Politechnika Świętokrzyska w Kielcach

Wydział Elektroniki, Automatyki i Informatyki

Projekt: **Technologie obiektowe**

Grupa: **1ID21B**

Piotr Rojek

Data: 26.06.2022 r.

Spis treści

1. Wstęp

Celem projektu było utworzenie wtyczki do środowiska IntelliJ IDEA, która umożliwiałaby generowanie diagramów klas. W czasie rozwoju wtyczki rozszerzono pierwotne założenia funkcjonalności o metryki oprogramowania.

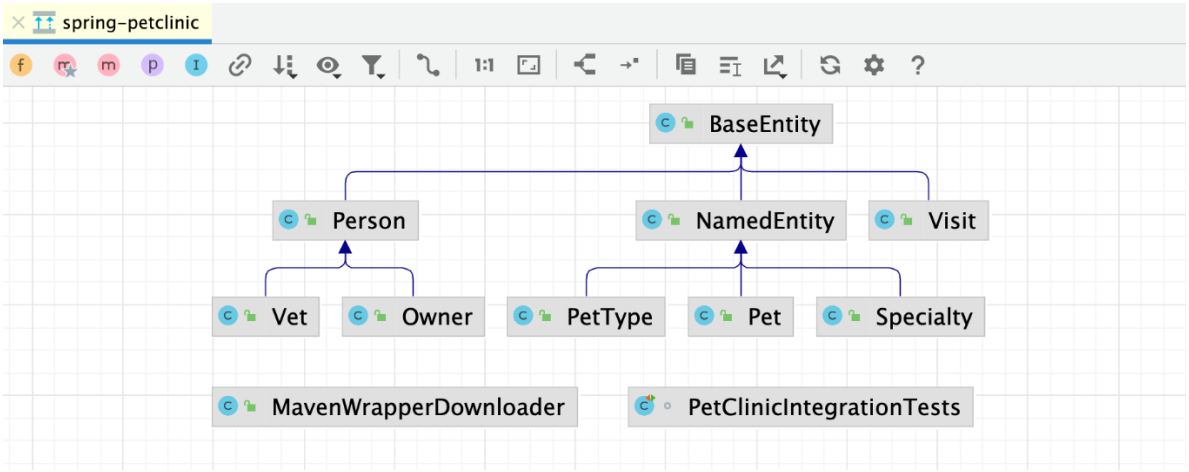
Zaimplementowane są następujące funkcjonalności:

- generowanie diagramu klas,
- wyświetlenie metryk dla projektu,
- wyświetlenie listy klas i interfejsów,
- wyświetlenie listy pól, metod i interfejsów po wybraniu klasy z listy.

Wtyczka napisana została w języku Java z wykorzystaniem API środowiska IntelliJ, biblioteki Swing i PlantUML.

