RYSUNEK 5.7. **Komentarze w klasie `Klasa` dla wszystkich atrybutów**

```
 7 /**
 8  * metoda z wyjątkiem
 9  * @param i liczba całkowita
10  * @return 0
11  * @throws Wyjatek jeżeli parametr jest równy 2
12  */
13 private int metodaWyjatkowa(int i) throws Wyjatek{
14     if (i == 2)
15         throw (new Wyjatek());
16     return 0;
```

RYSUNEK 5.8. **Komentarz w klasie `KlasaPackage` dla metody `metodaWyjatkowa`**

5.3. Zadanie 3 – dodanie opisu klasy

Dodaj krótki opis klasy `Klasa` *w postaci html, zawierający wypunktowaną listę atrybutów i ich widoczność* [5].

```
/**
 * Przykładowa klasa.
 * <h1>Lista atrybutów:</h1>
 * <ol>
 *      <li><b> stringAtr </b> public </li>
 *      <li><b> boolAtr </b> protected </li>
 *      <li><b> doubleAtr </b> package-private </li>
 *      <li><b> intAtr </b> private </li>
 * </ol>
 */
public abstract class Klasa {
```

RYSUNEK 5.9. Opis klasy `Klasa` w postaci html

5.4. Zadanie 4 – generowanie dokumentacji

Uruchom wewnątrz `obszar _ roboczy/Projekt` i porównaj wynik ze strukturą projektu i treścią komentarzy:

- *`javadoc -d html pakiet`*;
- *`javadoc -d html -subpackages pakiet`*;
- *`javadoc -d html -public -subpackages pakiet`*;
- *`javadoc -d html -protected -subpackages pakiet`*;
- *`javadoc -d html -package -subpackages pakiet`*;
- *`javadoc -d html -private -subpackages pakiet`* [5].

Uruchamiamy wiersz polecenia i wpisujemy komendę `javadoc` w celu sprawdzenia, czy jest rozpoznawana.

Jeśli zostanie wyświetlony komunikat, że komenda ta nie jest rozpoznawana, to chcąc wygenerować dokumentację, należy postąpić według jednego z niżej opisanych sposobów:

a) W wierszu polecenia przejść do katalogu `obszar _ roboczy/Projekt` i wpisać komendę zawierającą pełną ścieżkę do programu `javadoc`, np. `"C:\Program Files\Java\jdk-21\bin\javadoc" -d html pakiet`.

b) Przejść do **Panel Sterowania → System i zabezpieczenia → System → Zaawansowane ustawienia systemu → Zmienne środowiskowe**, a następnie w sekcji „Zmienne systemowe" zaznaczyć zmienną „Path", wybrać „Edytuj", kliknąć „Nowy" i wpisać ścieżkę do programu `javadoc` (np. `C:\Program Files\Java\jdk-21\bin`). Od teraz polecenie `javadoc` będzie już rozpoznawalne w wierszu

polecenia. W związku z tym uruchamiamy wiersz polecenia i przechodzimy do katalogu `obszar _ roboczy/Projekt`. Wprowadzamy komendę, która wygeneruje nam dokumentację, np. `javadoc -d html pakiet`.

W przypadku, gdy komenda `javadoc` jest rozpoznawana w wierszu polecenia, przechodzimy do katalogu `obszar _ roboczy/Projekt` i wpisujemy po kolei sześć komend w takiej postaci, jak podano w treści zadania.

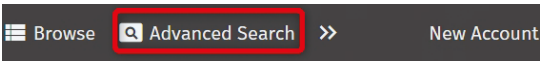
Generując różne wersje dokumentacji, porównujemy je ze strukturą projektu oraz treścią komentarzy. Wygenerowana dokumentacja będzie znajdowała się w katalogu `obszar _ roboczy/Projekt/html`. Należy otworzyć plik `index.html`.

6. Zarządzanie błędami – Bugzilla

6.1. Zadanie 1 – wyszukiwanie rozwiązanych zgłoszeń błędów

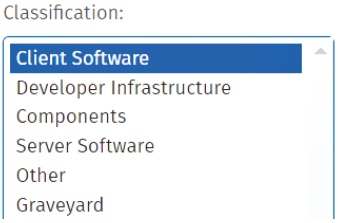
Wyszukaj w systemie Bugzilla projektu Mozilla (`https://bugzilla.mozilla.org`) *rozwiązane zgłoszenia błędów dotyczących instalatora dla produktu Firefox. Porównaj liczbę zgłoszeń, które zostały rozwiązane* (*fixed*)*, zgłoszeń błędnych* (*invalid*)*, uznanych za zduplikowane* (*duplicate*) *i nieokreślonych, odłożonych na później* (*worksframe*) [5].

