1. **Maksymilian Tara 264000** (kierownik)
2. **Łukasz Gawron 264475**
3. **Piotr Kuchnowski 263938**

26.06.2025
rok 2024/2025
semestr letni

Modelowanie i Analiza Systemów Informatycznych
projekt
cz. II

Transformacja modelu testów API do skryptu w języku dziedzinowym

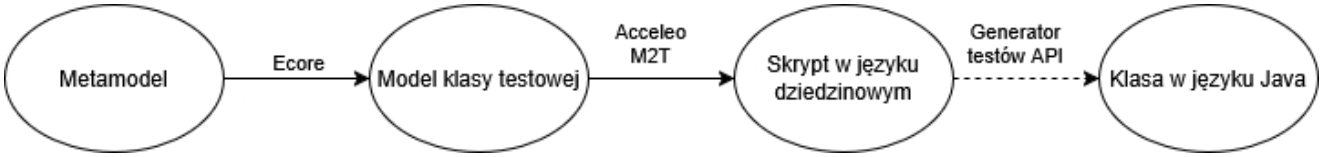
1. Wprowadzenie

Projekt ten jest rozszerzeniem projektu „Generator testów API w bibliotece REST-Assured", którego celem było stworzenie języka dziedzinowego umożliwiającego generowanie kodu w języku Java, służącego do przeprowadzania testów API za pomocą biblioteki REST-Assured.

Celem projektu jest transformacja **M2T** modelu testów do skryptu w języku dziedzinowym. Na testy zawarte w modelu może się składać wiele klas testowych, z których każda może składać się z wielu metod testowych. Otrzymany skrypt powinien być zgodny z zasadami języka i umożliwiać przetłumaczenie go na działającą klasę testową w języku Java za pomocą wcześniej zaimplementowanego generatora.

Dane wejściowe to model klasy testowej w formacie **.xmi**, którego możliwą strukturę opisuje metamodel stworzony za pomocą narzędzia **Ecore**. Plik wyjściowy to plik tekstowy - skrypt w języku dziedzinowym.

Narzędziem do wykonania transformacji M2T jest **Acceleo**.



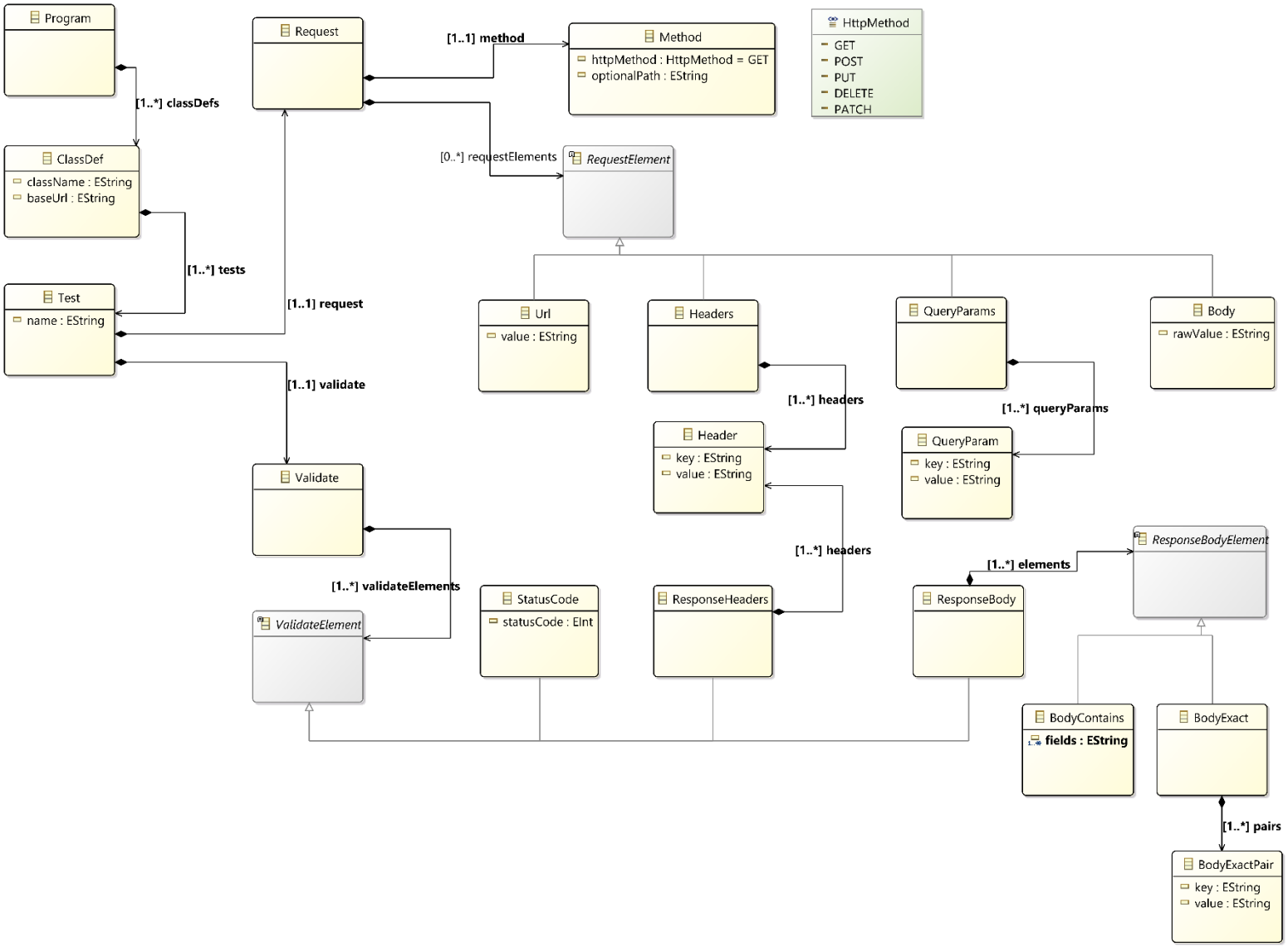
2. Zadania projektu

1. Zbudowanie metamodelu Ecore **MM1** (restAssuredTestGenerator) do transformacji **T1**.
2. Zbudowanie wejściowego modelu **M1** (konkretny model testów w formacie .xmi) do transformacji **T1**.
3. Opracowanie algorytmu transformacji **T1** typu M2T.
4. Implementacja algorytmu transformacji **T1** w postaci projektu Acceleo.
5. Wygenerowanie wyjściowego pliku **P1** (skryptu w języku dziedzinowym) w transformacji **T1** na podstawie modelu **M1**.
6. Przeprowadzenie testów poprawności generowanego skryptu dla nietypowych przypadków, obsługa nietypowych przypadków w skrypcie transformacji M2T.