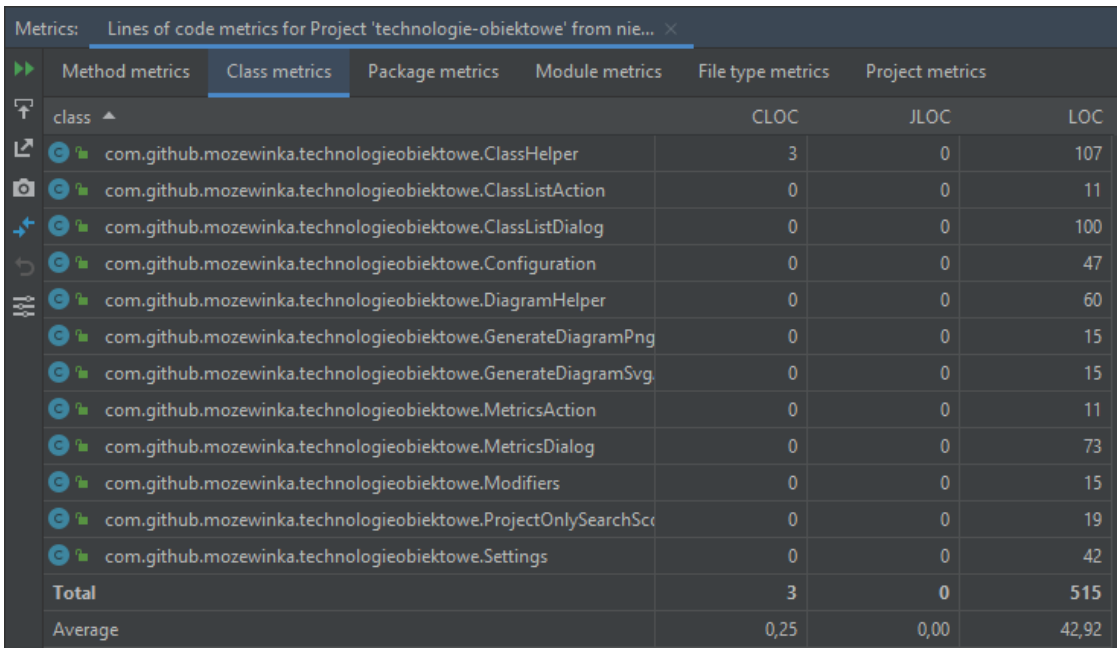
2. Rozwiązania alternatywne

W płatnej wersji środowiska IntelliJ IDEA Ultimate istnieje możliwość generowania diagramów klas. Użytkownik ma możliwość wygenerowania diagramu dla dowolnego projektu oraz dostęp do przycisków umożliwiających zmianę widoczności poszczególnych elementów klas.



Rysunek 2.1 Okno z diagramem klas w IntelliJ IDEA Ultimate.
Źródło: https://www.jetbrains.com/help/idea/class-diagram.html

W przypadku metryk istnieje kilka wtyczek możliwych do zainstalowania z Internetu. Jedną z nich jest MetricsReloaded. Oferuje ona szeroką gamę metryk, od prostych typu liczba linii kodu, liczba klas, interfejsów, metod itp., po bardziej złożone np. zależność między obiektami, brak spójności metod, czy odpowiedzialność danej klasy.

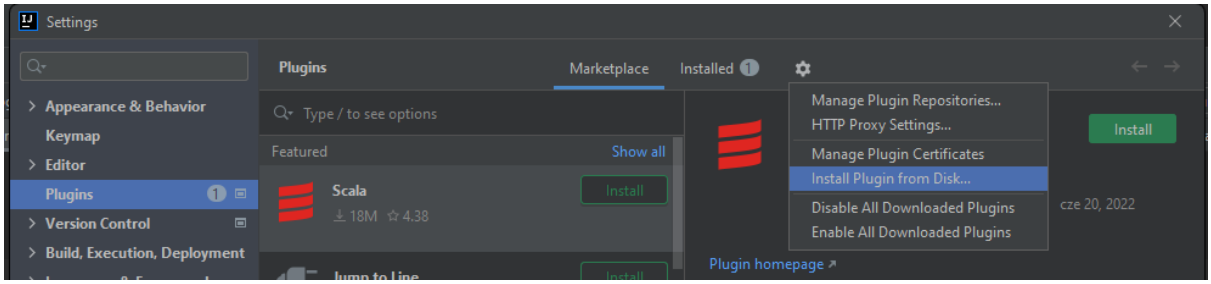


class	CLOC	JLOC	LOC
com.github.mozewinka.technologieobiektowe.ClassHelper	3	0	107
com.github.mozewinka.technologieobiektowe.ClassListAction	0	0	11
com.github.mozewinka.technologieobiektowe.ClassListDialog	0	0	100
com.github.mozewinka.technologieobiektowe.Configuration	0	0	47
com.github.mozewinka.technologieobiektowe.DiagramHelper	0	0	60
com.github.mozewinka.technologieobiektowe.GenerateDiagramPng	0	0	15
com.github.mozewinka.technologieobiektowe.GenerateDiagramSvg	0	0	15
com.github.mozewinka.technologieobiektowe.MetricsAction	0	0	11
com.github.mozewinka.technologieobiektowe.MetricsDialog	0	0	73
com.github.mozewinka.technologieobiektowe.Modifiers	0	0	15
com.github.mozewinka.technologieobiektowe.ProjectOnlySearchSc	0	0	19
com.github.mozewinka.technologieobiektowe.Settings	0	0	42
Total	**3**	**0**	**515**
Average	0,25	0,00	42,92

Rysunek 2.2 Przykładowe metryki linii kodu dla projektu. Wygenerowane przy pomocy MetricsReloaded.

