



POLITECHNIKA WARSZAWSKA  
Wydział Elektroniki i Technik Informacyjnych  
Instytut Telekomunikacji

## **PRACA DYPLOMOWA INŻYNIERSKA**

Marcin Maciorowski

**Narzędzie wspomagające projektowanie procesów w języku BPEL  
w architekturze SOA**

Praca wykonana pod kierunkiem  
dra inż. Andrzeja Ratkowskiego

.....  
Ocena pracy

.....  
Podpis Przewodniczącego Komisji

Warszawa, 2014

## **Życiorys**

Urodziłem się 22 września 1988 roku w Radzynie Podlaskim. W 2004 roku rozpocząłem naukę w I Liceum Ogólnokształcącym w Radzynie Podlaskim, gdzie uczęszczałem do klasy o profilu matematyczno-fizyczno-informatycznym. Po uzyskaniu świadectwa dojrzałości w 2007 roku, rozpocząłem studia na Politechnice Warszawskiej na Wydziale Elektroniki i Technik Informatycznych. W trakcie studiów wybrałem specjalizację Systemy informacyjno-decyzyjne prowadzoną przez Instytut Automatyki i Informatyki Stosowanej.

Marcin Maciorowski

## **Streszczenie**

Głównym celem niniejszej pracy inżynierskiej jest stworzenie narzędzia wspierającego tworzenie procesu w języku BPEL. Niniejsza praca zawiera opis najważniejszych z perspektywy omawianego projektu wtyczki elementów języka WSDL, BPEL. Omówiony został projekt wykonanej wtyczki oraz opis graficznego interfejsu użytkownika dostarczonego wraz z narzędziem. Zamieszczona na końcu pracy instrukcja zawiera prezentuje proces instalacji narzędzia.

**Słowa kluczowe:** Java, Eclipse, WSDL, BPEL, EMF.

## **Abstract**

The main purpose of this work is creation of tool supporting modelling business process in BPEL language. The thesis contains description of the most important elements of WSDL and BPEL in the meaning of this project. Project of created plugin and its graphical user interface delivered with a tool description was discussed. At the end of the thesis there were posted installation instruction which contains tool installation process.

**Key words:** Java, Eclipse, WSDL, BPEL, EMF.

## Spis treści.

1. Wstęp.....	5
2. Cel pracy.....	6
3. Układ pracy .....	6
4. Wprowadzenie teoretyczne. ....	8
4.1. Język BPEL + WSDL. ....	8
4.1.1. Język WSDL. ....	8
4.1.2. Język BPEL. ....	11
4.2. EMF (Eclipse Modeling Framework). ....	15
4.2.1. TreeIterator.....	16
5. Wtyczka Eclipse BPEL Designer. ....	17
5.1. Interfejs użytkownika.....	17
5.1.1. Edytor.....	18
5.1.2. Widoki.....	19
5.2. Przykładowy proces BPEL. ....	23
6. Wtyczka generująca instrukcje kopiujące. ....	26
6.1. Transformacja procesu do postaci grafu. ....	27
6.2. Analiza grafu procesu. ....	33
6.3. Aktualizacja instrukcji kopiujących.....	39
6.4. Graficzny interfejs użytkownika. ....	40
6.5. Konfiguracja wtyczki.....	42
7. Testy funkcjonalne. ....	43
8. Podsumowanie.....	51
8.1. Napotkane problemy.....	51
8.2. Możliwości rozwoju.....	52
9. Bibliografia.....	53
10. Załączniki. ....	54
10.1. Płyta CD.....	54
10.2. Instrukcja instalacji wtyczki BPELcg ( <i>BPEL copy generator</i> ). ....	54
10.3. Dokumentacja wtyczki BPELcg ( <i>BPEL copy generator</i> ).....	54

## 1. Wstęp.

Architektura usługowa (*ang. Service Oriented Architecture – SOA*) stanowi popularne podejście do tworzenia systemów informatycznych mające na celu poprawę powiązania strony biznesowej organizacji z jej zasobami. Koncepcja tworzenia systemów informatycznych SOA opiera się na definiowaniu usług, które mają spełniać wymagania użytkownika. Usługa jest określoną funkcją systemu, która działa niezależnie od innych mając zdefiniowany interfejs, który opisuje tę usługę jej oraz położenie. Pod interfejsem usługi kryje się jej implementacja, której konsument nie zna, a jedynie wie „jak” jej używać oraz „gdzie” jej szukać. Koncepcja architektury usługowej abstrahuje od konkretnej technologii, w której powinna zostać zaimplementowana usługa, a dodatkowo jej implementacja może być dowolnie zmieniana pod warunkiem, że nie wpłynie to na zmianę interfejsu usługi.

Udostępniane w ten sposób usługi mogą zostać wykorzystane do stworzenia kolejnej nowej usługi składającej się z tych pierwotnie udostępnianych, a następnie to ona może zostać użyta do budowy następnej itd. Język BPEL umożliwia opisanie procesu biznesowego wykorzystując przy tym wywołania usług zewnętrznych – jest narzędziem do składania zestawu usług sieciowych w tzw. proces, który następnie może zostać udostępniony jako nowa usługa sieciowa.

BPEL, język bazujący na składni XML, umożliwia budowanie procesów biznesowych przy użyciu szeregu narzędzi dostępnych na rynku, m.in. Oracle BPEL Process Manager, Eclipse BPEL Designer, NetBeans BPEL Designer, które to posiadają możliwość tworzenia procesów BPEL za pomocą graficznego interfejsu i budowania jego struktury praktycznie nie ingerując w kod języka, a jedynie używając graficznych reprezentantów aktywności i struktur języka BPEL. Narzędzia tego typu znacznie ułatwiają pracę przy budowaniu usług złożonych z usług atomowych.

Niniejsza praca przedstawia propozycję wtyczki umożliwiającej automatyczne generowanie instrukcji kopiujących wartości zmiennych, używanych w procesie BPEL. Narzędzie będące wygodnym dodatkiem do już istniejącego Eclipse BPEL Designer oferujące funkcję do tej pory niedostępną w istniejących narzędziach używanych do projektowania procesów BPEL. Automatyczna generacja pozwoli na dużą oszczędność czasu poświęcanego na projektowanie procesu biznesowego, różnica szczególnie odczuwalna będzie przy procesach biznesowych bardziej złożonych. Nieodłącznym elementem projektowania procesu jest zaopatrzenie go w instrukcje przepisywania wartości pomiędzy zmiennymi, które do tej pory były dodawane przez projektanta procesu za pośrednictwem interfejsu użytkownika stosowanego do projektowania narzędzia. Powstała w ramach niniejszej pracy wtyczka umożliwia jednym kliknięciem wygenerowanie co najmniej części instrukcji kopiujących, a tym samym oszczędza czas, a tym samym pieniądze.

## 2. Cel pracy.

Celem pracy jest opracowanie narzędzia wspierającego tworzenie procesów biznesowych w języku BPEL. Wsparcie procesu ma się odbywać poprzez automatyzację przepływów danych w procesie stworzonym przy użyciu istniejącego narzędzia do projektowania procesów BPEL jakim jest Eclipse BPEL Designer – wtyczka do zintegrowanego środowiska programistycznego Eclipse. Realizacja celu polega na napisaniu wtyczki stanowiącej rozszerzenie środowiska Eclipse, która zautomatyzuje przepływ danych na podstawie przeprowadzonej analizy procesu. Wtyczka dodatkowo ma umożliwiać projektantowi przegląd wyników analizy oraz edycję wszystkich elementów procesu odpowiedzialnych za przepływ danych. Wtyczka powinna spełniać wymienione wymagania funkcjonalne przy założeniu przedstawionych ograniczeń.

Wymagania funkcjonalne:

- Narzędzie generuje instrukcje kopiujące w procesie BPEL.
- Narzędzie umożliwia przegląd wyników działania generatora instrukcji kopiujących, z wyróżnieniem instrukcji wygenerowanych dla wszystkich bloków przepisywania danych.
- Narzędzie umożliwia dodawanie nowych instrukcji kopiujących do dowolnego bloku przepisywania wartości.
- Narzędzie umożliwia usuwanie instrukcji kopiujących z procesu.
- Narzędzie umożliwia zmianę kolejności instrukcji kopiujących.

Ograniczenia:

- Wszystkie usługi zewnętrzne jak również proces biznesowy są stworzone z wykorzystaniem tej samej konwencji nazewnictwa.
- Generowane instrukcje kopiujące są ograniczone tylko do kopiowania wartości pomiędzy zmiennymi procesu.
- Interfejs graficzny aplikacji umożliwia edycję instrukcji kopiujących wartości pomiędzy zmiennymi procesu.

## 3. Układ pracy

Rozdział 4. Opisuje teoretyczne zagadnienia dotyczące procesów biznesowych w architekturze usługowej – język BPEL, służący do definiowania procesów, język WSDL używany do opisu interfejsu usługi sieciowej.

Rozdział 5. Zawiera opis narzędzia Eclipse BPEL Designer do definiowania procesów biznesowych BPEL dostarczającego graficznej reprezentacji procesu oraz przykładowego procesu zdefiniowanego przy użyciu tego narzędzia.

Rozdział 6. Zawiera projekt aplikacji – wtyczki generującej instrukcje kopiujące w procesie BPEL – z podziałem na poszczególne kroki działania wtyczki od wczytania procesu po prezentację rezultatu działania.

Rozdział 7. Opisuje proces testów funkcjonalnych dostarczonego narzędzia z opisem przykładowych przypadków testowych.

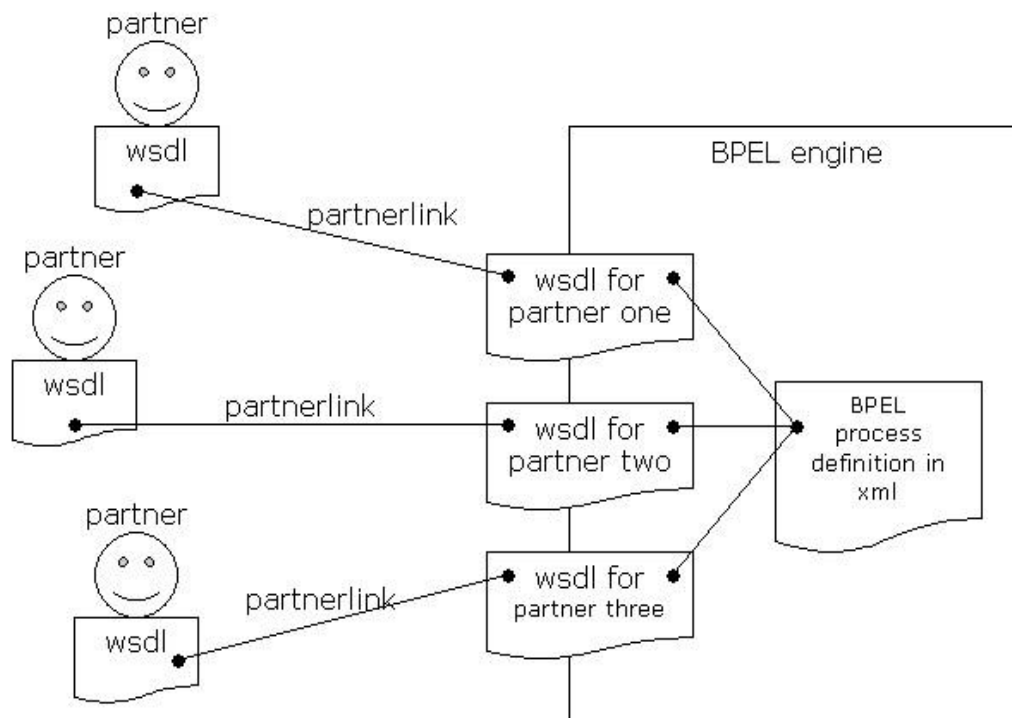
Rozdział 8. Zawiera podsumowanie projektu, prezentuje napotkane problemy w pracach prowadzonych w ramach projektu oraz przedstawia możliwości dalszego rozwoju stworzonego narzędzia.