3. Użyte metamodele

Metamodel MM1 – *restAssuredTestGenerator*, stworzony przy pomocy narzędzia Ecore, definiuje strukturę oraz jakie elementy mogą występować w konkretnych modelach opisujących testy API. Metamodel składa się z klas definiujących struktury jakie oferuje opracowany przez nas język dziedzinowy oraz relacje między nimi.

3.1. Metamodel MM1 – restAssuredTestGenerator



Metamodel składa się z następujących elementów:

1. **Program** – korzeń całego metamodelu, odpowiada głównej regule gramatycznej opracowanego przez nas języka dziedzinowego:
    - Relacje:
        i. *classDefs*: **Program** posiada w kompozycji wiele elementów typu **ClassDef** – definicji klas testowych.
2. **ClassDef** – klasa opisującą pojedynczą klasę testową, w której zawarte są metody testujące.
    - Atrybuty:
        i. *className*: Nazwa klasy, obsługiwana jest również nazwa poprzedzona pakietem, np. *pl.edu.pwr.myExampleTestClass*.
        ii. *baseUrl*: Opcjonalny adres URL pod jakim wystawione jest testowanie API. Służy do globalnej konfiguracji domyślnej.
    - Relacje:

i. tests: Każda klasa testowa posiada w kompozycji wiele metod testowych, inaczej poszczególnych testów.

3. **Test** – reprezentuje metodę testową. Składa się z części odpowiedzialnej za wykonanie zapytania do API oraz część odpowiedzialną za asercje.
    - Atrybuty:
        i. *name*: nazwa metody.
    - Relacje:
        i. *request*: Posiadana w kompozycji klasa odpowiadająca zapytaniu do API.
        ii. *validate*: Posiadana w kompozycji klasa odpowiadająca część asercji.
4. **Request** – reprezentuje jedną z dwóch głównych części testu endpointu API, czyli tę odpowiedzialną za wykonanie zapytania.
    - Relacje:
        i. *method*: Obligatoryjny obiekt typu Method, która pozwala na doprecyzowanie metody http oraz dodatkowego segmentu ścieżki.
        ii. *requestElements*: Kolekcja klas abstrakcyjnych, każda z której stanowi dodatkowe elementy zapytania, takie jak np. nagłówki http, ciało zapytania itd.
5. **Method** – reprezentuje zapytanie http.
    - Atrybuty:
        i. *httpMethod*: Typ Enum, odpowiadający podstawowym metodom http, takie jak GET, POST, PUT, DELETE, PATCH.
        ii. *optionalPath*: Opcjonalny, dodatkowy segment ścieżki, który zostanie dołączony do ścieżki bazowej.
6. **RequestElement** – klasa abstrakcyjna, będąca rodzicem dla klas reprezentujących dodatkowe parametry zapytania http.
    - Klasy dziedziczące: **Url, Headers, QueryParams, Body.**
7. **Url** – reprezentuje adres URL używany w zapytaniu HTTP. Możliwy do ustawienia, jeśli nie ustawiono wcześniej adresu URL testowanego API globalnie dla całej klasy.
    - Atrybuty:
        i. *value*: Wartość adresu URL jako ciąg znaków.
8. **Headers** – reprezentuje zbiór nagłówków HTTP wysyłanych w zapytaniu.
    - Relacje:
        i. *headers*: Kompozycja wielu elementów typu **Header**, reprezentujących pojedyncze nagłówki.
9. **QueryParams** – reprezentuje parametry zapytania (*query parameters*), dołączane do adresu URL.
    - Relacje:

i. *queryParams*: Kompozycja wielu elementów typu **QueryParam**, zawierających pary klucz-wartość.

10. **QueryParam** – reprezentuje pojedynczy parametr zapytania (w adresie URL, która przekazuje dodatkowe dane do serwera).
    - Atrybuty:
        i. *key*: Nazwa parametru zapytania.
        ii. *value*: Wartość przypisana do danego parametru.
11. **Body** – reprezentuje ciało zapytania HTTP.
    - Atrybuty:
        i. *rawValue*: Surowa treść zapytania, najczęściej w formacie JSON, reprezentowana jako string.
12. **Validate** – reprezentuje część testu odpowiedzialną za weryfikację odpowiedzi z serwera.
    - Relacje:
        i. *validateElements*: Kompozycja wielu elementów abstrakcyjnego typu **ValidateElement**, odpowiadających różnym typom asercji. Kolekcja musi zawierać co najmniej jeden element.
13. **ValidateElement** – klasa abstrakcyjna, będąca rodzicem dla elementów sprawdzających poprawność odpowiedzi, takich jak status HTTP, ciało odpowiedzi czy nagłówki.\
    - Klasy dziedziczące: **StatusCode, ResponseHeaders, ResponseBody.**
14. **StatusCode** – reprezentuje asercję sprawdzającą kod statusu HTTP odpowiedzi.
    - Atrybuty:
        i. *statusCode*: Oczekiwany kod odpowiedzi HTTP, np. 200, 404.
15. **ResponseHeaders** – reprezentuje asercję sprawdzającą poprawność nagłówków odpowiedzi.
    - Relacje:
        i. *headers*: Kompozycja wielu elementów typu **Header**, zawierających oczekiwane wartości nagłówków.
16. **Header** – reprezentuje pojedynczy nagłówek HTTP, zarówno w kontekście zapytania, jak i odpowiedzi.
    - Atrybuty:
        i. *key*: Nazwa nagłówka.
        ii. *value*: Wartość nagłówka.
17. **ResponseBody** – reprezentuje zbiór asercji dotyczących treści odpowiedzi z serwera.
    - Relacje:
        i. *elements*: Kompozycja elementów typu **ResponseBodyElement**, takich jak **BodyContains** i **BodyExact**. Kolekcja musi zawierać co najmniej jeden element.
18. **ResponseBodyElement** – klasa abstrakcyjna, będąca bazą dla elementów weryfikujących treść odpowiedzi.