Należy przejść na stronę `https://bugzilla.mozilla.org` i z górnego menu wybrać opcję „Advanced Search" (rys. 6.1).



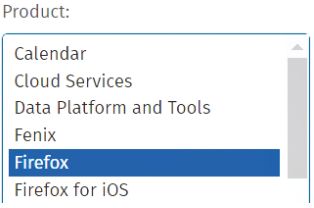
RYSUNEK 6.1. **Wybór opcji „Advanced Search"**

Wybieramy opcję: **Classification → Client Software** (rys. 6.2).



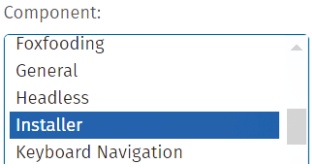
RYSUNEK 6.2. **Wybór opcji „Classification"**

Wybieramy opcję **Product → Firefox** (rys. 6.3).



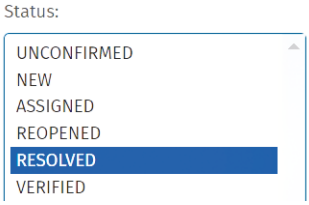
RYSUNEK 6.3. **Wybór opcji „Product"**

Należy wybrać komponent „Installer”: **Component** → **Installer** (rys. 6.4).



RYSUNEK 6.4. **Wybór opcji „Component”**

Wybieramy opcję rozwiązanych zgłoszeń błędów: **Status** → **Resolved** (rys. 6.5).



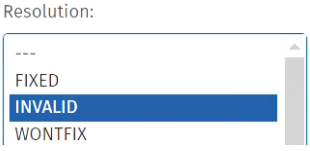
RYSUNEK 6.5. **Wybór opcji „Status”**

Wybieramy opcję: **Resolution** → **Fixed** (rys. 6.6).

RYSUNEK 6.6. **Wybór opcji „Resolution” (*fixed*)**

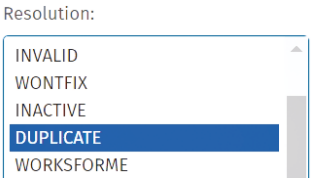
Po wybraniu odpowiednich opcji możemy uruchomić wyszukiwanie poprzez kliknięcie przycisku „Search”. Wyniki wyszukiwania mają ograniczenie związane z możliwością wyświetlenia do 500 błędów. Jeśli błędów jest więcej, to jeśli chcemy zobaczyć je wszystkie, musimy wybrać opcję „See all search results for this query”.

Zostało znalezionych **527** zgłoszeń, które zostały rozwiązane (*fixed*). Zmieniamy opcję „Resolution” z *fixed* na *invalid* (rys. 6.7) i ponownie klikamy przycisk „Search”.



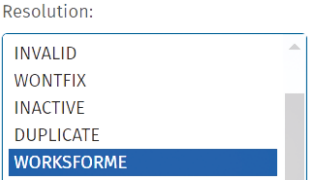
RYSUNEK 6.7. **Wybór opcji „Resolution” (*invalid*)**

Znaleziono **254** zgłoszenia błędne (*invalid*). Zmieniamy opcję „Resolution" z *invalid* na *duplicate* (rys. 6.8) i uruchamiamy proces wyszukiwania, klikając przycisk „Search".



RYSUNEK 6.8. **Wybór opcji „Resolution" (*duplicate*)**

Zostały znalezione **403** zgłoszenia uznane za zduplikowane (*duplicate*). Zmieniamy opcję „Resolution" z *duplicate* na *worksforme* (rys. 6.9) i klikamy przycisk „Search".



RYSUNEK 6.9. **Wybór opcji „Resolution" (*worksforme*)**

Znaleziono **322** zgłoszenia odłożone na później (*worksforme*).

Najwięcej znalezionych wyników dotyczy zgłoszeń, które zostały rozwiązane (*fixed* – 527 zgłoszeń), natomiast najmniej zgłoszeń błędnych (*invalid* – 254 zgłoszenia).