Rozdział 9. Bibliografia zawierająca spis książek, artykułów, dokumentacji oraz specyfikacji i adresy stron internetowych, w których zawarte są informacje wykorzystane przy pisaniu pracy inżynierskiej.

Rozdział 10. Zawiera listę załączników pracy inżynierskiej, opis płyty CD, instrukcję instalacji stworzonej w ramach pracy inżynierskiej wtyczki oraz dokumentację implementacji stworzonego narzędzia.

## 4. Wprowadzenie teoretyczne.

### 4.1. Język BPEL + WSDL.



Rys. 4.1 BPEL – powiązanie z WS [potrzebne źródło].

#### 4.1.1. Język WSDL.

WSDL (*ang. Web Service Description Language*) jest językiem bazującym na składni XML służącym do opisu interfejsu usługi sieciowej [8]. Specyfikuje sposób dostępu do usługi. Struktura dokumentu w języku WSDL zawiera następujące elementy:

- `<types>` – wprowadza definicje typów złożonych używanych w komunikacji – w elemencie `<message>`,
- `<message>` – definiuje format komunikatów przesyłanych wymienianych pomiędzy konsumentem a usługą sieciową, komunikat składa się z jednego lub wielu elementów podelementów `<part>` identyfikujących poszczególne partie danych składające się na ten komunikat z danymi oraz typy danych do których się odnoszą,
- `<portType>` – element specyfikuje podzbiór operacji wspieranych przez punkt końcowy (*ang. EndPoint*) danej usługi sieciowej,
- `<binding>` – element określający konkretny protokół oraz specyfikujący format danych dla danego elementu `<portType>`,



- `<service>` – element identyfikujący daną usługę sieciową – określa adresn URL oraz nazwę usługi, grupuje jeden lub więcej elementów `<port>` reprezentujących punkty dostępu do usługi.

Na listingu 4.1 przedstawiono strukturę dokumentu WSDL na przykładzie opisu usługi notowań giełdowych. Opisana w dokumencie usługa sieciowa ma jedną operację *GetLastTradePrice*. Operacja używa *GetLastTradePriceInput* jako komunikatu wejściowego zawierającego jeden element *tickerSymbol* typu prostego *string* oraz *GetLastTradePriceOutput* zawiera również jeden element *price* typu *float*. Na potrzeby przykładu, dokument WSDL [10] został odpowiednio zmodyfikowany. Brak bloku `<types>...</types>` implikuje brak definicji typów złożonych w dokumencie opisującym interfejs usługi.

```
<message name="GetLastTradePriceInput">
  <part name="tickerSymbol" element="xsd:string"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="price" element="xsd:float"/>
</message>
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
```

Listing 4.1 Dokument WSDL usługi „Notowania akcji” na podstawie [10].

### Definicja typów.

Dokument opisanego języka WSDL może zawierać jako część swojej struktury element `<types>...</types>` - czyli blok zawierający definicje typów danych ściśle

związanych z wymianą wiadomości [8] w komunikacji konsument-usługa. Dla zachowania uniwersalności języka WSDL – niezależny od platformy – konsorcjum W3C (*ang. World Wide Web Consortium*) w specyfikacji języka WSDL 1.1 do definicji typów złożonych rekomenduje użycie języka XSD (*ang. XML Schema Definition*). XSD jest językiem służącym do definiowania struktury dokumentu XML, zaprojektowany również w celu umożliwienia definiowania struktur typów danych. Definicja typu złożonego opisuje dozwoloną zawartość zdefiniowanego typu przez definicję listy elementów posiadających określony typ. Definicja typu prostego nie wprowadza takiej możliwości, jednak obie definicje prowadzą do utworzenia nowego typu. Na listingu 4.2 przedstawiono przykładową definicję dwóch typów złożonych.

```
<types>
  <schema ...>
    <element name="TradePriceRequest">
      <complexType>
        <sequence>
          <element name="tickerSymbol" type="string"/>
          <element name="volume" type="integer"/>
        </sequence>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <sequence>
          <element name="price" type="float"/>
          <element name="currency" type="string"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</types>
```

Listing 4.2 Definicja typu złożonego na podstawie [10].

Zaprezentowany przykład zawiera definicję elementów typu złożonego, całość deklaracji jest ujęta w blok `<types>...</types>` będący opcjonalnym elementem dokumentu WSDL. Każdy zdefiniowany typ danych zawarty jest w znacznikach `<element>...</element>`. Przedstawiony na listingu 4.2 przykład definiuje typ złożony – niezbędna w przypadku takiej definicji jest ujęta w blok `<complexType>...</complexType>` lista elementów składających się na definiowany typ oraz atrybutów tych elementów – nazwa, typ danych wbudowany w język XSD. W przykładzie z listingu 4.2 pierwsza z definicji o nazwie *TradePriceRequest* składa się z dwóch elementów:

- *tickerSymbol* typu prostego *string*,
- *volume* typu prostego *integer*.

Kolejny z definiowanych typów jest *TradePrice* na który również składają się dwa elementy:

- *price* typu prostego *float*,
- *currency* typu prostego *float*.

Sekcja definicji typów w dokumencie WSDL poza definicją typów złożonych umożliwia również definiowanie typów prostych. Nowe typy proste są wyprowadzone z istniejących typów prostych wbudowanych [12]. Definiowanie nowych typów prostych polega na wprowadzaniu ograniczeń na istniejące typy proste wbudowane poprzez użycie elementu `<restriction>...</restriction>`. Na listingu 4.3 przedstawiono przykład definicji typu prostego na podstawie [12].

```
<types>
  <schema ...>
    <element name="newInteger">
      <simpleType>
        <restriction base="integer">
          <minInclusive value="10000"/>
          <maxInclusive value="99999"/>
        </restriction>
      </simpleType>
    </element>
  </schema>
</types>
```

Listing 4.3 Definicja typu prostego na podstawie [12].

Na zaprezentowanym przykładzie przedstawiono definicję nowego elementu typu prostego – *newInteger*. Stanowi on rozszerzenie wbudowanego typu *integer*, na który nałożono ograniczenie przyjmowanych wartości ( $10000 \leq \text{newInteger} \leq 99999$ ).

W przedstawiony sposób zdefiniowane są typy parametrów wywołań metod publikowanych przez dokumenty WSDL wykorzystywanych przy wywołaniu usług zewnętrznych w procesie BPEL.

#### 4.1.2. Język BPEL.

BPEL (ang. *Business Process Execution Language*, pełna nazwa *Web Services Business Process Execution Language*, *WS-BPEL*) jest językiem do formalnego opisu procesów biznesowych, zaprojektowany w celu zapewnienia wsparcia transakcji biznesowych rozszerzając model interakcji usług sieciowych [14]. Na listingu 4.4 przedstawiono ogólną strukturę podstawowego procesu BPEL. Definicja procesu zamknięta jest w bloku `<process>...</process>` i zawiera następujące elementy:

- `<partnerLinks />` - definicja partnerów procesu, czyli zewnętrznych usług sieciowych biorących udział w zdefiniowanym procesie,

- `<variables />` - definicja listy zmiennych procesu
- `<sequence />` - sekwencja główna procesu BPEL, definiuje faktyczny przebieg procesu – rozmieszczenie wywołań usług sieciowych w procesie.

```
<process>
  <partnerLinks>
    <partnerLink1>...</partnerLink1>
    <partnerLink2>...</partnerLink2>
  </partnerLinks>
  <variables>
    <variable1>...</variable1>
    <variable2></variable2>
  </variables>
  <sequence>
    <receive>
    <assign>
      <copy1>
        <from>...</from>
        <to>...</to>
      </copy1>
      <copy2>
        <from>...</from>
        <to>...</to>
      </copy2>
    </assign>
    <reply>...</reply>
  </sequence>
</process>
```

Listing 4.4 Ogólna struktura procesu BPEL na podstawie [2].

W sekwencji głównej do zdefiniowania logiki procesu biznesowego wykorzystywane są dostarczone przez WS-BPEL aktywności dzielące się na dwie grupy:

- podstawowe (*ang. Basic Activities*),
- strukturalne (*ang. Structured Activities*).

Aktywności podstawowe opisują elementarne kroki w zachowaniu procesu [13]. Przykładowymi elementami języka WS-BPEL określanymi jako aktywności podstawowe są instrukcje:

- *Invoke* – wywołania usługi zewnętrznej,
- *Receive* oraz *Reply* – umożliwia wywołanie operacji procesu,
- *Assign* – aktualizuje zmienne procesu,
- *Throw* – sygnalizuje wystąpienie sytuacji wyjątkowej,
- *Wait* – zatrzymanie wykonania procesu na pewien okres lub do pewnej chwili,
- *Empty* - instrukcja pusta,

- *ExtensionActivity* – dodanie aktywności nowego typu,
- *Rethrow* - przekazanie błędu/wyjątku.

Aktywności strukturalne definiują logikę wykonania – przepływu – procesu, dodatkowo mogą zawierać inne aktywności podstawowe i/lub strukturalne rekursywnie [13]. WS-BPEL dostarcza następujących instrukcji reprezentujących aktywności strukturalne:

- *Sequence* – wykonanie instrukcji procesu sekwencyjnie,
- *If* – instrukcja warunkowa,
- *While* – pętla wykonywana dopóki warunek jest spełniony,
- *RepeatUntil* - pętla wykonywana dopóki warunek nie jest spełniony,
- *Pick* – wybiórcza instrukcja związana ze zdarzeniem,
- *Flow* – instrukcja umożliwiające równoległe wykonania jego wewnętrznych instrukcji,
- *ForEach* – iteracja po wszystkich elementach z dostarczonego zestawu instrukcji.

Niniejsza praca zwraca szczególną uwagę na wykorzystanie trzech elementów języka BPEL, tj. *Variable*, *Invoke*, *Assign*.