19. **BodyContains** – reprezentuje asercję sprawdzającą obecność określonych pól w ciele odpowiedzi.
    - Atrybuty:
        i. *fields*: Lista ciągów znaków reprezentujących oczekiwane pola (ich obecność w ciele odpowiedzi).
20. **BodyExact** – reprezentuje asercję sprawdzającą zgodność dokładnych wartości pól w odpowiedzi.
    - Relacje:
        i. *pairs*: Lista par klucz-wartość typu **BodyExactPair**, określających oczekiwane wartości w odpowiedzi.
21. **BodyExactPair** – reprezentuje parę klucz-wartość, służącą do porównania konkretnych pól w odpowiedzi z oczekiwaną wartością.
    - Atrybuty:
        i. *key*: Nazwa pola.
        ii. *value*: Oczekiwana wartość pola.
    - Klasy dziedziczące: **BodyContains, BodyExact.**

4. Użyte języki

W projekcie wykorzystano następujące języki plików tekstowych jako dane wejściowe lub wyjściowe:

4.1. DSL do opisu testów API

Nazwa: TestDSL
Wersja: 1.0.0
Użycie: Kod w tym języku jest wyjściem transformacji Acceleo.

Charakterystyka:
TestDSL to dziedzinowy język tekstowy zaprojektowany specjalnie na potrzeby opisywania testów integracyjnych REST API, oparty na uproszczonej składni blokowej. Charakteryzuje się czytelną i zwartą strukturą, umożliwiającą zdefiniowanie klas testowych, zapytań HTTP i odpowiadających im asercji w prosty sposób. Został opracowany w ramach poprzedniego projektu przy użyciu narzędzia ANTLR4.

Przeznaczenie (dziedzina):
TestDSL jest językiem dziedzinowym (Domain-Specific Language – DSL), stworzonym specjalnie na potrzeby testowania REST API. Skupia się na opisie scenariuszy testowych w kontekście żądań HTTP i walidacji odpowiedzi, co czyni go narzędziem dedykowanym dla testerów i programistów Java.

Właściwości i cechy:

- Reprezentacja klas testowych i metod testowych, pogrupowane w logiczne bloki, otoczone nawiasami klamrowymi.
- Obsługa podstawowych metod HTTP (GET, POST, PUT, DELETE, PATCH).
- Możliwość zdefiniowania zapytania: metoda, URL, nagłówki, parametry zapytania, ciało zapytania.
- Możliwość opisania oczekiwanego rezultatu: kod odpowiedzi, nagłówki odpowiedzi, zawartość odpowiedzi.
- Składnia inspirowana językami programowania i formułami testowymi, lecz uproszczona i bardziej czytelna dla użytkowników nietechnicznych.

Składnia w notacji EBNF (ANTLR4):

```
grammar TestGenerator;

program : classDef* EOF;

// Class definition
classDef    : 'CLASS' CLASS_NAME '{' baseUrl? test* '}';

// Test definition
test    : 'TEST' NAME '{' request validate '}';

// Request definition
request : 'REQUEST' '{' method requestElement* '}';
requestElement  : url | headers | queryParams | body;

// HTTP method
method  : 'METHOD' HTTP_METHOD (STRING)?;

// URL
baseUrl : 'URL' STRING;
url     : 'URL' STRING;

// Headers
headers : 'HEADERS' '{' header+ '}';
header  : STRING ':' STRING;

// Query parameters
queryParams : 'QUERY_PARAMS' '{' queryParam+ '}';
queryParam  : STRING '=' STRING;

// Request body
body    : 'BODY' RAW_STRING;

// Assertions
validate       : 'ASSERT' '{' validateElement+ '}';
validateElement : statusCode | responseBody | responseHeaders;

statusCode     : 'STATUS' INT;
responseBody   : (bodyContains | bodyExact)+;
bodyContains   : 'BODY_CONTAINS' STRING+;       // Check if a field exists
bodyExact      : 'BODY_EXACT' bodyExactPair+;   // Check if a field has an exact value
bodyExactPair  : STRING '=' STRING;
responseHeaders: 'HEADER' header+;

// Other tokens
HTTP_METHOD : 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH';
NAME    : [a-zA-Z_][a-zA-Z0-9_]*;
CLASS_NAME    : [a-zA-Z_][a-zA-Z0-9_.]*;
STRING  : '"' (~["])* '"';
RAW_STRING : '"""' .*? '"""';
INT     : [0-9]+;


//NEWLINE : [\r\n]+ -> skip;
NEWLINE : [\r\n]+ -> channel(HIDDEN);

//WS : [ \t]+ -> skip ;
WS : [ \t]+ -> channel(HIDDEN) ;

COMMENT : '/*' .*? '*/' -> channel(HIDDEN) ;
LINE_COMMENT : '//' ~'\n'* '\n' -> channel(HIDDEN) ;
```

Przykład zastosowania języka dziedzinowego:
Pierwszym krokiem jest zdefiniowanie testów za pomocą języka dziedzinowego, poniżej prosty przykład:

```
CLASS pl.edu.pwr.myExampleTestClass {
      URL "http://localhost:8085/api/example"

      TEST myExampleTest {
              REQUEST {
                  METHOD GET "/1"
              }
              ASSERT {
                  STATUS 200
                  BODY_CONTAINS "items"
                  BODY_EXACT
      "id" = "1" "customerName" = "Jan Kowalski"
      "items[0].productName" = "Laptop"
              }
          }
}
```

Poniżej zamieszczony szkielet skryptu przedstawia sposób korzystania z języka dziedzinowego służącego do generowania testów API za pomocą biblioteki REST-Assured.

- Tekst w kolorze czarnym odpowiada stałym wyrażeniom językowym które mają określony format.
- Tekst w kolorze [zielonym] wraz z obejmującymi go kwadratowymi nawiasami włącznie powinien zostać zastąpiony przez użytkownika danymi w zależności od jego potrzeb lub wymagań dotyczących tworzonego testu
- Tekst w kolorze [żółtym] opisuje dodatkowe cechy każdego z wyrażeń, jest to forma komentarza.

URL do testowanego API może zostać ustawiony dla wszystkich testów znajdujących się w danej klasie – parametry połączenia dla testowanego API zostaną ustawione globalnie dla całej klasy w metodzie setup. URL można również ustawiać wewnątrz każdego testu z osobna. Jeżeli zdecydujemy się skorzystać z obydwóch opcji definicji URL, ten zdefiniowany w danym teście bierze priorytet.
W polu METHOD znajdującym się w TEST/REQUEST dostępnych jest 5 metod: GET, POST, PUT, DELETE i PATCH, z których należy wybrać jedną poprzez napisanie jej nazwy w wyznaczonym miejscu.