6.2. Zadanie 2 – prośba o wprowadzenie nowej funkcjonalności w aplikacji

Chcemy zwrócić się do twórców aplikacji USOS wspierającej proces dydaktyki na PB z prośbą o wprowadzenie nowej, interesującej nas funkcjonalności (np. możliwość zgłaszania propozycji tematów prac inżynierskich przez studentów). Prośbę można skierować za pomocą systemu Bugzilla obsługującego projekt USOS (`http://82.139.178.21/bugzilla`). Do wykonania zadania potrzebne będzie założenie konta w systemie Bugzilla [5].

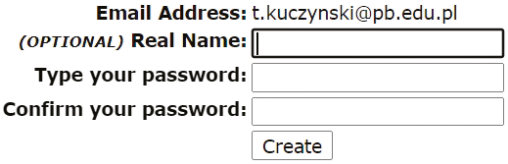
Należy przejść na stronę `http://82.139.178.21/bugzilla`, a następnie wybrać opcję „New Account" w celu założenia nowego konta w systemie Bugzilla (rys. 6.10).

RYSUNEK 6.10. **Wybór opcji „New Account"**

W polu „Email address" wpisujemy nasz adres e-mail (rys. 6.11) i klikamy przycisk „Send".

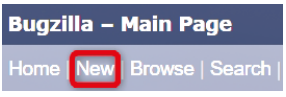
RYSUNEK 6.11. **Wprowadzanie adresu e-mail**

Na podany adres e-mail otrzymamy wiadomość, która pozwoli na utworzenie konta. Należy kliknąć w link umieszczony na początku wiadomości. W formularzu, który się pojawi, wprowadzamy hasło oraz zatwierdzamy proces tworzenia nowego konta poprzez kliknięcie przycisku „Create" (rys. 6.12).



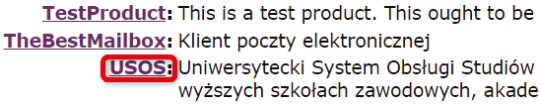
RYSUNEK 6.12. **Tworzenie nowego konta**

Po założeniu konta z paska na górze po lewej stronie należy wybrać opcję nowego zgłoszenia – „New" (rys. 6.13) – lub ewentualnie kliknąć przycisk „File a Bug".



RYSUNEK 6.13. **Wybór opcji nowego zgłoszenia**

Następnie wybieramy produkt USOS (rys. 6.14).

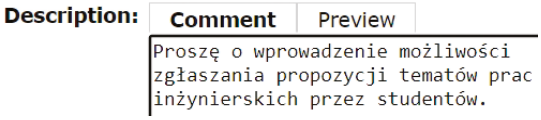


RYSUNEK 6.14. **Wybór produktu USOS**

Wypełniamy formularz zgłoszenia. W oknie „Component” wybieramy opcję „APD”, czyli moduł prac dyplomowych (rys. 6.15).

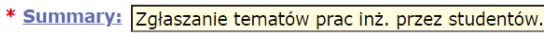
RYSUNEK 6.15. **Wybór modułu prac dyplomowych**

W polu „Description” wprowadzamy opis nowej funkcjonalności (rys. 6.16).



RYSUNEK 6.16. **Opis nowej funkcjonalności**

W polu „Summary” wpisujemy krótkie podsumowanie naszego zgłoszenia (rys. 6.17).



RYSUNEK 6.17. **Podsumowanie zgłoszenia**

W celu wysłania zgłoszenia klikamy przycisk „Submit Bug” znajdujący się na samym dole formularza.

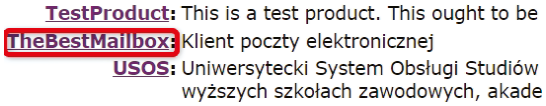
6.3. Zadanie 3 – zgłoszenie hipotetycznego błędu

Pracujemy nad aplikacją klienta poczty elektronicznej TheBestMailbox. System Bugzilla obsługujący zgłoszenia błędów naszej aplikacji jest dostępny pod adresem `http://82.139.178.21/bugzilla`. *Zgłoś hipotetyczny błąd znaleziony przez Ciebie podczas pracy z aplikacją. Jako adresata zgłoszenia określ kolegę/koleżankę. Niech kolega/koleżanka podejmie odpowiednie działania, aby zareagować na zgłoszenie* [5].

Należy przejść na stronę `http://82.139.178.21/bugzilla` i wybrać opcję „New” (rys. 6.18) lub ewentualnie „File a Bug” w celu zgłoszenia błędu.