### **Zmienna procesu BPEL (*Variable*).**

Element języka wspomagający proces zarządzania stanem procesu pomiędzy akcjami wymiany komunikatów z usługami zewnętrznymi. Stanowi odzwierciedlenie zmiennych znanych w popularnych językach programowania np. Java – służy do przechowywania wartości określonego typu. Na listingu 4.5 przedstawiono przykładowe definicje zmiennych w procesie BPEL. Zadeklarowano dwie zmienne: *creditCardPLResponse* będąca wiadomością typu *doCreditCardCheckingResponse* – typy zmiennych zdefiniowanych w procesie są zdefiniowane w dokumentach WSDL usług biorących udział w procesie – oraz zmienna *creditCardPLRequest* typu *doCreditCardCheckingRequest*.

```
<process>
...
<variables>
  <variable name="creditCardPLResponse"
            messageType="ns0:doCreditCardCheckingResponse">
  </variable>
  <variable name="creditCardPLRequest"
            messageType="ns0:doCreditCardCheckingRequest">
  </variable>
</variables>
...
</process>
```

Listing 4.5 Definicja zmiennej procesu BPEL – na podstawie [3].

Zmienna procesu nie stanowi elementu języka, który wchodzi w skład definicji struktury procesu – nie jest aktywnością BPEL, a jedynie statycznym elementem procesu – dlatego nie posiada graficznej reprezentacji.

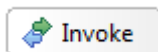
### Instrukcja wywołania usługi (*Invoke*).

Umożliwia w procesie biznesowym wywołanie usługi zewnętrznej oferowanej przez partnera procesu (*partnerLink*). Przedstawione na listingu 4.6 przykładowe użycie instrukcji *Invoke* w procesie BPEL definiuje atrybuty takie jak *name* – nazwę bloku wywołania usługi, *inputVariable* i *outputVariable* określające zmienne zdefiniowane w procesie przy użyciu instrukcji *Variable* typów zgodnych z parametrami usługi, *operation* – nazwę metody wywoływanej w ramach instrukcji *Invoke* zdefiniowaną w dokumencie WSDL partnera procesu, *partnerLink* – określający, który ze zdefiniowanych partnerów procesu bierze udział w wywołaniu oraz opcjonalny *portType*.

```
<process>
...
<partnerLink name="creditCardPL"
  partnerLinkType="tns:creditCardPLT"
  partnerRole="serviceRole"/>
...
<invoke inputVariable="creditCardPLRequest"
  name="checkCreditCard"
  operation="doCreditCardChecking"
  outputVariable="creditCardPLResponse"
  partnerLink="creditCardPL"
  portType="ns0:CreditCardCheckingService"/>
```

Listing 4.6 Wywołanie usługi zewnętrznej – na podstawie [3].

Instrukcja *Invoke* – aktywność BPEL – może być elementem struktury procesu. Graficzne narzędzia służące do projektowania procesów oferują graficzną reprezentację między innymi instrukcji wywołania usług, rysunek 4.1 przedstawia graficzną reprezentację instrukcji w notacji zaproponowanej przez Eclipse w narzędziu BPEL Designer.



Rysunek 4.1 Reprezentacja graficzna instrukcji *Invoke* w notacji Eclipse.

### Instrukcja przepisania wartości zmiennych (*Assign*).

Używana do aktualizowania wartości zmiennych o nowe dane [13]. Aktualizowanie odbywa się poprzez kopiowanie wartości pomiędzy zmiennymi, kopiowanie punktów dostępu do usług pomiędzy partnerami procesu. Instrukcja *Assign* daje również możliwość dodania rozszerzalnych operacji manipulujących danymi przy użyciu rozszerzeń z innych przestrzeni nazw niż WS-BPEL [13]. Blok przepisywania wartości zmiennych może zawierać dowolną liczbę pojedynczych przepisania w postaci instrukcji kopiujących -

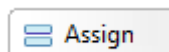
`<copy>...</copy>`, lub operacji aktualizacji danych - `<extensionAssignOperation>...</extensionAssignOperation>`. Na listingu 4.7 przedstawiono przykładową instrukcję przepisywania wartości pochodząca z przykładowego procesu utworzonego na podstawie [3]. Prezentowany blok *Assign* zawiera instrukcję kopiowania wartości pomiędzy zmiennymi typu prostego, które wchodzi w skład złożonych typów. Definicje typów dostarczone są przez dokumenty WSDL usług biorących udział w procesie.

```
<process>
...
<assign name="dataMap4" validate="no">
  <copy>
    <from part="input"
          variable="input">
      <query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:subLang:xpath1.0">
        <![CDATA[tns:cardNumber]]>
      </query>
    </from>
    <to part="request"
        variable="carReservationPLRequest">
      <query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:subLang:xpath1.0">
        <![CDATA[ns3:cardNumber]]>
      </query>
    </to>
  </copy>
  ...
</assign>
...
</process>
```

Listing 4.7 Instrukcja przepisywania wartości zmiennych – na podstawie [3].

Pojedyncza instrukcja kopiująca wartości pomiędzy zmiennymi składowymi typu złożonego posiada blok definiujący element źródłowy kopiowania `<from>...</from>` oraz docelowy `<to>...</to>`. Obie te instrukcje w atrybucie *variable* zawierają nazwę zmiennej zdefiniowanej w procesie – na przedstawionym przykładzie zmienne są typu złożonego – wartość atrybutu *part* określa nazwę elementu wiadomości zdefiniowanej w dokumencie WSDL usługi. Najbardziej zagnieżdżonym elementem jest `<query>...</query>` definiujący wyrażenie w określonym przez parametr *queryLanguage* języku zapytań pobierające wartość konkretnego elementu typu złożonego.

Instrukcja *Assign* jest podobnie jak *Invoke* aktywnością w procesie BPEL, która może składać na strukturę procesu, dlatego ma określony symbol w graficznych reprezentacjach tego języka. Na rysunku 4.2 przedstawiono blok *Assign* w notacji Eclipse.



Rysunek 4.2 Reprezentacja graficzna *Assign* – notacja Eclipse.

## 4.2. EMF (Eclipse Modeling Framework).

EMF jest frameworkiem do budowania narzędzi oraz innych aplikacji bazujących na strukturalnym modelu danych [15]. Specyfikacja modelu jest opisana w języku XMI (*ang. XML Metadata Interchange*), który jest standardem wymiany informacji o meta-danych poprzez użycie XML. Eclipse Modeling Framework pozwala zamienić opisany model na poprawny kod w języku Java. Główne cechy EMF:

- umożliwia generowanie kodu Javy z modeli danych,
- umożliwia korzystanie z innych narzędzi oraz aplikacji bazujących na EMF.

EMF używający do definicji modelu korzystającego z XMI może zostać stworzony na kilka sposobów:

- przez utworzenie dokument XMI manualnie używając edytora tekstu lub XML,
- przez eksport dokument XMI z narzędzia służącego do modelowania,
- przez wygenerowanie modelu na podstawie zestawu interfejsów Javy z odpowiednimi adnotacjami
- przez opis modelu używając XML Schema

Wtyczka służąca do modelowania procesów BPEL, o której mowa w niniejszej pracy – *Eclipse BPEL Designer* – wykorzystuje EMF do opisu modelu języka BPEL.

#### 4.2.1. TreeIterator

*Interface TreeIterator<E>* pochodzący z *org.eclipse.emf.common.util*, czyli dostarczony przez framework EMF stanowi definicję mechanizmu służącego do iteracji po wszystkich węzłach drzewa, w pierwszej kolejności iterując w głąb.

Wykorzystująca framework EMF wtyczka BPEL Designer pozwala na użycie *TreeIterator*, a dzięki temu umożliwia w łatwy sposób przeiterować po wszystkich elementach procesu BPEL.

```
...  
TreeIterator<E> iterator = EMF_model_element_instanceof_E.eAllContents();  
...
```

Listing 4.8 Stworzenie instancji *TreeIterator<E>* - pseudokod.



## 5. Wtyczka Eclipse BPEL Designer.

Eclipse BPEL Designer jest wtyczką integrującą się ze zintegrowanym środowiskiem programistyczne (ang. *Integrated Development Environment* – *IDE*) Eclipse dostarczając własnej perspektywy wspomagającej projektowanie procesów BPEL. Wtyczka dostarcza wsparcie w definiowaniu, edytowaniu, instalacji oraz testowaniu i debuggowaniu procesów WS-BPEL 2.0, czyli języka do definiowania procesów biznesowych opartego o usługi sieciowe, dostarczonego przez konsorcjum OASIS.

Główne cechy wtyczki:

- *Designer* – edytor graficzny (oparty o GEF – Graphical Editing Framework) wprowadzający graficzne oznaczenia elementów procesu BPEL.
- *Model* – reprezentacja modelu BPEL (specyfikacja WS-BPEL 2.0) reprezentowana przez model oparty o EMF (Eclipse Modelling Framework).
- *Validation* – operujący na modelu EMF walidator informujący o błędach i ostrzeżeniach dotyczących procesu BPEL, wynikających ze specyfikacji.
- *Runtime Framework* – zestaw narzędzi umożliwiających instalację oraz wykonanie procesu BPEL.
- *Debug* – zestaw narzędzi umożliwiający śledzenie kolejnych kroków wykonywanego procesu oraz dostarczających obsługę przerw wywołania.

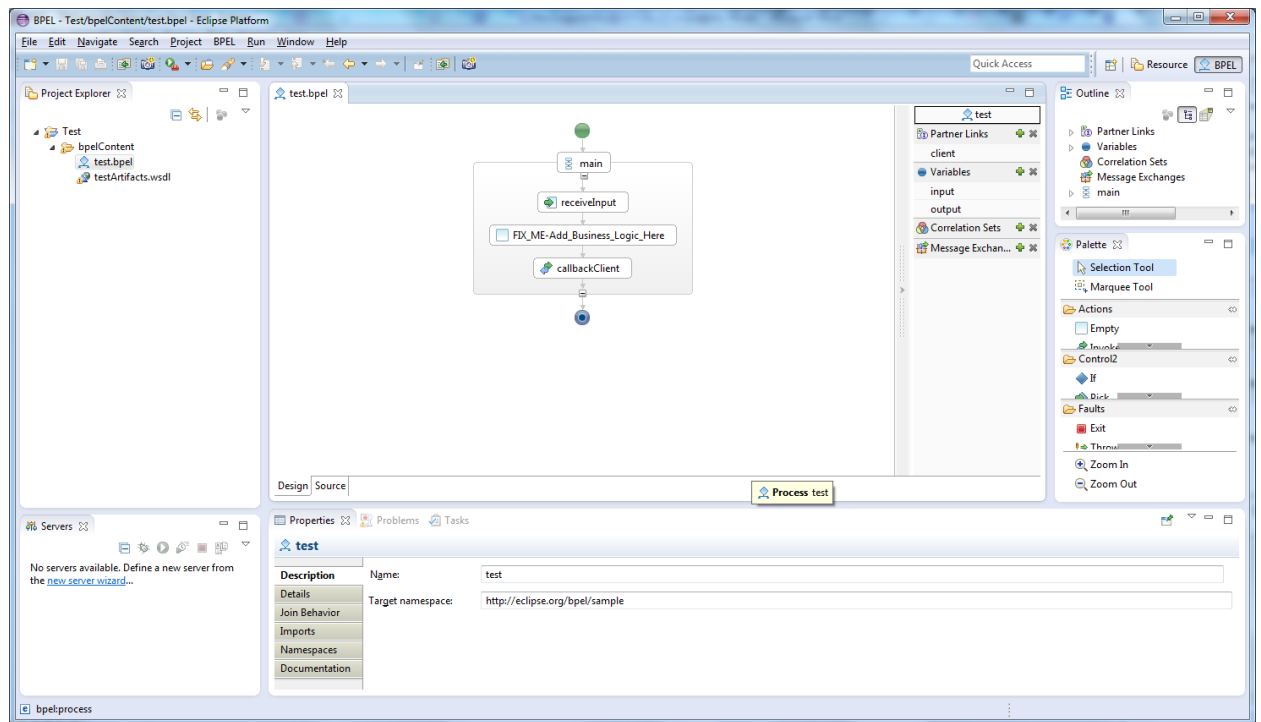
W niniejszej pracy wykorzystywana jest wtyczka Eclipse BPEL Designer w wersji 1.0.3.