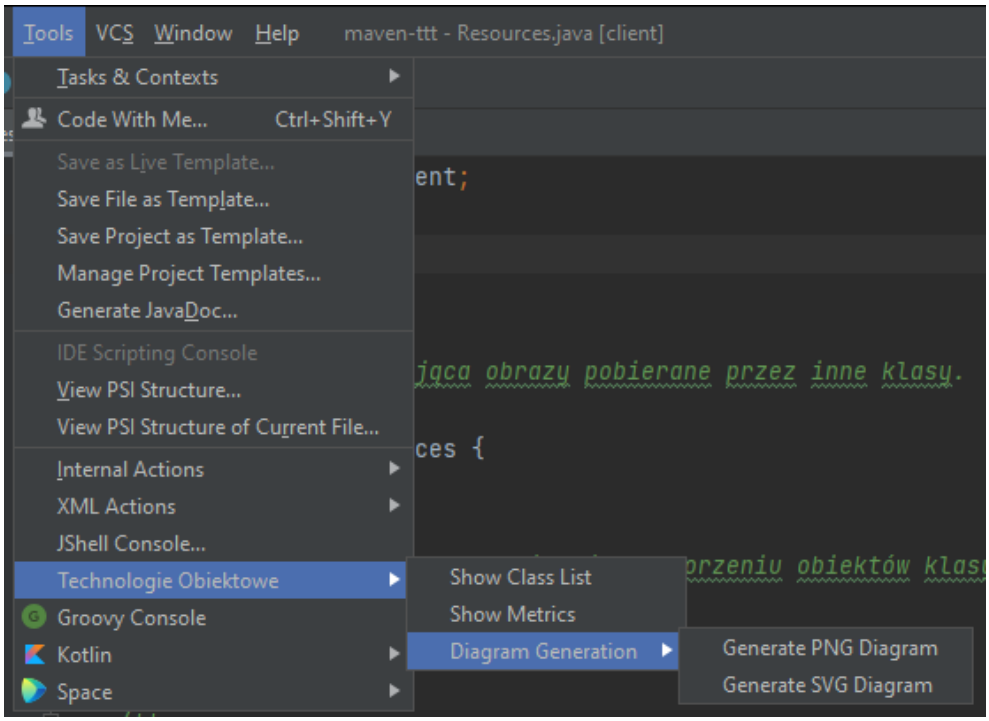
3. Opis rozwiązania

Wtyczkę można zainstalować pobierając najnowszą wersję z platformy GitHub, a następnie wybierając opcję *Install plugin from disk* w ustawieniach wtyczek środowiska IntelliJ.



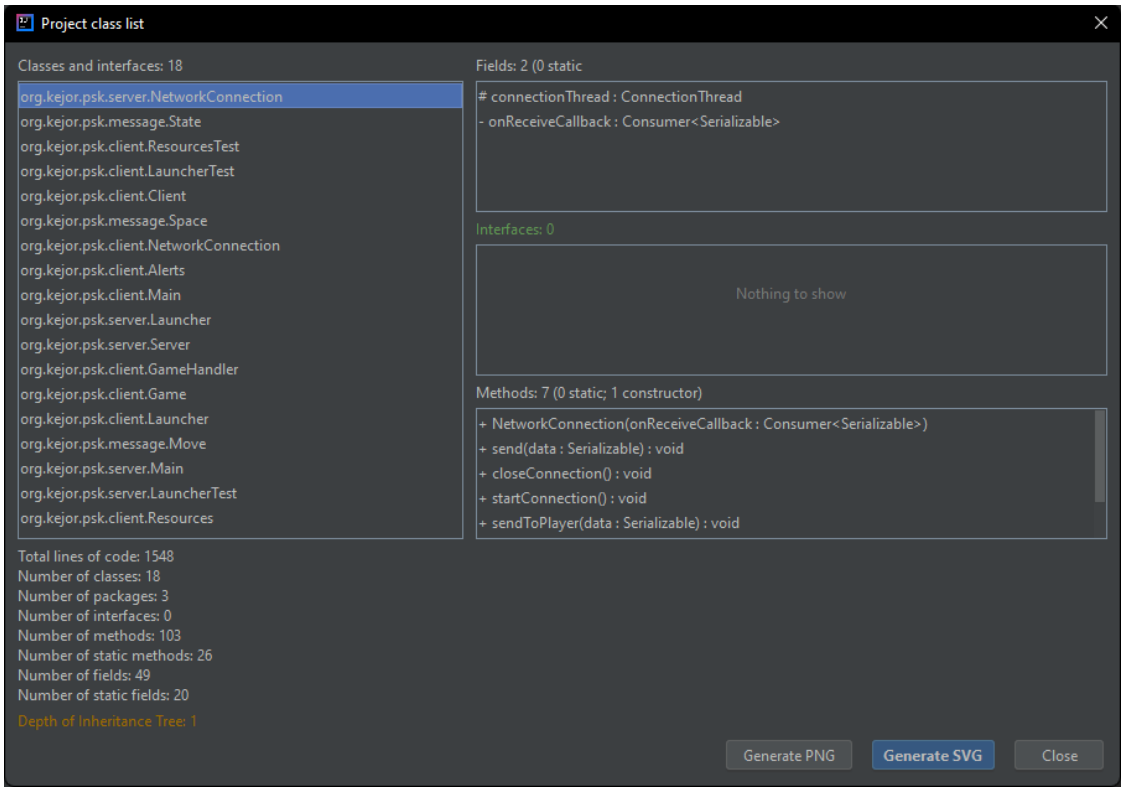
Rysunek 3.1 Ustawienia wtyczek w IntelliJ.