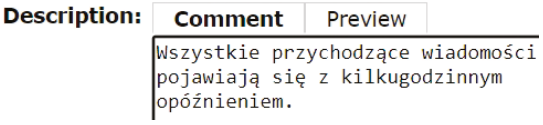
RYSUNEK 6.18. **Wybór opcji zgłoszenia błędu**

Wybieramy produkt TheBestMailbox (rys. 6.19).



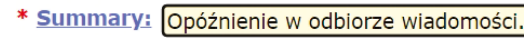
RYSUNEK 6.19. **Wybór produktu TheBestMailbox**

Wypełniamy formularz zgłoszenia. W polu „Description” wprowadzamy opis błędu (rys. 6.20).



RYSUNEK 6.20. **Opis błędu**

W polu „Summary” wpisujemy krótkie podsumowanie zgłoszenia błędu (rys. 6.21).



RYSUNEK 6.21. **Podsumowanie zgłoszenia błędu**

W formularzu zgłoszenia błędu wybieramy opcję „Show Advanced Fields” znajdującą się na górze po lewej stronie. W polu „Assignee” wpisujemy adres e-mail osoby (kolegi/koleżanki), do której adresowane jest zgłoszenie (rys. 6.22).

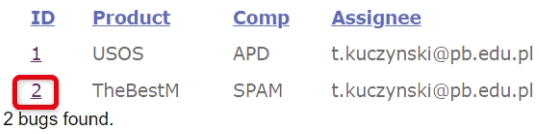
RYSUNEK 6.22. **Wpisywanie adresata zgłoszenia**

W celu wysłania zgłoszenia błędu klikamy przycisk „Submit Bug” znajdujący się na samym dole formularza.

Kolega/koleżanka powinien/powinna na swoim koncie wybrać opcję „Open bugs assigned to me” (rys. 6.23).

RYSUNEK 6.23. **Wybór opcji „Open bugs assigned to me”**

Osoba obsługująca zgłoszenie klika w numer („ID”) zgłoszenia, którym chce się zająć (rys. 6.24).



RYSUNEK 6.24. **Wybór zgłoszenia**

Pojawi się formularz, który należy wypełnić. Osoba obsługująca zgłoszenie może wypełnić np. takie pola, jak:

- „Hours Worked” – czas poświęcony na naprawianie błędu;
- „Additional Comments” – komentarz;
- „Status” – status zgłoszenia.

Po wypełnieniu odpowiednich pól formularza osoba obsługująca zgłoszenie zapisuje zmiany poprzez kliknięcie przycisku „Save Changes”.

7. Dystrybucja oprogramowania – Docker

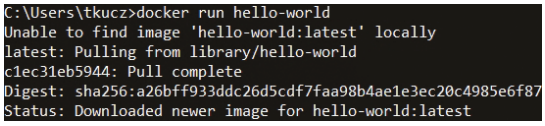
7.1. Zadanie 1 – prosta aplikacja uruchomiona we własnym kontenerze

Skorzystaj z najbardziej podstawowego obrazu z Docker Huba, który będzie zawierał prostą aplikację konsolową wypisującą tekst na konsoli, ale uruchomioną we własnym kontenerze: `https://hub.docker.com/ _ /hello-world`.
Wykonaj poniższe czynności:

1. *Uruchom kontener z obrazu z Docker Huba.*
2. *Wylistuj wszystkie kontenery aktualnie obecne na Twoim komputerze.*
3. *Wylistuj wszystkie obrazy, które są zapisane w pamięci cache.*
4. *Usuń jeden z kontenerów* [5].

Obraz Dockera jest niemodyfikowalnym szablonem zawierającym aplikację i jej środowisko uruchomieniowe, natomiast kontener jest uruchomioną instancją tego obrazu, działającą jako odizolowany proces w systemie operacyjnym.

Uruchamiamy aplikację Docker Desktop (jeśli wcześniej nie została uruchomiona) oraz wiersz polecenia. W celu uruchomienia kontenera z obrazu z Docker Huba (rys. 7.1) używamy w wierszu polecenia komendy: `docker run nazwa-obrazu`.