### 5.1. Interfejs użytkownika.

Wtyczka Eclipse BPEL Designer dostarcza własnej perspektywy dodając:

- Okno edytora procesów BPEL (umożliwiający również edycję kodu BPEL).
- Widok *Palette* zawierający graficzne elementy będące reprezentacją znaczników języka BPEL biorących udział w procesie.
- Widok *Outline* - schemat procesu w postaci drzewa elementów.
- Widok *Properties* służący do modyfikacji konfiguracji poszczególnych elementów w procesie – specyficzny dla różnych elementów.

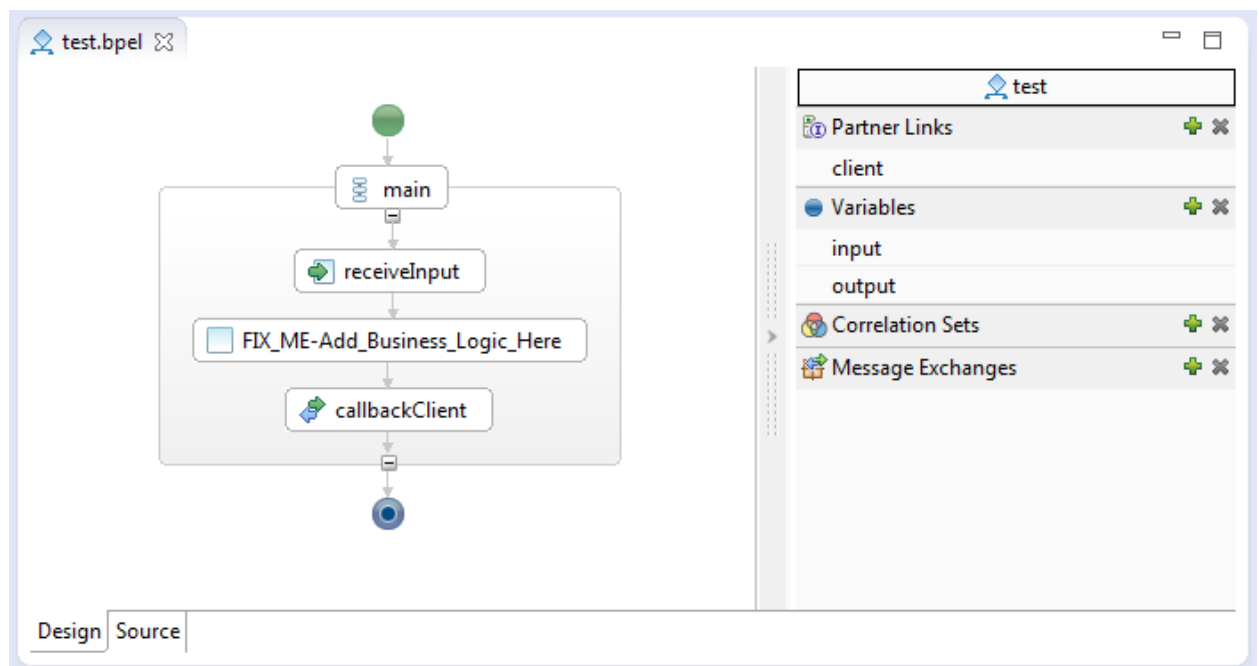
Na rysunku 5.1 przedstawiono wygląd IDE Eclipse w perspektywie BPEL.



Rysunek 5.1 UI wtyczki Eclipse BPEL Designer.

### 5.1.1. Edytor.

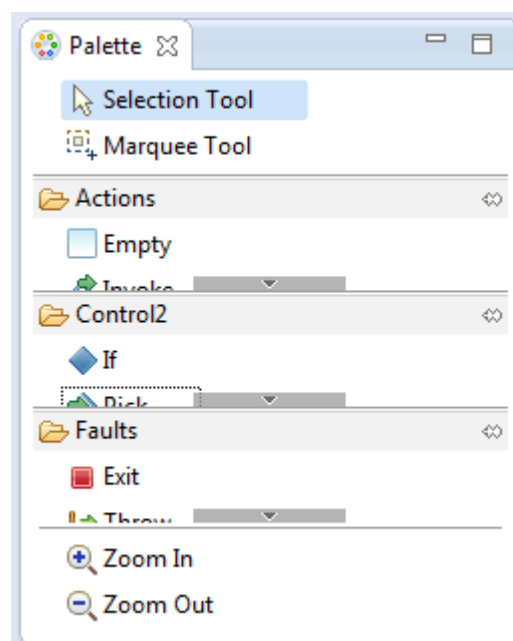
Edytor projektu umożliwia podgląd procesu oraz jego edycję przy użyciu edytora graficznego oraz standardowego edytora kodu BPEL. Edytor graficzny dostarcza możliwość modyfikacji reprezentacji graficznej procesu wizualizującej strukturę głównej sekwencji przebiegu przy użyciu dostarczonych przez wtyczkę Eclipse BPEL Designer graficznych reprezentacji aktywności (*Activity*) BPEL. Graficzny edytor umożliwia również edycję elementów wchodzących w skład procesu takich jak zmienne (*Variables*), połączenia z partnerami biorącymi udział w procesie (*Partner Links*), listę definicji korelacji (*Correlation Sets*) i definicje wiadomości używanych do komunikacji dostawca-konsument usługi (*Message Exchanges*). Ponadto proces może być również edytowany przy użyciu standardowego edytora kodu (zakładka *Source* na rysunku 5.2).



Rysunek 5.2 Edytor procesu BPEL – reprezentacja graficzna.

### 5.1.2. Widoki.

Jednym z widoków jakie dostarcza Eclipse BPEL Designer jest widok palety, zawierający elementy wykorzystywane do zbudowania struktury procesu w oknie edytora.




Rysunek 5.3 Widok palety BPEL Designer

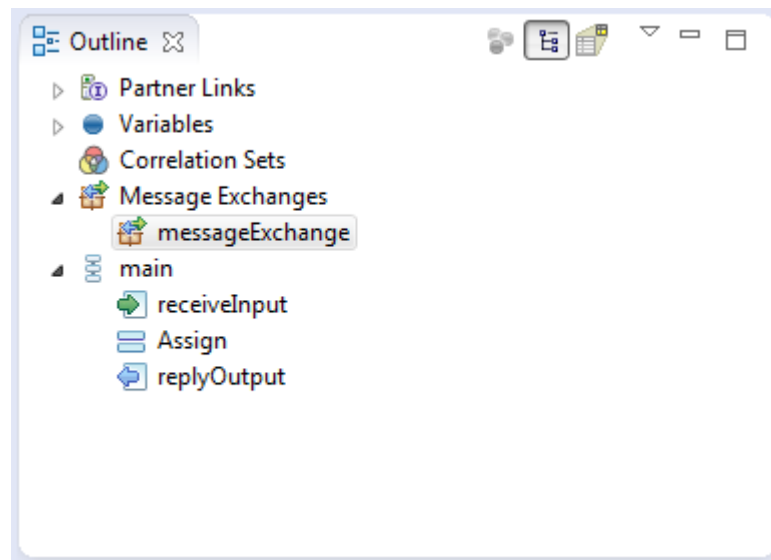
Tabela 5.1 Opis elementów palety BPEL Designer.

Element palety	Opis (odpowiednik w języku BPEL)
Empty	instrukcja pusta ( <i>&lt;empty&gt;</i> )

 Invoke	wywołanie zewnętrznej usługi sieciowej ( <i>&lt;invoke&gt;</i> )
 Receive	odebranie komunikatu wywołania procesu ( <i>&lt;receive&gt;</i> )
 Reply	odpowiedź wysyłana do obiektu wywołującego process ( <i>&lt;reply&gt;</i> )
 Opaque Activity	
 Assign	instrukcja przypisania wartości zmiennej ( <i>&lt;assign&gt;</i> )
 Validate	walidacja wartości zmiennej na podstawie definicji typu ( <i>&lt;validate&gt;</i> )
 If	warunkowy wybór jednej instrukcji do wykonania ( <i>&lt;if&gt;</i> )
 Pick	oczekiwanie na nadejście wiadomości lub przekroczenie czasu oczekiwania ( <i>&lt;pick&gt;</i> )
 While	pętla wykonywana dopóki warunek jest spełniony ( <i>&lt;while&gt;</i> )
 For Each	pętla wykonywana określoną ilość razy ( <i>&lt;forEach&gt;</i> )
 Repeat Until	pętla wykonywana do spełnienia warunku ( <i>&lt;repeatUntil&gt;</i> )
 Wait	oczekiwanie przez określony czas lub do określonego momentu ( <i>&lt;wait&gt;</i> )
 Sequence	zbiór instrukcji uruchamianych sekwencyjnie ( <i>&lt;sequence&gt;</i> )
 Scope	definicja nowej aktywności ( <i>&lt;scope&gt;</i> )
 Flow	zbiór instrukcji wykonywanych jednocześnie ( <i>&lt;flow&gt;</i> )
 Exit	natychmiastowe zakończenie wykonywanego procesu ( <i>&lt;exit&gt;</i> )
 Throw	instrukcja generująca wystąpienie błędu w procesie ( <i>&lt;throw&gt;</i> )
 Rethrow	instrukcja generująca wystąpienie już obsługiwanego błędu ( <i>&lt;rethrow&gt;</i> )
 Compensate	odwrócenie działania wszystkich zakończonych wewnętrznych instrukcji procesu ( <i>&lt;compensate&gt;</i> )