Po instalacji w zakładce „*Tools*" mamy do wyboru nową opcję utworzoną przez wtyczkę. Po jej wybraniu wyświetlają się trzy kolejne opcje, czyli *Show Class List*, *Show Metrics* oraz *Diagram Generation*.



Rysunek 3.2 Menu Tools.

Opcja *Show Class List* wyświetla okno z listą klas i interfejsów projektu. Po wybraniu danej klasy widoczne są również jej pola, interfejsy oraz metody. Dla każdego z tych elementów widoczna jest również ich liczba. Dla pól i metod przedstawiona jest także liczba elementów statycznych. Dla metod dodatkowo wyświetlana jest liczba konstruktorów.



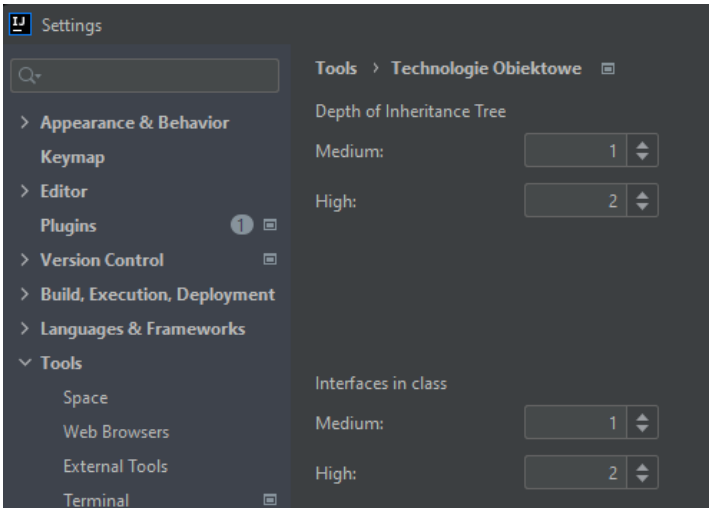
Rysunek 3.3 Okno listy klas.

Pod listą widoczne są metryki oprogramowania dla danego projektu. Metryki, które zostały ujęte w tej wtyczce to:

- liczba linii kodu,
- liczba klas,
- liczba pakietów,
- liczba interfejsów,
- liczba metod,
- liczba metod statycznych,
- liczba pól,
- liczba pól statycznych,
- głębokość drzewa dziedziczenia.

Dla liczby interfejsów zaimplementowanych w klasie oraz dla głębokości drzewa dziedziczenia zastosowano kolorowanie wartości uwzględniając ustawienia użytkownika. Na zrzucie ekranu przyjęto przykładowe wartości gdzie:

- 0 – kolor zielony (poziom niski),
- 1 – kolor pomarańczowy (poziom średni),
- 2 – kolor czerwony (poziom wysoki).