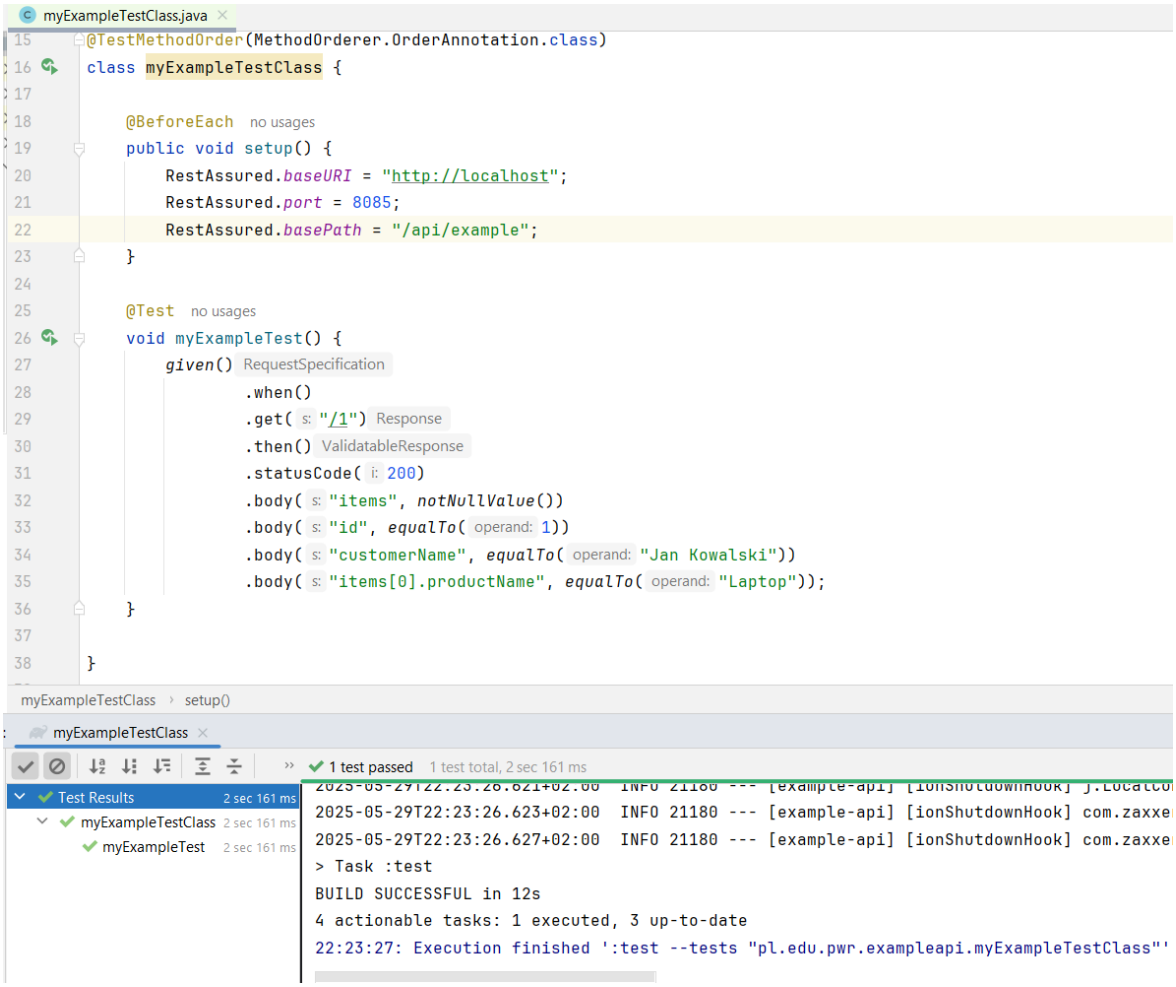
```
CLASS [String - Your class name] {                          [package name is optional]
  URL "[String - Class-wide URL]"                           [optional]

  TEST [String - Your test name 1] {                        [Can have multiple tests in one class]
    REQUEST {
      METHOD *[HTTP method] "[String - URL appendix]"       [method – required, String – optional]
      URL "[String - Test-specific URL]"                    [optional – if URL defined in CLASS]
      HEADERS {                                             [optional]
        "[String]": "[String]"                              [can be repeated any number of times]
      }
      QUERY_PARAMS {                                        [optional]
        "[String]" = "[String]"                             [can be repeated any number of times]
      }
     BODY """                                               [optional]
     [JSON formatted data]
     """
    }
    ASSERT {                                                [order of inner elements is not relevant]
```

```
        STATUS [Number]                                                    [optional]
        BODY_CONTAINS "[String]"                  [optional] [can be repeated any number of times]
        BODY_EXACT "[List element | String]" = "[String]" [optional] [can be repeated any number of times]
        HEADER "[String]": "[String]"             [optional] [can be repeated any number of times]
    }
}
                                                                    *[GET | POST | PUT | DELETE | PATCH]
```

Po sporządzeniu skryptu w języku dziedzinowym, można następnie za pomocą narzędzia (translatora) na jego podstawie wygenerować kod klasy testowej w języku Java:

```java
package pl.edu.pwr;

import io.restassured.RestAssured; // REST-assured library
import io.restassured.http.ContentType; // REST-assured dependency
import org.junit.jupiter.api.*; // JUnit

import static io.restassured.RestAssured.given; // REST-assured given method
import static org.hamcrest.Matchers.*; // Hamcrest matchers

class myExampleTestClass {

    @BeforeEach
    public void setup() {
        RestAssured.baseURI = "http://localhost";
        RestAssured.port = 8085;
        RestAssured.basePath = "/api/example";
    }

    @Test
    void myExampleTest() {
        given()
            .when()
                .get("/1")
            .then()
                .statusCode(200)
                .body("items", notNullValue())
                .body("id", equalTo(1))
                .body("customerName", equalTo("Jan Kowalski"))
                .body("items[0].productName", equalTo("Laptop"));
    }

}
```

Wygenerowany kod jest gotowy do wykorzystania do testów API, na przykład w projekcie Java Spring Boot:



5. Transformacje M2T

5.1. Transformacja Model2DSL

Użyte narzędzia: Acceleo

Metamodele wejściowe: MM1 - restAssuredTestGenerator

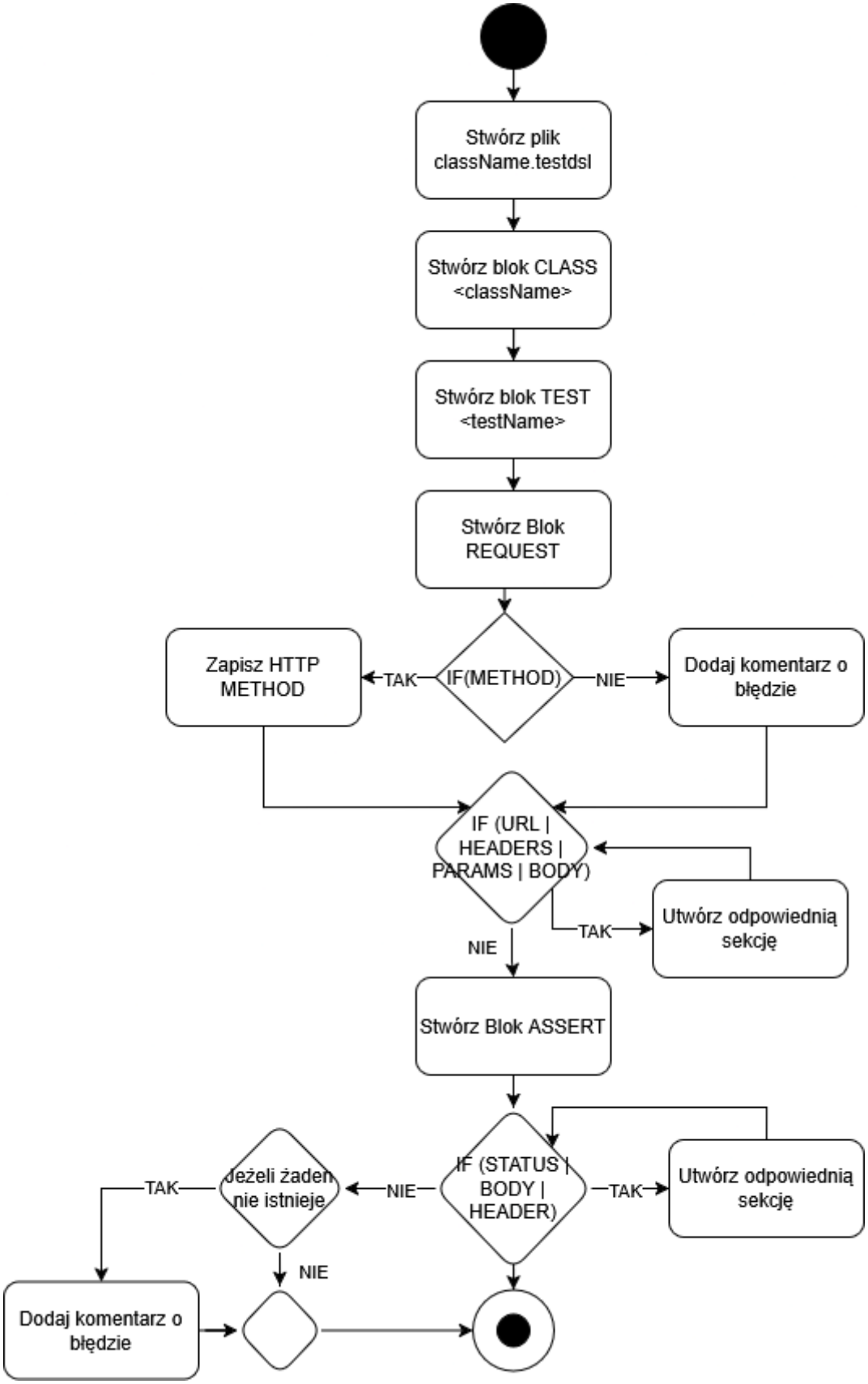
Języki plików wyjściowych: Język dziedzinowy do generowania testów API w języku JAVA

Celem transformacji jest wygenerowanie pliku testowego w specjalnie zaprojektowanym języku dziedzinowym na podstawie modelu testów REST API zapisanego w metamodelu restAssuredTestGenerator. Wygenerowany plik zawiera klasę testową, a w niej zestawy testów zbudowanych z informacji takich jak metoda HTTP, nagłówki, parametry zapytania, ciało zapytania oraz oczekiwane odpowiedzi.

Transformacja:

- Dla każdego obiektu ClassDef tworzy plik .testdsl z nazwą na podstawie className.
- Wewnątrz pliku generuje deklarację klasy z adresem bazowym URL.
- Iteruje przez testy (tests) i generuje strukturę dla każdego testu:
    - Sekcja REQUEST: zawiera metodę, opcjonalną ścieżkę, nagłówki, parametry i ciało.
    - Sekcja ASSERT: zawiera warunki walidacyjne, takie jak status odpowiedzi, zawartość ciała odpowiedzi oraz nagłówki.

Poniżej przedstawiono kolejno diagram czynności obrazujący przebieg algorytmu.



Poniżej zamieszczono kod źródłowy transformacji (.mtl) z projektu Acceleo.