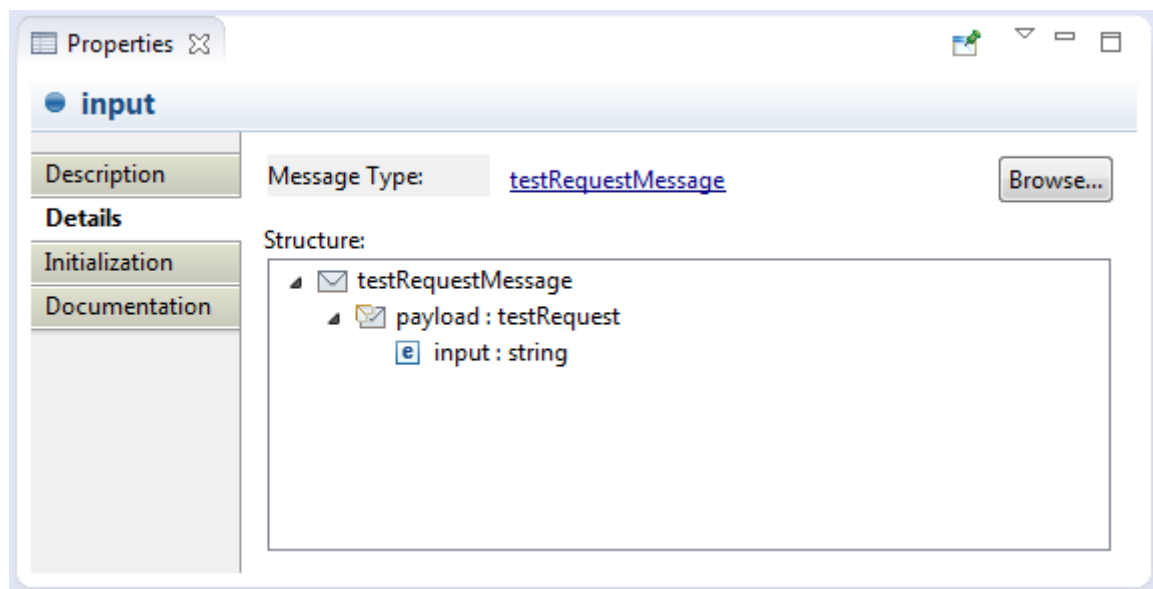
 CompensateScope	odwrócenie działania określonej wewnętrznej instrukcji procesu ( <code>&lt;compensate&gt;</code> )
---	---

Kolejny widok jakim dysponuje Eclipse BPEL Designer jest schemat procesu widoczny na rysunku 5.4. Schemat przy użyciu struktury drzewiastej obrazuje strukturę zagnieżdżeń wszystkich elementów w całym procesie.



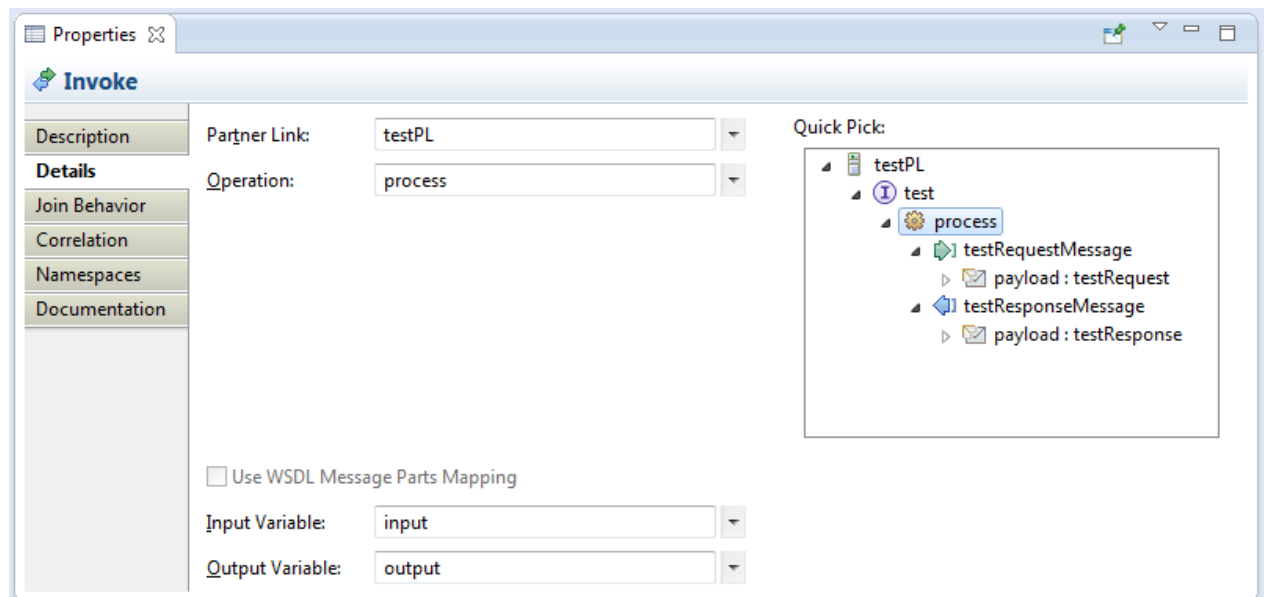
Rysunek 5.4 Widok schematu procesu.

Widok konfiguracji (*Properties*) poszczególnych elementów procesu jest ściśle związany z rodzajem edytowanego elementu. Każdy widok konfiguracyjny zawiera elementy podwidoki wspólne dla wszystkich elementów procesu oraz podwidoki specyficzne dla aktualnie edytowanego procesu. Na rysunku 5.5 przedstawiono szczegółową konfigurację zmiennej procesu umożliwiającą określenie typu danych konfigurowanej zmiennej, który może być typem prostym lub złożonym. Po wyborze typu wyświetlana jest jego struktura, przy czym w przypadku wyboru typu prostego ogranicza się ona do jednego poziomu.



Rysunek 5.5 Szczegółowa konfiguracja zmiennej procesu (*Variable*).

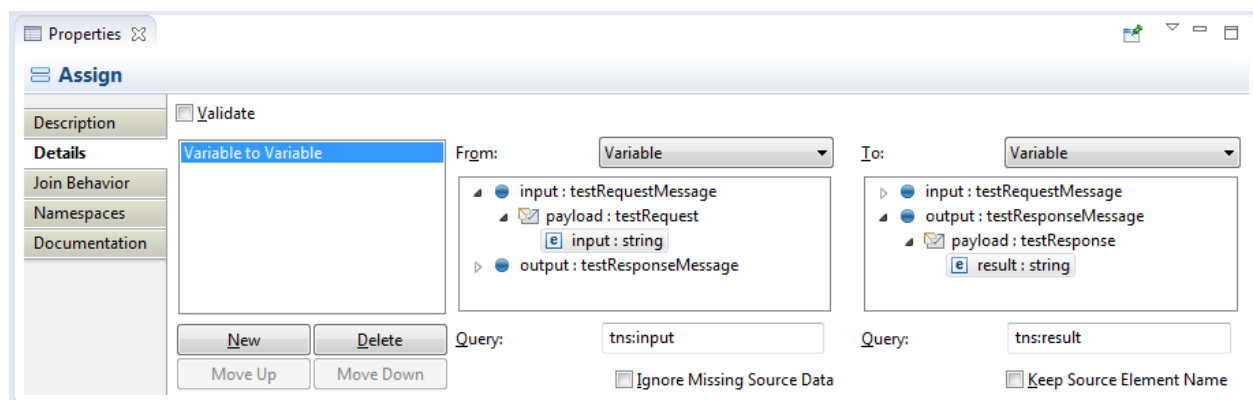
Na rysunku 5.6 przedstawiono widok szczegółowej konfiguracji instrukcji wywołania usługi zewnętrznej umożliwiającą zdefiniowanie partnera procesu oraz operacji wywołanej przez instrukcję. Dokonanie wyboru możliwe jest poprzez sekwencyjne określenie partnera procesu z listy rozwijanej, a następnie operacji – z listy operacji publikowanych przez usługę – z listy rozwijanych, lub przy użyciu sekcji szybkiego wyboru. Po określeniu operacji usługi zewnętrznej wywołanej w konfigurowanej instrukcji konieczne jest również określenie zmiennych procesu, które powinny zostać użyte do wywołania usługi (*Input Variable*) oraz do której zostanie zapisany wynik wywołania operacji usługi zewnętrznej (*Output Variable*).



Rysunek 5.6 Szczegółowa konfiguracja instrukcji wywołania usługi zewnętrznej (*Invoke*).

Na rysunku 5.6 przedstawiono widok szczegółowej konfiguracji instrukcji przepisania wartości zmiennej/zmiennych w procesie. Widok dostarcza możliwość

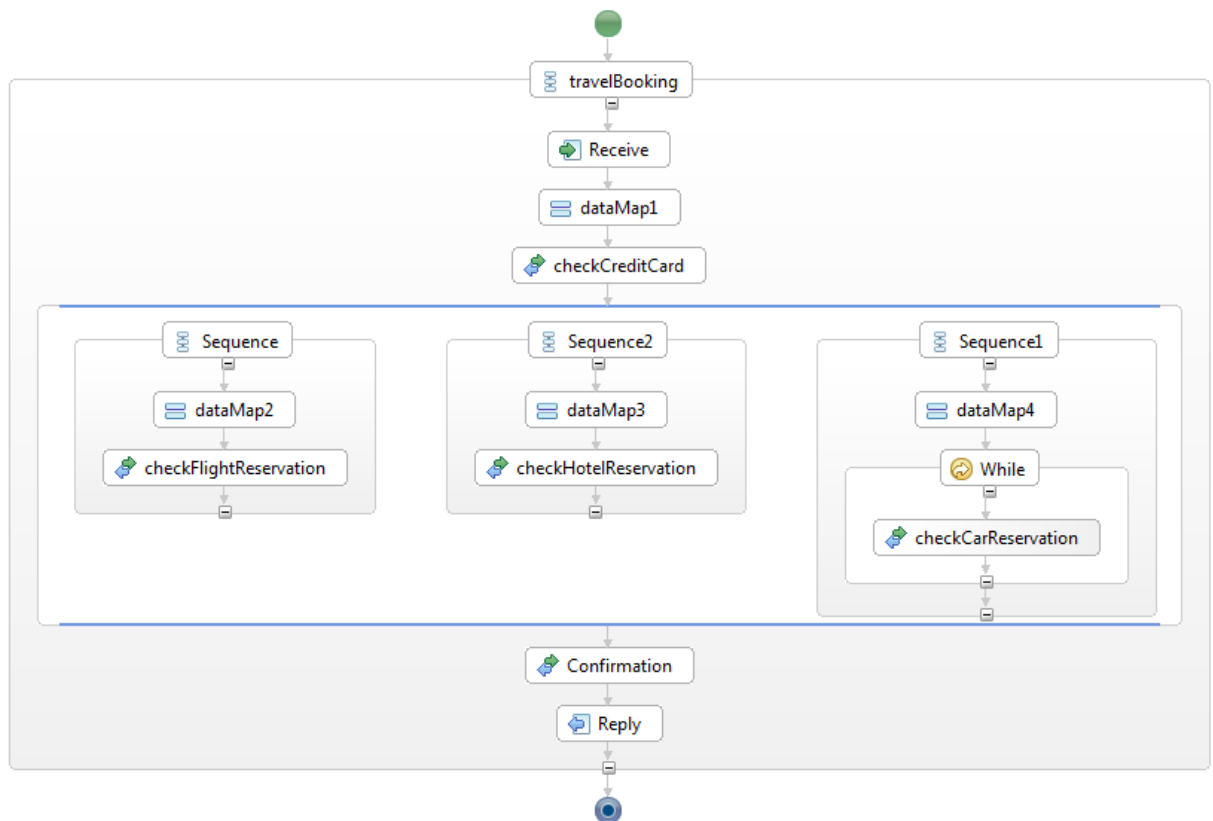
tworzenia/usuwania/zmiany kolejności instrukcji kopiujących w instrukcji przepisania. Możliwe jest również określenie rodzaju elementu z którego oraz do którego wartość ma zostać skopiowana, a następnie określenie tych elementów. Na rysunku zaprezentowano instrukcję przepisania wartości z jedną instrukcją kopiującą. Wybrany typ elementu źródłowego to zmienna procesu, element źródłowy to zmienna typu prostego wchodząca w skład typu złożonego wybranej zmiennej procesu. Podobnie jest w przypadku elementu docelowego.