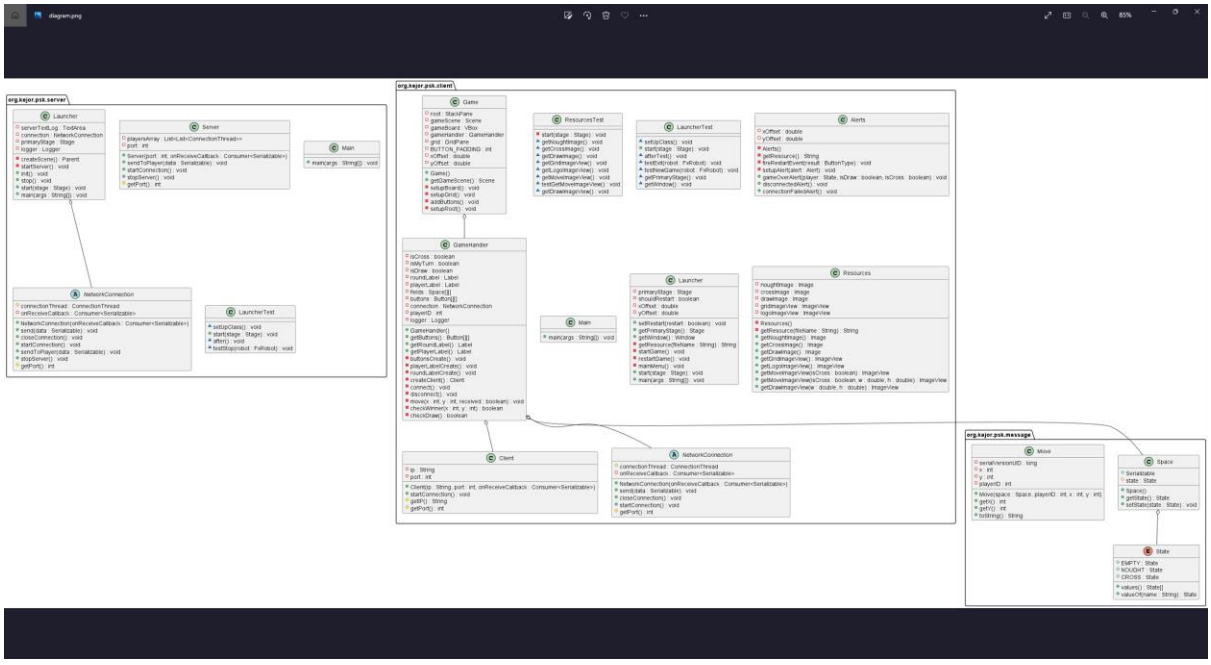


Rysunek 3.4 Ustawienia wtyczki.

Na samym dole okna z listą klas widnieją trzy przyciski:

- *Generate PNG* – wygenerowanie diagramu w formacie PNG,
- *Generate SVG* – wygenerowanie diagramu w formacie SVG,
- *Close* – zamknięcie okna.

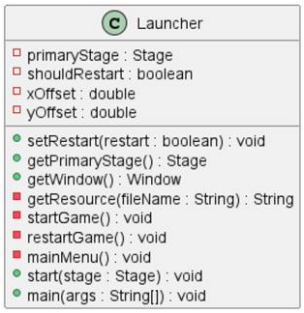
Wybranie opcji generowania diagramu utworzy nowy plik w wybranym formacie, a następnie otworzy go w domyślnym programie dla danego typu w danym systemie.



Rysunek 3.5 Wygenerowany diagram w formacie PNG, widok z programu Windows Photos.

Na diagramie widoczne są wszystkie klasy projektu pogrupowane w pakiety oraz relacje miedzy nimi.

Każda klasa przedstawiona jest w następujący sposób:



Rysunek 3.6 Reprezentacja klasy na diagramie.

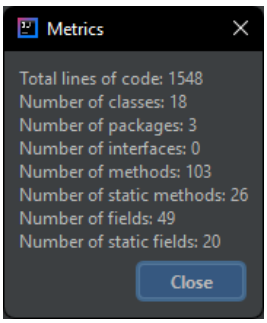
Na samej górze widoczny jest symbol wraz z nazwą. Symbol różni się zależnie od tego, czy jest to klasa (C), klasa abstrakcyjna (A), interfejs (I), czy typ wyliczeniowy (E).