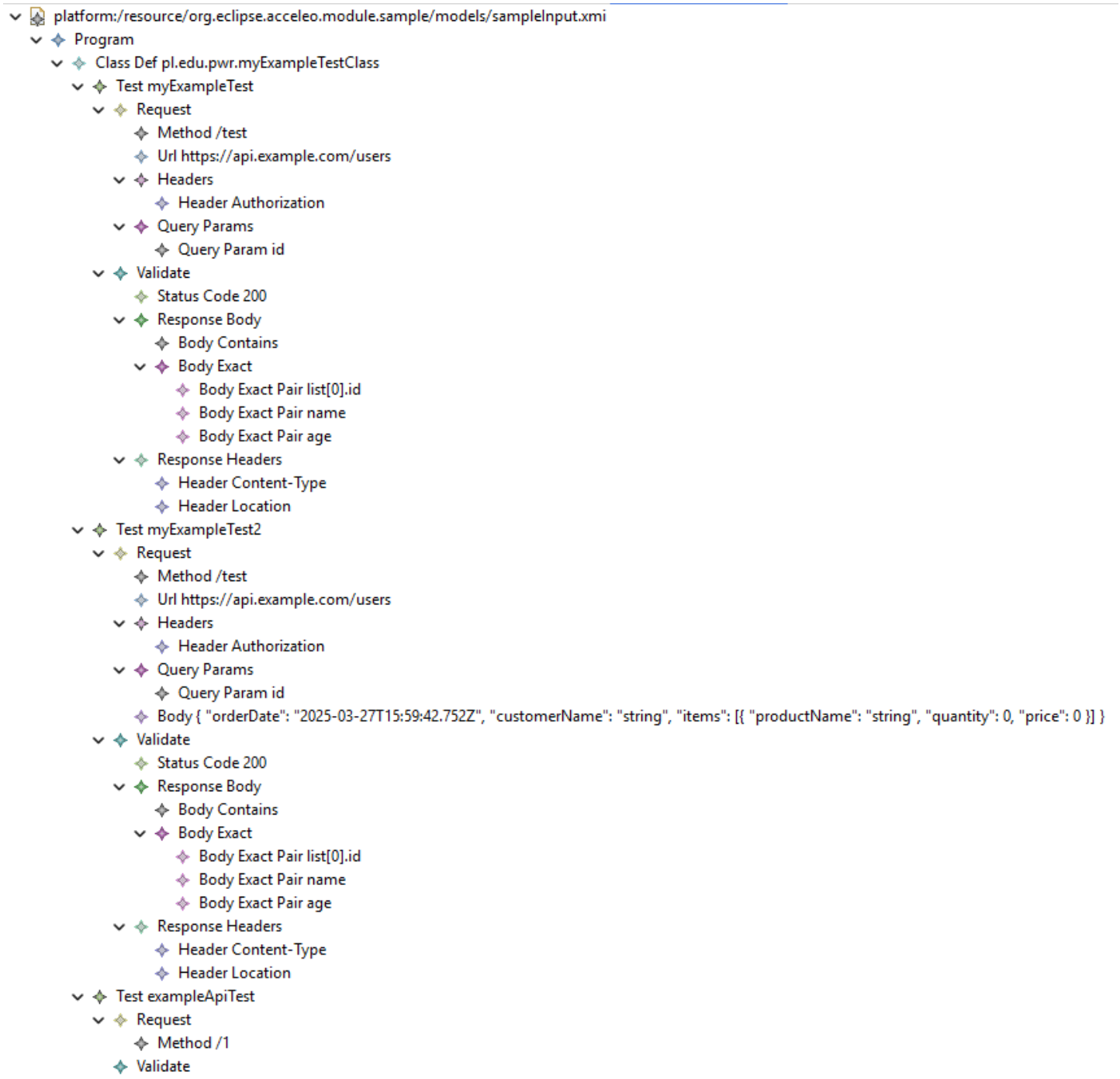
```
[module generate('http://www.example.org/restAssuredTestGenerator')]

[comment encoding = UTF-8 /]

[template public generateTestClass(testClass : ClassDef)]
[comment @main /]
[file (testClass.className.concat('.testdsl'), false, 'UTF-8')]
CLASS [testClass.className/] {
    [if (testClass.baseUrl.oclIsUndefined())]
    [else]
    URL "[testClass.baseUrl/]"

    [/if]
    [for (test : Test | testClass.tests)]
    TEST [test.name/] {
        REQUEST {
            [if (test.request.method.httpMethod.oclIsUndefined())]
            // error: request block must include an element: METHOD
            [else]
            METHOD [test.request.method.httpMethod.toString()/] "[test.request.method.optional-
Path.toString()/]"
            [/if]
            [if  (test.request.requestElements->exists(oclIsTypeOf(Url)))]
            URL "[test.request.requestElements->filter(Url).value/]"
            [/if]
            [if (test.request.requestElements->exists(oclIsTypeOf(Headers)))]
            HEADERS {
                [for (p : Header | test.request.requestElements->filter(Headers).headers)]
                "[p.key/]": "[p.value/]"
                [/for]
            }
            [/if]
            [if (test.request.requestElements->exists(oclIsTypeOf(QueryParams)))]
            QUERY_PARAMS {
                [for (p : QueryParam | test.request.requestElements->filter(QueryParams).query-
Params)]
                "[p.key/]" = "[p.value/]"
                [/for]
            }
            [/if]
            [if  (test.request.requestElements->exists(oclIsTypeOf(Body)))]
            BODY """[test.request.requestElements->filter(Body).rawValue/]"""
            [/if]
        }
        [if (test.validate.validateElements->first().oclIsUndefined())]
        // error: assert block should have at least one element: STATUS|BODY_CONTAINS|BODY_EX-
ACT|HEADER
        [else]
        ASSERT {
            [if (test.validate.validateElements->exists(oclIsTypeOf(StatusCode)))]
            STATUS [test.validate.validateElements->filter(StatusCode).statusCode.toString()/]
            [/if]
            [if (test.validate.validateElements->exists(oclIsTypeOf(ResponseBody)))]
                [if (test.validate.validateElements->filter(ResponseBody).elements->exists(oclIs-
TypeOf(BodyContains)))]
            BODY_CONTAINS
                [for (p : String | test.validate.validateElements->filter(ResponseBody).elements-
>filter(BodyContains).fields)]
                "[p/]"
                [/for]
                [/if]
                [if (test.validate.validateElements->filter(ResponseBody).elements->exists(oclIs-
TypeOf(BodyExact)))]
            BODY_EXACT
                [for (p : BodyExactPair | test.validate.validateElements->filter(ResponseBody).ele-
ments->filter(BodyExact).pairs)]
                "[p.key/]" = "[p.value/]"
                [/for]
```

```
                [/if]
            [/if]
            [if (test.validate.validateElements->exists(oclIsTypeOf(ResponseHeaders)))]
            HEADER
            [for (p : Header | test.validate.validateElements->filter(ResponseHeaders).headers)]
                "[p.key/]": "[p.value/]"
            [/for]
            [/if]
        }
        [/if]
    }
    [/for]
}
[/file]
[/template]
```

6. Wejściowe i wyjściowe modele i pliki

6.1. Plik sampleInput.xmi

Poniżej przedstawiono zrzut ekranu treści przykładowego modelu wejściowego w postaci drzewa.