Rysunek 5.6 Szczegółowa konfiguracja instrukcji przepisania wartości (*Assign*).

## 5.2. Przykładowy proces BPEL.

Na rysunku 5.1 przedstawiony został przykładowy proces BPEL utworzony przy użyciu Eclipse BPEL Designera, na podstawie procesu rezerwacji wycieczki [2]. W momencie wywołania procesu rezerwacji, zostają mu przekazane informacje dotyczące karty kredytowej, celu oraz okresie podróży. Poprzez wywołanie zewnętrznych usług następuje najpierw sprawdzenie dostępności środków – na karcie kredytowej, następnie równoległe rezerwacja lotu, hotelu oraz samochodu. Po zakończeniu równoległych przebiegów do konsumenta usługi trafia żądanie potwierdzenia rezerwacji, po którym zostaje wysłana informacja o poprawnym zakończeniu procesu.



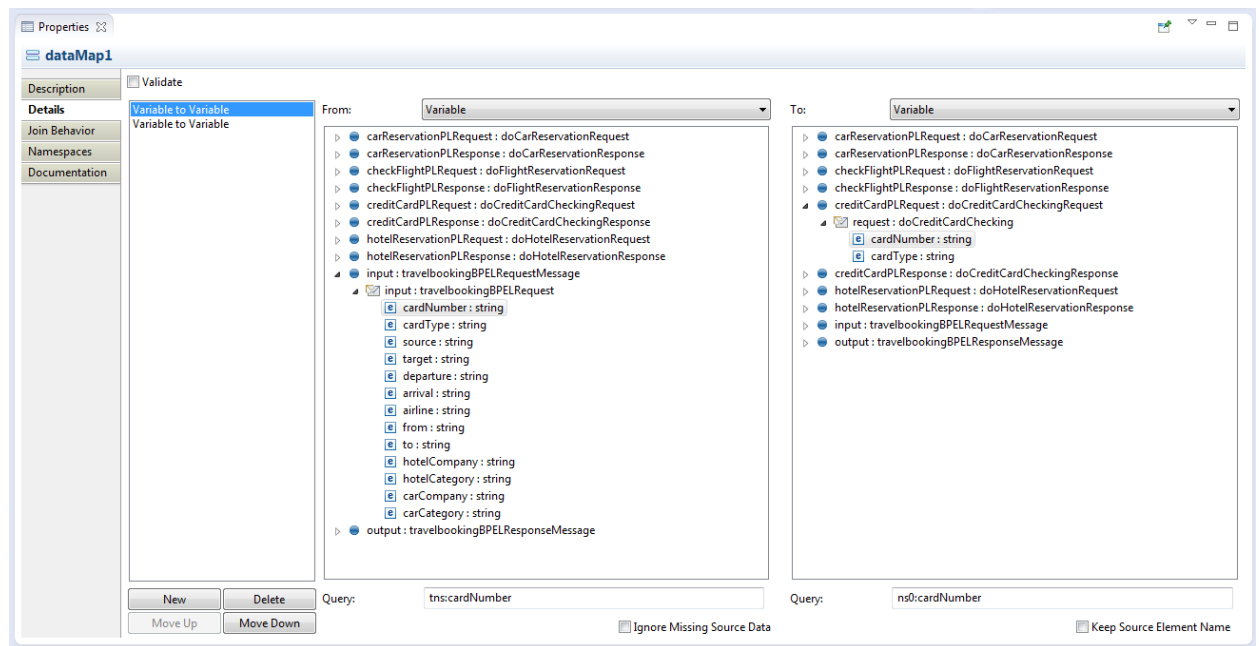
Rysunek 5.1 Przykładowy proces BPEL utworzony w Eclipse BPEL Designer – *travelBooking*

W przedstawionym procesie występują trzy bloki przepisania danych (*Assign*):

- *dataMap1* – zawiera instrukcje kopiujące odpowiednie wartości wejściowe procesu do zmiennych będących elementami parametru wywołania usługi *checkCreditCard*.
- *dataMap2* – analogicznie do *dataMap1* dla usługi *checkFlightReservation*.
- *dataMap3* – analogicznie do *dataMap1* dla usługi *checkHotelReservation*.
- *dataMap4* – analogicznie do *dataMap1* dla usługi *checkCarReservation*.

Na rysunku X przedstawiona została konfiguracja instrukcji kopiujących dane na przykładzie bloku przypisywania danych *dataMap1*. Sekcja zawiera listę instrukcji kopiujących opisanych jako para typów (elementu źródłowego oraz elementu docelowego dla instrukcji kopiowania) oraz dwie listy zmiennych o zasięgu nie mniejszym niż aktualnie konfigurowany blok *Assign*. W obu listach *From* oraz *To* zaznaczono zmienne odpowiednio źródłowa i docelowa.





Rysunek 3. Sekcja *Properties* bloku przepisywania danych *dataMap1*, zakładka z listą instrukcji kopiujących.

Na rysunku 4. Przedstawiono wygenerowany przez Eclipse BPEL Designer kod w języku BPEL odpowiadający konfiguracji przedstawionej na rysunku 3. dla bloku *dataMap1*.

```

69<bpel:assign validate="no" name="dataMap1">
70  <bpel:copy>
71    <bpel:from part="input" variable="input">
72      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
73        <![CDATA[tns:cardNumber]]>
74      </bpel:query>
75    </bpel:from>
76    <bpel:to part="request" variable="creditCardPLRequest">
77      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
78        <![CDATA[ns0:cardNumber]]>
79      </bpel:query>
80    </bpel:to>
81  </bpel:copy>
82  <bpel:copy>
83    <bpel:from part="input" variable="input">
84      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
85        <![CDATA[tns:cardType]]>
86      </bpel:query>
87    </bpel:from>
88    <bpel:to part="request" variable="creditCardPLRequest">
89      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
90        <![CDATA[ns0:cardType]]>
91      </bpel:query>
92    </bpel:to>
93  </bpel:copy>
94</bpel:assign>

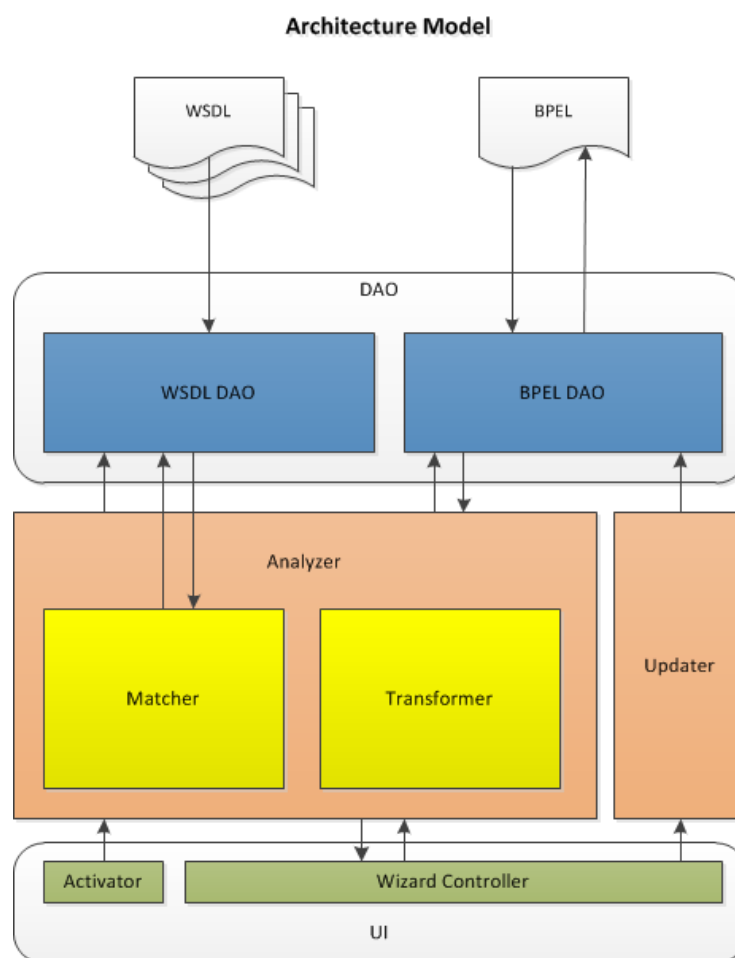
```

Rysunek 4. Kod BPEL bloku *dataMap1*.

## 6. Wtyczka generująca instrukcje kopiujące.

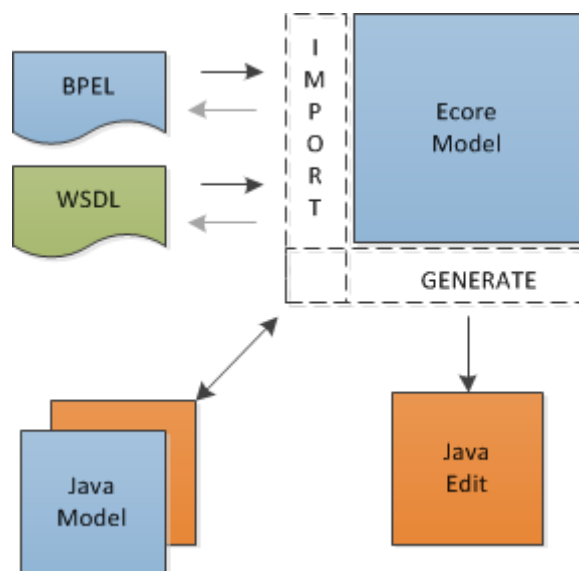
W ramach niniejszej pracy dyplomowej inżynierskiej została opracowana wtyczka generująca instrukcje kopiujące dla instrukcji przepisania wartości zmiennych w procesie BPEL. Wtyczka stanowi rozszerzenie IDE Eclipse bazując na wyniku analizy procesu przetransformowanego z postaci EMF – w jakiej znajduje się po wczytaniu – do postaci grafu. W podstawowym scenariuszu istnieje plik procesu BPEL, który zostaje wczytany przy użyciu obiektu wczytującego. Następnie wczytany proces zostaje przetransformowany do postaci grafu. Graf poddany zostaje analizie przeprowadzonej przez analizator. Wyniki analizy w postaci listy instrukcji kopiujących zmapowanych na konkretne instrukcje przepisania wartości zmiennych występujących w procesie zostają wykorzystane przez aktualizator do załadowania do procesu.