Niżej widnieją pola i metody danej klasy wraz z ich typami oraz symbolami oznaczającymi odpowiednie modyfikatory dostępu. Pola i metody oddzielone są poziomą linią. Dla metody widoczne są także ich parametry.

Ikona dla pola	Ikona dla metody	Widoczność
□	■	private
◇	◆	protected
△	▲	package private
○	●	public

Tabela 1 Ikony dla pól i metod dla odpowiedniej widoczności.

Opcja *Show Metrics* wyświetla okno zawierające wyłącznie metryki dla całego projektu. Są to te same metryki, które były widoczne w oknie z listą klas.



Rysunek 3.7 Okno z metrykami.

4. Implementacja rozwiązania

W projekcie wykorzystano API środowiska IntelliJ do analizy kodu. Pozwala ono na posługiwanie się folderami i plikami jako obiektami PSI (Project Structure Interface). Interfejs ten odpowiedzialny jest za parsowanie plików i tworzenie składniowego oraz semantycznego modelu kodu. Pozwala nam to na operowanie elementami w formie bardziej zbliżonej do refleksji, niż do zwykłego kodu źródłowego. Wykorzystując klasy takie jak *PsiDirectory* oraz *PsiPackage* możemy zebrać wszystkie klasy zawarte w projekcie w postaci obiektów *PsiClass*.

```
private List<PsiClass> getClasses(VirtualFile[] directories, PsiManager psiManager) {
    List<PsiClass> result = new ArrayList<>();
    for (VirtualFile d : directories) {
        PsiDirectory psiDirectory = psiManager.findDirectory(d);
        if (psiDirectory != null)
            addClasses(psiDirectory, result);
    }
    return result;
}

private void addClasses (PsiDirectory psiDirectory, List<PsiClass> classes) {
    PsiPackage psiPackage = JavaDirectoryService.getInstance().getPackage(psiDirectory);
    if (psiPackage != null) {
        if (psiPackage.getSubPackages().length == 0) {
            packages.add(psiPackage);
        }
        PsiClass[] packageClasses = psiPackage.getClasses();
        for (PsiClass psiClass : packageClasses) {
            if (PsiSearchScopeUtil.isInScope(searchScope, psiClass))
                classes.add(psiClass);
        }
        PsiDirectory[] psiSubDirectories = psiDirectory.getSubdirectories();
        for (PsiDirectory sub : psiSubDirectories)
            addClasses(sub, classes);
    }
}
```

Rysunek 4.1 Metody odpowiedzialne za wyszukanie wszystkich pakietów i klas z projektu.

Następnie posiadając takie obiekty mamy dostęp do metod takich jak np. *PsiClass.getFields()*, czy *PsiClass.getMethods()*. W ten sposób możemy uzyskać wszystkie informacje potrzebne do wygenerowania diagramu, czy do stworzenia metryk.

W przypadku diagramów generowane są one na podstawie opisów w składni biblioteki PlantUML. Opisy te przechowywane są w postaci łańcuchów znaków, a następnie diagramy zapisywane są do plików, które mogą być odczytane w systemie.

```java
void generatePng(HashMap<String, PsiClass> classesMap) throws IOException {
    saveDiagramStringToFile(classesMap);
    SourceFileReader reader = new SourceFileReader(new File( pathname: "diagram.plantuml"));
    List<GeneratedImage> list = reader.getGeneratedImages();
    File diagramPng = list.get(0).getPngFile();

    Desktop desktop = Desktop.getDesktop();
    desktop.open(diagramPng);
}

void generateSvg(HashMap<String, PsiClass> classesMap) throws IOException {
    SourceStringReader stringReader = new SourceStringReader(getDiagramString(classesMap));
    final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    DiagramDescription description = stringReader.outputImage(outputStream, new FileFormatOption(FileFormat.SVG));
    description.getDescription();
    outputStream.close();
    final String diagramSvg = outputStream.toString(StandardCharsets.UTF_8);

    FileWriter diagramWriter = new FileWriter( fileName: "diagram.svg");
    diagramWriter.write(diagramSvg);
    diagramWriter.close();

    Desktop desktop = Desktop.getDesktop();
    desktop.open(new File( pathname: "diagram.svg"));
}
```

Rysunek 4.2 Metody odpowiedzialne za wygenerowanie diagramów.