Poniżej znajduje się treść przykładowego pliku wejściowego XMI.

```
<?xml version="1.0" encoding="UTF-8"?>
<restAssuredTestGenerator:Program
  xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:restAssuredTestGenerator="http://www.example.org/restAssuredTestGenerator"
  xsi:schemaLocation="http://www.example.org/restAssuredTestGenerator restAssuredTestGen-
erator.ecore">
  <classDefs className="pl.edu.pwr.myExampleTestClass" baseUrl="http://lo-
calhost:8080/api/users">
    <tests name="myExampleTest ">
      <request>
        <method optionalPath="/test"/>
```

```
        <requestElements xsi:type="restAssuredTestGenerator:Url" value="https://api.exam-
ple.com/users"/>
        <requestElements xsi:type="restAssuredTestGenerator:Headers">
          <headers key="Authorization" value="Bearer token"/>
        </requestElements>
        <requestElements xsi:type="restAssuredTestGenerator:QueryParams">
          <queryParams key="id" value="123"/>
        </requestElements>
      </request>
      <validate>
        <validateElements xsi:type="restAssuredTestGenerator:StatusCode" status-
Code="200"/>
        <validateElements xsi:type="restAssuredTestGenerator:ResponseBody">
          <elements xsi:type="restAssuredTestGenerator:BodyContains">
            <fields>username</fields>
            <fields>password</fields>
            <fields>email</fields>
          </elements>
          <elements xsi:type="restAssuredTestGenerator:BodyExact">
            <pairs key="list[0].id" value="123"/>
            <pairs key="name" value="John Doe"/>
            <pairs key="age" value="30"/>
          </elements>
        </validateElements>
        <validateElements xsi:type="restAssuredTestGenerator:ResponseHeaders">
          <headers key="Content-Type" value="application/json"/>
          <headers key="Location" value="https://api.example.com/users/123"/>
        </validateElements>
      </validate>
    </tests>
    <tests name="myExampleTest2">
      <request>
        <method optionalPath="/test"/>
        <requestElements xsi:type="restAssuredTestGenerator:Url" value="https://api.exam-
ple.com/users"/>
        <requestElements xsi:type="restAssuredTestGenerator:Headers">
          <headers key="Authorization" value="Bearer token"/>
        </requestElements>
        <requestElements xsi:type="restAssuredTestGenerator:QueryParams">
          <queryParams key="id" value="123"/>
        </requestElements>
        <requestElements xsi:type="restAssuredTestGenerator:Body" rawValue="{ &quot;or-
derDate&quot;: &quot;2025-03-27T15:59:42.752Z&quot;, &quot;customerName&quot;:
&quot;string&quot;, &quot;items&quot;: [{ &quot;productName&quot;: &quot;string&quot;,
&quot;quantity&quot;: 0, &quot;price&quot;: 0 }] }"/>
      </request>
      <validate>
        <validateElements xsi:type="restAssuredTestGenerator:StatusCode" status-
Code="200"/>
        <validateElements xsi:type="restAssuredTestGenerator:ResponseBody">
          <elements xsi:type="restAssuredTestGenerator:BodyContains">
            <fields>username</fields>
            <fields>password</fields>
            <fields>email</fields>
          </elements>
          <elements xsi:type="restAssuredTestGenerator:BodyExact">
            <pairs key="list[0].id" value="420"/>
            <pairs key="name" value="Silas Baraniapupa"/>
            <pairs key="age" value="69"/>
          </elements>
        </validateElements>
        <validateElements xsi:type="restAssuredTestGenerator:ResponseHeaders">
          <headers key="Content-Type" value="application/json"/>
          <headers key="Location" value="https://api.example.com/users/123"/>
        </validateElements>
      </validate>
```

```
    </tests>
    <tests name="exampleApiTest">
      <request>
        <method optionalPath="/1"/>
      </request>
      <validate>
      </validate>
    </tests>
  </classDefs>
</restAssuredTestGenerator:Program>
```

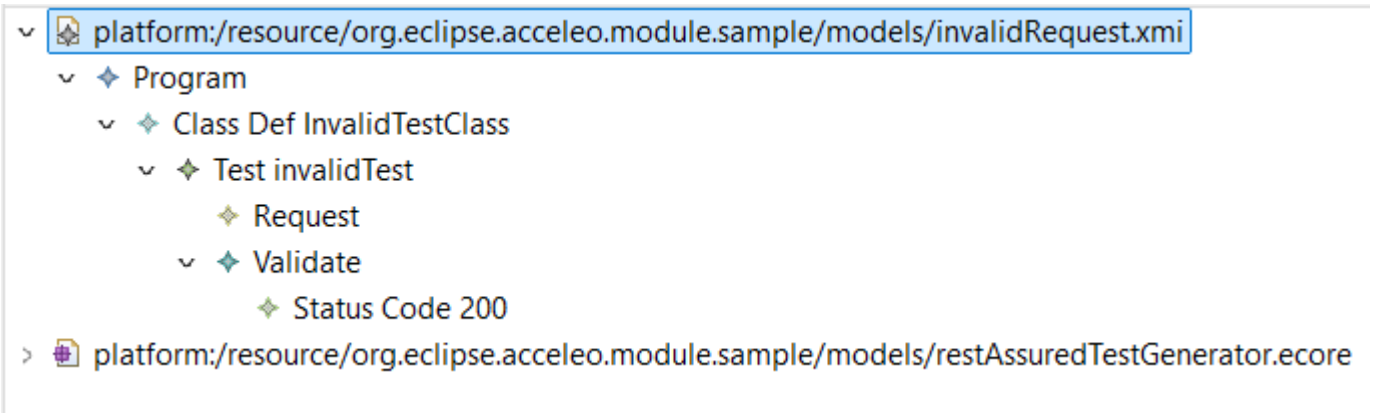
6.2. Plik myExampleTestClass.testdsl

Poniżej przedstawiono plik wyjściowy - wygenerowany kod przy pomocy transformacji Acceleo, na podstawie wcześniej zamieszczonego modelu wejściowego.