Na rysunku 6.1 przedstawiono model architektury omawianej wtyczki. Dostępem do danych – opublikowane pliki opisujące usługi sieciowe oraz proces biznesowy BPEL – zajmują się obiekty DAO (*Data Access Object*). Komponenty DAO zostały zastosowane w celu oddzielenia warstwy dostępu do danych od logiki znajdującej w elemencie *Analizer* oraz elemencie *Updater* i warstwy prezentacji przedstawionych w modelu architektury.



Rysunek 6.1 Model architektury wtyczki generującej instrukcje kopiujące.

Zaprezentowany schemat architektury przedstawia dwa obiekty dostępu do danych. BPEL DAO użyty jest do wczytania oraz zapisu procesu biznesowego z i do pliku zawierającego jego definicję (\*.bpel) do modelu BPEL Designer przy użyciu modelu EMF. Drugim obiektem dostępu do danych jest WSDL DAO używany do wczytania plików definiujących opis usług sieciowych biorących udział w procesie BPEL. Na rysunku 6.2 przedstawiono schemat importu plików \*.bpel oraz \*.wsdl przy użyciu modeli EMF do modeli Java.

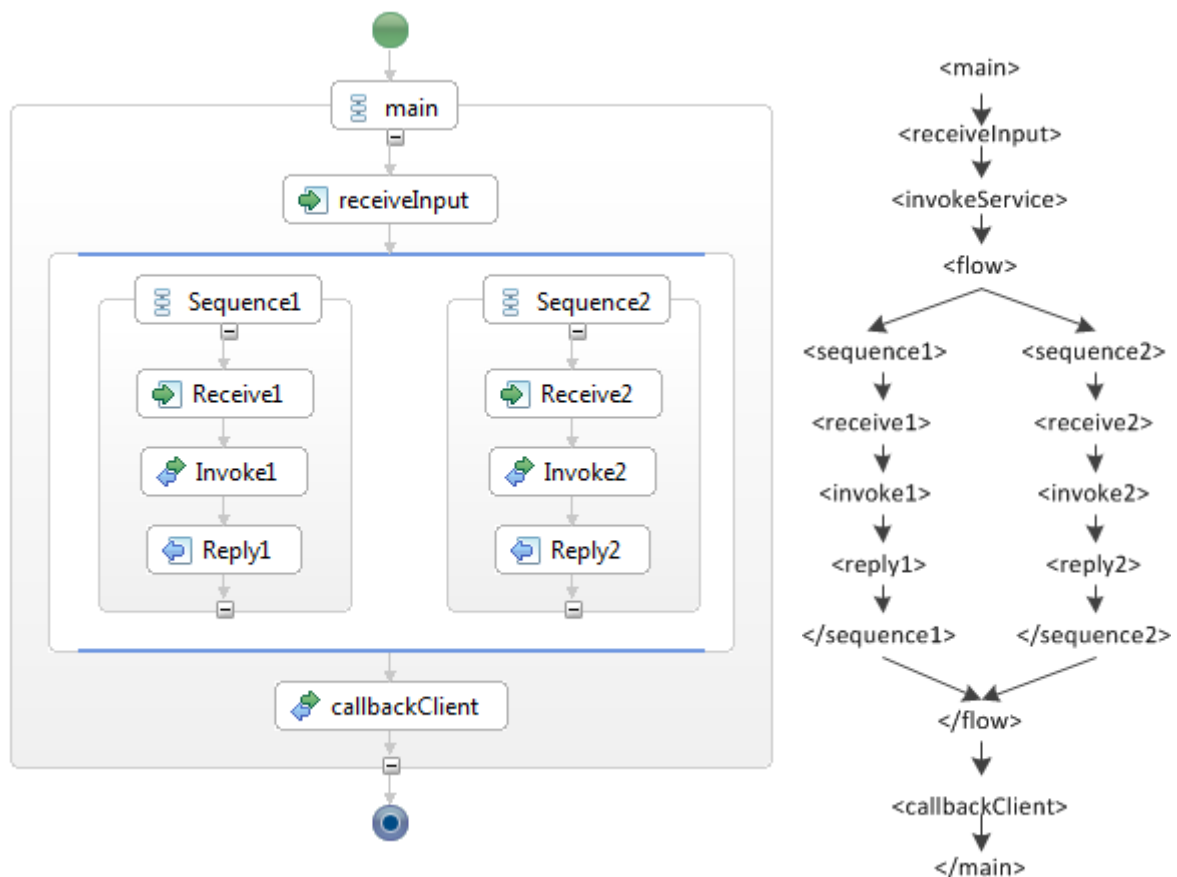


Rysunek 6.2 Schemat importu przy użyciu Eclipse Modelling Framework (EMF) na podstawie [5]

Komunikacja z obiektem BPEL DAO przedstawiona na rysunku 6.1 odbywa się dwukierunkowo w przypadku elementu *Analyzer* – wysłanie żądania wczytania procesu oraz w odpowiedzi odebranie wczytanego procesu w postaci obiektu EMF. W przypadku elementu *Updater* komunikacja z obiektem BPEL DAO zachodzi jednokierunkowo – wysłanie żądania do zapisu procesu w postaci EMF do pliku. Komunikacja z obiektem WSDL DAO prezentowanym na rysunku 6.1 zachodzi jednokierunkowo z elementem *Analyzer* – wysłanie żądania do wczytania listy plików zawierających opis usług sieciowych. Dwukierunkowa komunikacja z WSDL DAO zachodzi natomiast z elementem *Matcher* - jego zadaniem jest wyszukiwanie dopasowań pomiędzy zmiennymi procesu zarówno typów prostych jak i złożonych – polegając na wysłaniu żądania przesłania wczytanego do postaci EMF pliku WSDL oraz w odpowiedzi przekazanie obiektu.

### 6.1. Transformacja procesu do postaci grafu.

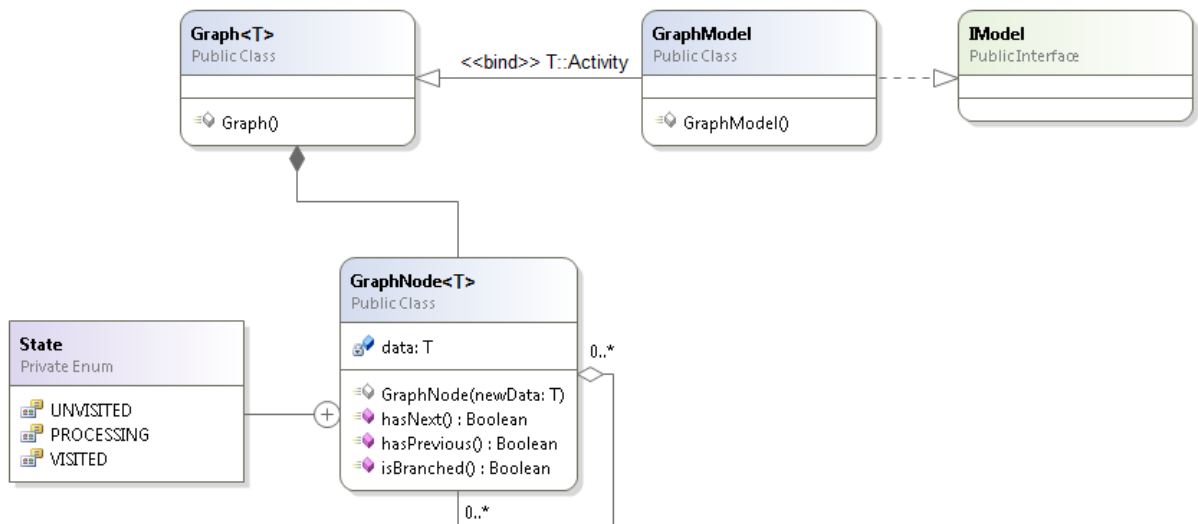
W ramach projektu wtyczki został zaprojektowany oraz zaimplementowany Transformator tworzący z procesu BPEL (model BPEL Designer) graf aktywności głównego przebiegu procesu. Proces BPEL w notacji Eclipse BPEL Designer oraz jego odpowiednik w postaci grafu przedstawiono na rysunku 6.3.



Rysunek 6.3 Proces BPEL z odpowiednikiem w postaci grafu na podstawie [6].

Model grafu reprezentuje strukturę procesu macierzystego, którego aktywności proste są reprezentowane jako pojedyncze węzły grafu. Aktywności strukturalne zostały zamodelowane w grafie jako pary węzłów – węzeł otwierający aktywność złożoną oraz węzeł zamykający. Model grafu jest tworzony na zasadzie iteracji po wszystkich elementach aktywności procesu BPEL i w zależności od złożoności elementu produkowany jest pojedynczy węzeł grafu lub ich para. Transformator trafiając na aktywność złożoną generuje węzły – otwierający oraz zamykający – po czym rekurencyjnie wywołuje metodę transformującą dla wszystkich aktywności znajdujących wewnątrz aktywności strukturalnej. W ten sposób powstaje model procesu w postaci grafu.

Zaprojektowany model grafu jest klasą implementującą interfejs modelu oraz dziedziczącą po klasie implementującej graf jak na diagramie klas przedstawionym na rysunku 6.4.



Rysunek 6.4 Diagram klas modelu grafu.

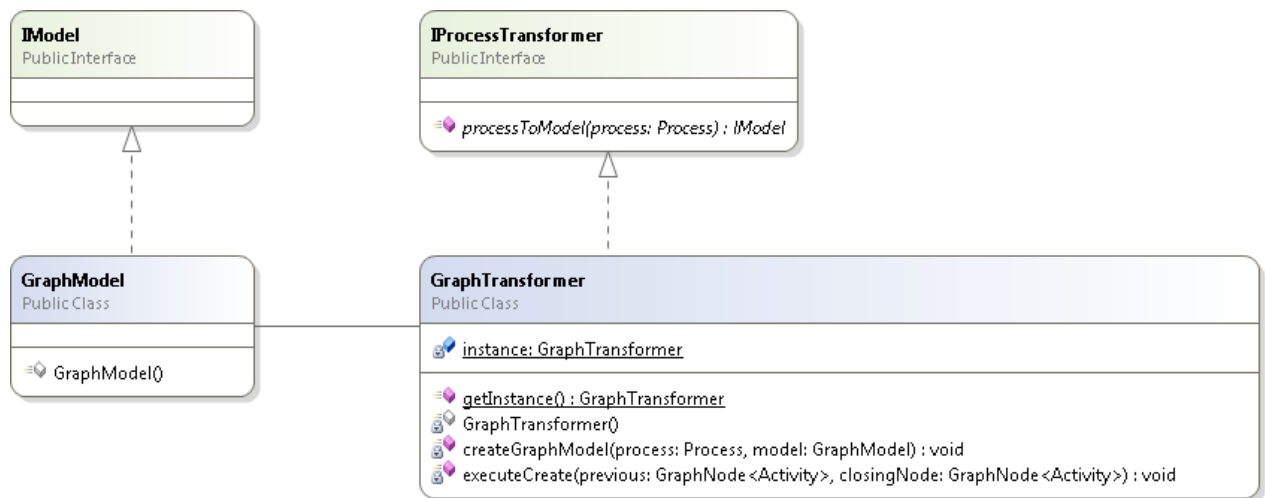
Graf zaimplementowany na potrzeby projektu ze względu na specyfikę problemu – nie mają znaczenia odległości między wierzchołkami, a jedynie istnienie połączeń między nimi – posiada jedynie informację o połączeniach z innymi wierzchołkami. Graf reprezentowany jest przez klasę generyczną, czyli także reużywalną na potrzeby innych modeli bazujących na strukturze takiego grafu. Graf reprezentowany jest przez klasę zawierającą jedynie referencję węzła głównego grafu (*ang. root node*). Pojedynczy węzeł grafu zawiera obiekt modelowanego typu oraz stan węzła będący wartością enumeracyjną. Pojedynczy wierzchołek grafu może znajdować się w stanie nieodwiedzony (*UNVISITED*), procesowany (*PROCESSING*) oraz odwiedzony (*VISITED*). Klasa węzła grafu zawiera także dwie listy z referencjami do obiektów omawianej klasy. Pierwsza zawiera listę węzłów grafu bezpośrednio poprzedzających, natomiast druga, listę węzłów grafu będących bezpośrednimi następnikami dla danego wierzchołka.