Łańcuch opisujący diagram tworzony jest z pomocą klasy *ClassHelper*, która implementuje metody powalające uzyskać sformatowany tekst na postawie pól, metod i interfejsów danego projektu.

```java
String getDiagramString(HashMap<String, PsiClass> classesMap) {
    StringBuilder diagram = new StringBuilder("@startuml\n\n");

    for (var entry : classesMap.entrySet()) {
        if (entry.getValue().isEnum()) {
            diagram.append("enum ").append(entry.getKey()).append(" {\n");
        } else if (entry.getValue().isInterface()) {
            diagram.append("interface ").append(entry.getKey()).append(" {\n");
        } else if (entry.getValue().hasModifierProperty(PsiModifier.ABSTRACT)) {
            diagram.append("abstract class ").append(entry.getKey()).append(" {\n");
        } else {
            diagram.append("class ").append(entry.getKey()).append(" {\n");
        }

        String[] methods = ClassHelper.methodsToString(entry.getKey(), classesMap);
        for (var method : methods) {
            diagram.append("  ").append(method).append("\n");
        }

        String[] interfaces = ClassHelper.interfacesToString(entry.getKey(), classesMap);
        for (var inter : interfaces) {
            diagram.append("  ").append(inter).append("\n");
        }

        String[] fields = ClassHelper.fieldsToString(entry.getKey(), classesMap);
        for (var field : fields) {
            diagram.append("  ").append(field).append("\n");
        }

        diagram.append("}\n");
    }

    for (var relation : ClassHelper.relationships) {
        diagram.append("  ").append(relation);
    }

    diagram.append("@enduml\n");

    return diagram.toString();
}
```

Rysunek 4.3 Metoda tworząca łańcuch znaków na podstawie klas, pól, metod i interfejsów.

```
@startuml

abstract class org.kejor.psk.server.NetworkConnection {
  + NetworkConnection(onReceiveCallback : Consumer<Serializable>)
  + send(data : Serializable) : void
  + closeConnection() : void
  + startConnection() : void
  + sendToPlayer(data : Serializable) : void
  + stopServer() : void
  # getPort() : int
  # connectionThread : ConnectionThread
  - onReceiveCallback : Consumer<Serializable>
}
enum org.kejor.psk.message.State {
  + values() : State[]
  + valueOf(name : String) : State
  + EMPTY : State
  + NOUGHT : State
  + CROSS : State
}
class org.kejor.psk.client.ResourcesTest {
  - start(stage : Stage) : void
  ~ getNoughtImage() : void
  ~ getCrossImage() : void
  ~ getDrawImage() : void
  ~ getGridImageView() : void
  ~ getLogoImageView() : void
  ~ getMoveImageView() : void
  ~ testGetMoveImageView() : void
  ~ getDrawImageView() : void
}
class org.kejor.psk.client.LauncherTest {
  ~ setUpClass() : void
  + start(stage : Stage) : void
  ~ afterTest() : void
  ~ testExit(robot : FxRobot) : void
  ~ testNewGame(robot : FxRobot) : void
  ~ getPrimaryStage() : void
  ~ getWindow() : void
}

...

  org.kejor.psk.client.Game o-- org.kejor.psk.client.GameHandler
  org.kejor.psk.message.Space o-- org.kejor.psk.message.State
  org.kejor.psk.client.GameHandler o-- org.kejor.psk.client.NetworkConnection
  org.kejor.psk.server.Launcher o-- org.kejor.psk.server.NetworkConnection
  org.kejor.psk.client.GameHandler o-- org.kejor.psk.message.Space
  org.kejor.psk.client.GameHandler o-- org.kejor.psk.client.Client
@enduml
```

Rysunek 4.4 Fragment przykładowego łańcucha opisującego diagram.