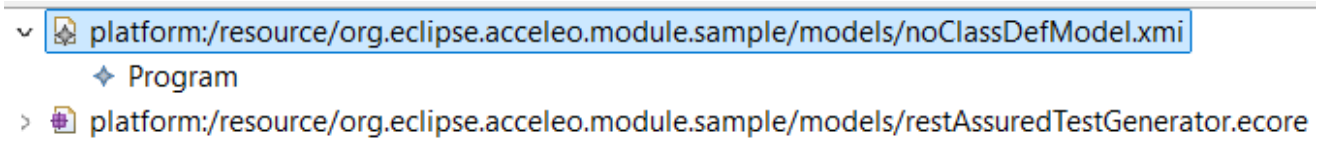
```
CLASS pl.edu.pwr.myExampleTestClass {
    URL "http://localhost:8080/api/users"

    TEST myExampleTest {
        REQUEST {
            METHOD GET "/test"
            URL "https://api.example.com/users"
            HEADERS {
                "Authorization": "Bearer token"
            }
            QUERY_PARAMS {
                "id" = "123"
            }
        }
        ASSERT {
            STATUS 200
            BODY_CONTAINS
                "username"
                "password"
                "email"
            BODY_EXACT
                "list[0].id" = "123"
                "name" = "John Doe"
                "age" = "30"
            HEADER
                "Content-Type": "application/json"
                "Location": "https://api.example.com/users/123"
        }
    }
    TEST myExampleTest2 {
        REQUEST {
            METHOD GET "/test"
            URL "https://api.example.com/users"
            HEADERS {
                "Authorization": "Bearer token"
            }
            QUERY_PARAMS {
                "id" = "123"
            }
            BODY """{ "orderDate": "2025-03-27T15:59:42.752Z", "customerName": "string",
"items": [{ "productName": "string", "quantity": 0, "price": 0 }] }"""
        }
        ASSERT {
            STATUS 200
            BODY_CONTAINS
                "username"
                "password"
                "email"
```

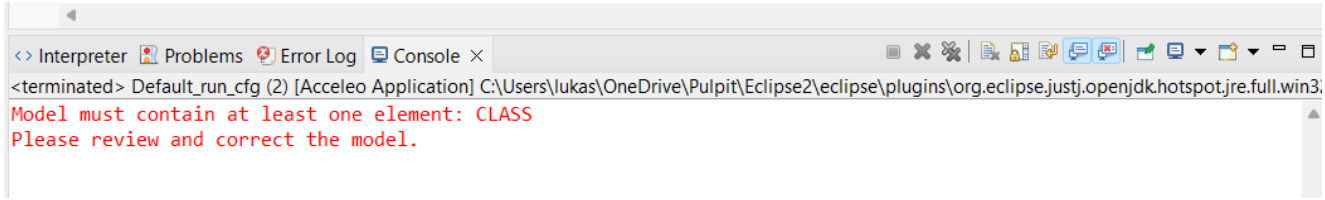
```
            BODY_EXACT
                "list[0].id" = "420"
                "name" = "Silas Baraniapupa"
                "age" = "69"
            HEADER
                "Content-Type": "application/json"
                "Location": "https://api.example.com/users/123"
        }
    }
    TEST exampleApiTest {
        REQUEST {
            METHOD GET "/1"
        }
        // error: assert block should have at least one element: STATUS|BODY_CON-
TAINS|BODY_EXACT|HEADER
    }
}
```

7. Przeprowadzenie testów odporności na błędy




```
CLASS InvalidTestClass {
    TEST invalidTest {
        REQUEST {
            // error: request block must include an element: METHOD
        }
        ASSERT {
            STATUS 200
        }
    }
}
```





```
/**
 * Launches the generation described by this instance.
 *
 * @param monitor
 *            This will be used to display progress information to the user.
 * @throws IOException
 *             This will be thrown if any of the output files cannot be saved to disk.
 * @generated NOT
 */
@Override
public void doGenerate(Monitor monitor) throws IOException {
    if (model != null && model.eResource() != null) {
        List<org.eclipse.emf.ecore.resource.Resource.Diagnostic> errors = model.eRe-
source().getErrors();
        for (org.eclipse.emf.ecore.resource.Resource.Diagnostic diagnostic : errors) {
            System.err.println(diagnostic.toString());
        }
        // Check if there's any classDef element
        boolean foundClassDef = false;
        boolean foundTests = false;

        for (TreeIterator<EObject> it = model.eAllContents(); it.hasNext(); ) {
            EObject obj = it.next();
            if ("ClassDef".equals(obj.eClass().getName())) {
                foundClassDef = true;
            } else if ("Test".equals(obj.eClass().getName())) {
                foundTests = true;
            }

            if (foundClassDef && foundTests) {
                break;
            }
        }

        if (!foundClassDef) {
            System.err.println("Model must contain at least one element: CLASS");
            System.err.println("Please review and correct the model.");
            return;
        }

        if (!foundTests) {
            System.err.println("Warning: The model lacks test definitions. Please add at
least one test, otherwise the generated test class will be empty.");
        }
    } else {
        System.err.println("Model is NULL");
    }

    super.doGenerate(monitor);
}
```

8. Podsumowanie wykonania projektu

W projekcie z powodzeniem opracowano metamodel w języku Ecore na podstawie zdefiniowanego DSL do testów API. Na jego podstawie utworzono przykładowe modele, które posłużyły jako dane wejściowe do transformacji model-na-tekst (M2T). Transformację zrealizowano za pomocą Acceleo, uzyskując poprawne pliki w języku DSL. Projekt został wykonany w pełni i spełnia wymagania dotyczące poprawności strukturalnej oraz jakości wygenerowanego kodu.

Nr i opis zadania (jak w rozdz. 2)	**Wykonano** (TAK, NIE, częściowo)	**Opis** (co wymaga poprawy i dlaczego, czego brakuje, co zasługuje na pochwałę…)

1. Zbudowanie metamodelu Ecore MM1 (restAssuredTestGenerator) do transformacji T1.	TAK	
2. Zbudowanie wejściowego modelu **M1** (konkretny model testów w formacie .xmi) do transformacji **T1**.	TAK	Można by dodać opcjonalnie skrypt automatyzujący budowanie modelu lub nawet integracja z narzędziem graficznym typu Eclipse Sirius.
3. Opracowanie algorytmu transformacji **T1** typu M2T.	TAK	
4. Implementacja algorytmu transformacji **T1** w postaci projektu Acceleo.	TAK	
5. Wygenerowanie wyjściowego pliku **P1** (skryptu w języku dziedzinowym) w transformacji **T1** na podstawie modelu **M1**.	TAK	
6. Przeprowadzenie testów poprawności generowanego skryptu dla nietypowych przypadków, obsługa nietypowych przypadków w skrypcie transformacji M2T.	TAK	Skrypt transformacji obsługuje brak wymaganych elementów poprzez dodanie w ich miejsce w pliku wyjściowym komentarza.