RYSUNEK 7.1. Uruchomienie kontenera z obrazu z Docker Huba

Chcąc wylistować wszystkie kontenery aktualnie obecne na komputerze (rys. 7.2), możemy użyć poleceń:
`docker ps`
`docker ps -all`.

```
C:\Users\tkucz>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS

C:\Users\tkucz>docker ps -all
CONTAINER ID   IMAGE         COMMAND    CREATED
842eacdfd5fe   hello-world   "/hello"   39 minutes ago
```

RYSUNEK 7.2. **Wylistowanie wszystkich kontenerów**

W celu wylistowania wszystkich obrazów z Dockera (rys. 7.3) użyjemy polecenia `docker images`.

```
C:\Users\tkucz>docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    d2c94e258dcb   12 months ago   13.3kB
```

RYSUNEK 7.3. **Wylistowanie wszystkich obrazów**

Aby usunąć jeden z kontenerów (rys. 7.4), możemy użyć polecenia `docker rm id-kontenera`.

```
C:\Users\tkucz>docker rm 842eacdfd5fe
842eacdfd5fe

C:\Users\tkucz>docker ps -all
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS
```

RYSUNEK 7.4. **Usuwanie kontenera**

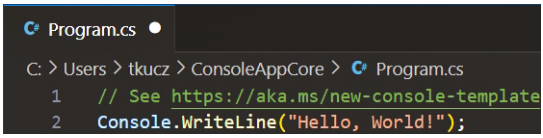
7.2. Zadanie 2 – własna przykładowa aplikacja konsolowa .NET Core

Utwórz przykładową aplikację konsolową .NET Core.

Chcąc utworzyć własną przykładową aplikację konsolową .NET Core (rys. 7.5 i 7.6), należy w wierszu polecenia wprowadzić komendę `dotnet new console -o ConsoleAppCore`.

```
C:\Users\tkucz>dotnet new console -o ConsoleAppCore
Pomyślnie utworzono szablon "Aplikacja konsoli".

Trwa przetwarzanie akcji po utworzeniu...
Przywracanie pakietu C:\Users\tkucz\ConsoleAppCore\ConsoleApp
  Trwa określanie projektów do przywrócenia...
  Przywrócono element C:\Users\tkucz\ConsoleAppCore\ConsoleAp
Przywracanie powiodło się.
```

RYSUNEK 7.5. **Tworzenie własnej aplikacji konsolowej .NET Core**



RYSUNEK 7.6. **Kod utworzonej aplikacji**

7.3. Zadanie 3 – tworzenie obrazu aplikacji z zadania 2

Wykonaj poniższe czynności:

1. *Przygotuj obraz wcześniej utworzonej aplikacji konsolowej. Potrzebny Ci będzie obraz bazowy .NET Core:* `https://hub.docker.com/ _ /microsoft-dotnet-sdk`.
2. *Uruchom przygotowany kontener* [5].

Dockerfile jest plikiem konfiguracyjnym zawierającym instrukcje umożliwiające automatyczne zbudowanie obrazu Dockera dla aplikacji.

Wykaz instrukcji do stworzenia pliku `Dockerfile`:

- `FROM` – inicjuje nowy proces budowania obrazu na podstawie wybranego obrazu bazowego;
- `WORKDIR` – ustawia katalog roboczy dla kolejnych instrukcji (jeżeli katalog nie istnieje, zostanie automatycznie utworzony);
- `COPY` – kopiuje pliki lub katalogi z systemu hosta do obrazu kontenera;
- `RUN` – uruchamia polecenia w trakcie budowania obrazu (np. *dotnet restore*), co pozwala m.in. na pobranie niezbędnych zależności;
- `ENTRYPOINT` – określa domyślne polecenie wykonywane po uruchomieniu kontenera;
- `EXPOSE` – informuje, na którym porcie aplikacja uruchomiona w kontenerze będzie nasłuchiwać (instrukcja ma charakter dokumentacyjny i sama nie powoduje automatycznego wystawienia portu na hosta, lecz może być wykorzystywana do jego publikacji).

LISTING 7.1. Przykładowy `Dockerfile` [5]:

```
FROM mcr.microsoft.com/dotnet/sdk
WORKDIR /ConsoleAppCoreDocker/

# kopiujemy plik .csproj oraz używamy polecenia dotnet restore
COPY *.csproj ./
RUN dotnet restore

# kopiujemy i dokonujemy build'a całej reszty
COPY . ./
RUN dotnet build -c Release
ENTRYPOINT [„dotnet”, „run”, „-c”, „Release”, „--no-build”]
```

W katalogu, w którym została utworzona aplikacja konsolowa (np. katalogu `ConsoleAppCore` jak w podrozdziale 7.2), umieszczamy plik `Dockerfile` o przykładowej zawartości (patrz: *Przykładowy* Dockerfile).