Na przykładzie metody *fieldsToString* możemy zobaczyć przykładowy proces tworzenia łańcucha opisującego pola danej klasy.

```
static String[] fieldsToString(String selectedClass, HashMap<String, PsiClass> classesMap) {
    PsiField[] psiFieldsArray = classesMap.get(selectedClass).getFields();
    String[] fieldsArray = new String[psiFieldsArray.length];

    for (int i = 0; i < psiFieldsArray.length; i++) {
        PsiType type = psiFieldsArray[i].getType();
        String name = psiFieldsArray[i].getName();
        fieldsArray[i] = getModifierSymbol(psiFieldsArray[i]) + " " + name + " : " + type.getPresentableText();

        addRelationship(type.getCanonicalText(), selectedClass, classesMap);
    }
    return fieldsArray;
}
```

Rysunek 4.5 Metoda fieldsToString.

Na początku tworzona jest tablica zawierająca pola danej metody (*getFields)*, następnie dla każdego pola uzyskanego w ten sposób sprawdzany jest typ (*getType),* nazwa (*getName*) oraz modyfikator dostępu (*getModifierSymbol*). Te trzy elementy tworzą pojedynczy opis pola.

```java
public static String getModifierSymbol(PsiModifierListOwner element) {
    String symbol = "";
    if (element.hasModifierProperty(PsiModifier.PRIVATE)) {
        symbol = "-";
    } else if (element.hasModifierProperty(PsiModifier.PROTECTED)) {
        symbol = "#";
    } else if (element.hasModifierProperty(PsiModifier.PACKAGE_LOCAL)) {
        symbol = "~";
    } else if (element.hasModifierProperty(PsiModifier.PUBLIC)) {
        symbol = "+";
    }
    return symbol;
}
```

Rysunek 4.6 Metoda zwracająca symbol modyfikatora dostępu.

W przypadku zliczania metryk implementacja opiera się na sumowaniu poszczególnych elementów w pętli.

```java
private void multiCounter(ClassHelper classHelper) {
    classesCount = classHelper.classesMap.size();
    packagesCount = classHelper.packages.size();

    for (PsiClass cls : classHelper.classesMap.values()) {
        linesCount += countLines(cls);

        if (cls.isInterface()) {
            interfacesCount++;
        }

        methodsCount += cls.getMethods().length;
        staticMethodsCount += countStatic(cls.getMethods());

        fieldsCount += cls.getFields().length;
        staticFieldsCount += countStatic(cls.getFields());
    }
}

private int countLines(PsiElement psiElement) {
    String code = psiElement.getText();
    int lines = 0;

    char[] chars = code.toCharArray();
    for (char c : chars) {
        if (c == '\n' || c == '\r') {
            lines++;
        }
    }

    return lines;
}

private int countStatic(PsiMember[] members) {
    int count = 0;
    for (PsiMember member : members) {
        if (member.hasModifierProperty(PsiModifier.STATIC)) {
            count++;
        }
    }
    return count;
}
```

Rysunek 4.7 Metody odpowiedzialne za zliczanie klas, pól, metod, linii kodu oraz elementów statycznych.

Obliczanie głębokości drzewa dziedziczenia (DIT) polega na wywoływaniu metody *getSuperClass*, do momentu, aż nie uzyskamy null.

```
private int getDepth(PsiClass cls) {
    int depth = 0;
    while (cls.getSuperClass() != null) {
        cls = cls.getSuperClass();
        depth++;
    }
    if (depth < settings.getMediumDepthThreshold()) {
        depthLabel.setForeground(JBColor.GREEN);
    } else if (depth < settings.getHighDepthThreshold()) {
        depthLabel.setForeground(JBColor.ORANGE);
    } else {
        depthLabel.setForeground(JBColor.RED);
    }
    return depth;
}
```

Rysunek 4.8 Metoda obliczająca DIT.

5. Podsumowanie

Udało się zrealizować cel projektu, czyli stworzenie wtyczki dla środowiska IntelliJ IDEA, która umożliwiałaby generowanie diagramów klas, a także zaimplementowano dodatkowe funkcjonalności związane z metrykami oprogramowania.

Projekt ten pozwolił na poszerzenie wiedzy na temat technologii obiektowych, metryk oprogramowania oraz wykorzystania API środowiska IntelliJ.

Bibliografia

1. PlantUML, https://plantuml.com/
2. IntelliJ Platform Plugin SDK, https://plugins.jetbrains.com/docs/intellij/welcome.html
3. MetricsReloaded, https://plugins.jetbrains.com/plugin/93-metricsreloaded