W wierszu polecenia, będąc w katalogu z wcześniej utworzoną aplikacją konsolową, wpisujemy instrukcję do budowania `Dockerfile` (rys. 7.7): `docker build -t consoleapp-core-docker .`.

```
C:\Users\tkucz\ConsoleAppCore>docker build -t consoleapp-core-docker .
[+] Building 0.0s (0/0)  docker:default
2024/05/11 13:25:09 http2: server: error reading preface from client //
[+] Building 72.6s (11/11) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 360B
 => [internal] load metadata for mcr.microsoft.com/dotnet/sdk:latest
 => [internal] load .dockerignore
 => => transferring context: 2B
```

RYSUNEK 7.7. **Budowanie** `Dockerfile`

Sprawdzamy, czy nowo utworzony obraz jest widoczny (rys. 7.8).

```
C:\Users\tkucz\ConsoleAppCore>docker images
REPOSITORY                TAG       IMAGE ID       CREATED          SIZE
consoleapp-core-docker    latest    866ee37631a5   27 minutes ago   866MB
hello-world               latest    d2c94e258dcb   12 months ago    13.3kB
```

RYSUNEK 7.8. **Wylistowanie obrazów**

Uruchamiamy kontener (rys. 7.9).

```
C:\Users\tkucz\ConsoleAppCore>docker run consoleapp-core-docker
Hello, World!
```

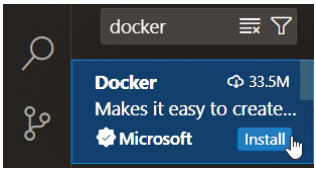
RYSUNEK 7.9. **Uruchomienie kontenera**

7.4. Zadanie 4 – instalacja dodatku do MS Visual Studio Code

Wykonaj poniższe czynności:

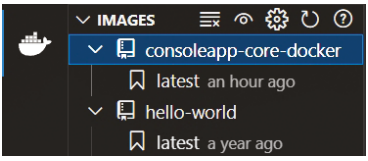
1. *Zainstaluj dodatek do Visual Studio Code umożliwiający pracę z Dockerem.*
2. *Zaprezentuj praktyczne działanie zainstalowanego dodatku.*

Uruchamiamy Visual Studio Code. Na pasku po lewej stronie wybieramy „Extensions" (`Ctrl + Shift + X`) i w polu wyszukiwania wpisujemy słowo „docker". Poprzez kliknięcie przycisku „Install" instalujemy znaleziony dodatek wspomagający pracę z Dockerem (rys. 7.10).



RYSUNEK 7.10. **Instalacja dodatku Docker**

Po instalacji dodatku możemy zarządzać Dockerem również z poziomu Visual Studio Code. Wystarczy kliknąć w ikonę Dockera znajdującą się na pasku po lewej stronie (rys. 7.11).



RYSUNEK 7.11. **Zarządzanie Dockerem z Visual Studio Code**

Bibliografia

[1] Pressman, R. S., & Maxim, B. R. (2019). *Software engineering: a practitioner's approach*. McGraw-Hill.

[2] Roman, A. (2022). *Testowanie i jakość oprogramowania: modele, techniki, narzędzia*. PWN.

[3] Sobczak, M. (2020). *Jakość oprogramowania: podręcznik dla profesjonalistów*. Helion.

[4] van Vliet, H. (2008). *Software engineering: principles and practice*. John Wiley and Sons.

[5] C. E. Z., Wydział Informatyki Politechniki Białostockiej. *Narzędzia Procesu Tworzenia Oprogramowania – stacjonarne*. https://cez.wi.pb.edu.pl/course/view.php?id=577

[6] C. E. Z., Wydział Informatyki Politechniki Białostockiej. *Narzędzia Procesu Tworzenia Oprogramowania - stacjonarne*. https://gitlab.com/npto-pb/npto-ps

[7] USOSweb: Wydział Informatyki Politechniki Białostockiej. *Karta przedmiotu: Narzędzia procesu tworzenia oprogramowania*. https://usosweb.pb.edu.pl/sylabusy_pdf/47-86561773831536.pdf

[8] Fowler, M., Beck, K., Roberts, D., & Gamma, E. (2017). *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*. WNT.

[9] Gajda, W. (2022). *Git: rozproszony system kontroli wersji*. Helion.

Spis rysunków

Politechnika
Białostocka