Model właściwy grafu wykorzystywanego w dalszej części analizy rozszerza klasę generyczną *Graph<T>* z parametrem typu *org.eclipse.bpel.model.Activity*. W efekcie model złożony jest z węzłów zawierających jako element *data* aktywności składające się na transformowany proces BPEL.

Zdefiniowany model jest w niniejszym projekcie wynikiem transformacji modelu procesu BPEL dostarczonego przez wtyczkę BPEL Designer.

Transformacja odbywa się przy użyciu obiektu klasy *GraphTransformer* implementującej zachowanie interfejsu *IProcessTransformer*, który narzuca na klasy implementujące go konieczność stworzenia definicji metody *processToModel*. Parametrem wywołania wymienionej metody jest instancja obiektu klasy implementującej interfejs *org.eclipse.bpel.model.Process*, czyli przechowywany w pamięci proces BPEL. Elementem zwracany przez metodę *processToModel* jest obiekt klasy implementującej interfejs *IModel*. W ten sposób zaprojektowany transformator procesu do wykorzystywanego w niniejszym

projekcie modelu w postaci grafu korzysta z wzorca projektowego *Fabryka*. Na rysunku Rysunek 6.5 przedstawiono diagram klas wchodzących w skład funkcji transformowania.



Rysunek 6.5 Transformator – diagram klas.

Na przedstawionym diagrami klas można zauważyć prywatny konstruktor klasy transformatora. W projekcie zablokowano możliwość tworzenia kilku instancji obiektu tej klasy. Podjęta została taka decyzja ponieważ obiekt klasy *GraphTransformer* jest obiektem bezstanowym i oferuje jedynie operacje na obiekcie dostarczonym mu z zewnątrz, dlatego wykorzystano wzorzec *Singletona* – do instancji obiektu możemy się odwołać jedynie poprzez statyczną metodę *getInstance()*, która zapewnia nam istnienie jednej instancji obiektu w obrębie maszyny wirtualnej *Java*.

Korzystając z wiedzy dotyczącej procesów BPEL wiadomo, że przebieg procesu jest zamknięty w sekwencji głównej – główny przebieg procesu – będącej instancją klasy implementującej interfejs *org.eclipse.bpel.model.Activity* – tak jak wszystkie elementy procesu składające się na przebieg. Rozpoczynając transformację konieczne jest znalezienie sekwencji głównej procesu rozpoczynającej ten przebieg, który jednocześnie zostanie użyty jako element *data* węzła *root* (węzła startowego) powstającego grafu procesu BPEL. Znalezienie aktywności głównej polega na iteracji z wykorzystaniem iteratora po wszystkich elementach składających się na proces. Na listingu 6.1 przedstawiono zapisaną w pseudokodzie funkcję inicjującą transformację procesu BPEL do postaci grafu. W wyróżnionej kolorem żółtym linijce zawierającej *process.eAllContents()* inicjalizowany jest iterator (*TreeIterator*) elementów procesu BPEL w postaci modelu EMF procesu dostarczonego przez Eclipse BPEL Designer. Następnie z wykorzystaniem iteratora kolejno sprawdzane są elementy procesu w poszukiwaniu pierwszego wystąpienia elementu *Activity*, czyli aktywności strukturalnej *Sequence* będącej sekwencją główną procesu BPEL.

```

procedure createGraphModel(process, model)
    processIterator = process.eAllContents()
    temp = null
    while processIterator.hasNext do
        begin
            temp = processIterator.next
            if temp instanceof Activity then
                begin
                    break
                end
            end
        end
    node rootActivity = new node(temp)
    node complexNodeClone = null
    model.root := rootActivity
    if temp not basic activity then
        begin
            complexNodeClone = new node(temp)
        end
    executeCreate(model.root, complexNodeClone)

```

Listing 6.1 Funkcją inicjująca transformację – pseudokod.

Po znalezieniu pierwszej aktywności – iteracja zostaje przerwana. Stworzony zostaje elementem *root* generowanego grafu, ponieważ sekwencja jest aktywnością złożoną zostaje utworzony dodatkowy węzeł grafu, który stanowi wierzchołek dodany jako ostatni do struktury.

Utworzony węzeł główny oraz węzeł „zamykający” stają się parametrami wywołania funkcji transformującej działającej wg założeń algorytmu transformacji. Funkcja transformująca została na listingu 6.1 wyróżniona kolorem żółtym

Rozpoczyna się iteracja po elementach stanowiących bezpośrednie potomstwo aktywności z węzła *previous*. Pierwszym krokiem jest sprawdzenie typu obiektu – procesowane są jedynie aktywności, czyli instancje klas implementujących interfejs *org.eclipse.bpel.Activity*. Następnie każda iterowana aktywność jest badana pod kątem złożoności, elementy nie będące aktywnościami strukturalnymi procesu natychmiast trafiają do grafu jako *data* nowo utworzonego węzła i jeśli parametr wywołania *previous* nie zawierał aktywności przepływu (*Flow*) następuje przepisanie referencji – *previous* zaczyna wskazywać na nowo dodany węzeł grafu. W sytuacji gdy iterowany element jest aktywnością strukturalną następuje wywołanie rekurencyjne (oznaczone kolorem żółtym na listingu 6.2) dla węzłów otwierającego – uprzednio dodanego do grafu – i zamykającego tę aktywność. Po zakończeniu wywołania rekurencyjnego przepisywana jest refencja *previous* dla tego samego warunku, co w przypadku aktywności prostych. Ostatnim krokiem pętli – w przypadku gdy parametr wywołania metody zawierał aktywność przepływu – dodanie węzła zamykającego (parametr wywołania) jako następnika dodanego nowego węzła. Dla aktywności strukturalnych innych niż przepływ (*Flow*) dokładnie takie samo przepisanie

następuje po zakończeniu ostatniej iteracji. Na listingu 6.2 przedstawiono algorytm transformacji (funkcja *executeCreate*) procesu BPEL do modelu grafu w postaci pseudokodu.

```

procedure executeCreate(startNode, closingNode)
begin
  contents := startNode.activityChildren
  insertedNode := null
  complexEndNode := null
  bool isFlow := startNode.activity instanceof Flow
  for processing : contents do
    begin
      if processing instanceof Activity then
        begin
          insertedNode := new node(processed)
          if processed is basic activity then
            begin
              insertedNode.previousList.add(previous)
              previous.nextList.add(insertedNode)
              if isFlow then
                begin
                  insertedNode.nextList.add(closingNode)
                  closingNode.previousList.add(insertedNode)
                end
              else
                begin
                  previous := insertedNode
                end
              end
            end
          else
            begin
              complexEndNode := new node(processed)
              insertedNode.previousList.add(previous)
              previous.nextList.add(insertedNode)

              executeCreate(insertedNode, complexEndNode)

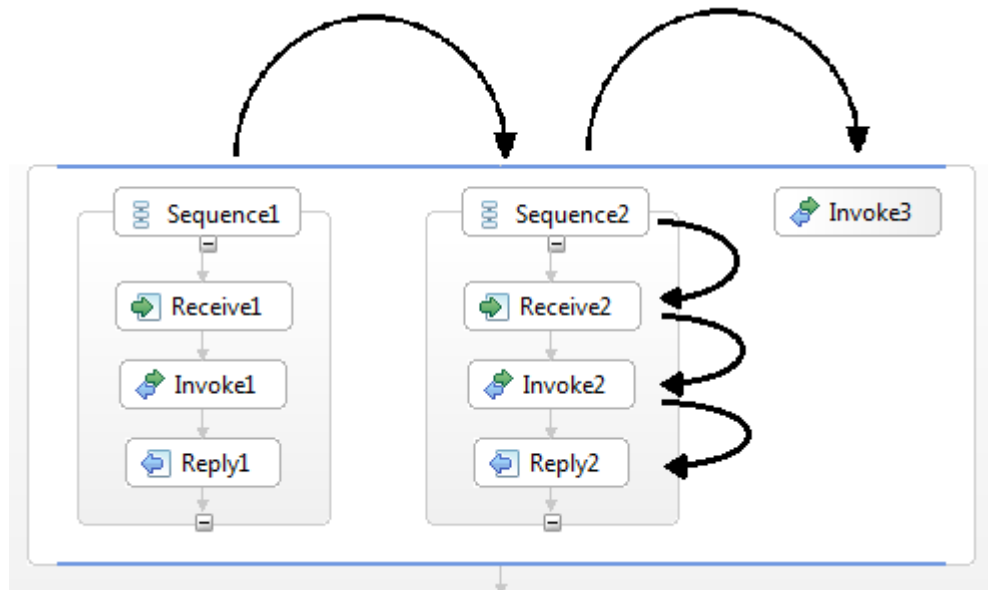
              if isFlow then
                begin
                  complexEndNode.addNextNode(closingNode)
                  closingNode.addPreviousNode(complexEndNode)
                end
              else
                begin
                  previous := complexEndNode
                end
              end
            end
          end
        end
      if not isFlow then
        begin
          previous.nextList.add(closingNode);
          closingNode.previousList.add(previous);
        end
      end
    end
  end

```

Listing 6.2 Algorytm transformacji procesu BPEL do postaci grafu – pseudokod.



Takie zachowanie algorytmu wynika z różnicy sposobu iteracji po bezpośrednich potomkach zachodzącej pomiędzy przepływem (*Flow*), a innymi aktywnościami strukturalnymi. Na rysunku 6.6 przedstawiono różnice pomiędzy iteracją *Flow* – każdy iterowany element powinien wskazywać na węzeł zamykający przepływ jako na element następny - a iteracją po innych strukturalnych aktywnościach procesu, w tym przypadku sekwencji – tylko ostatni iterowany element powinien wskazywać na węzeł zamykający jako na element następny.

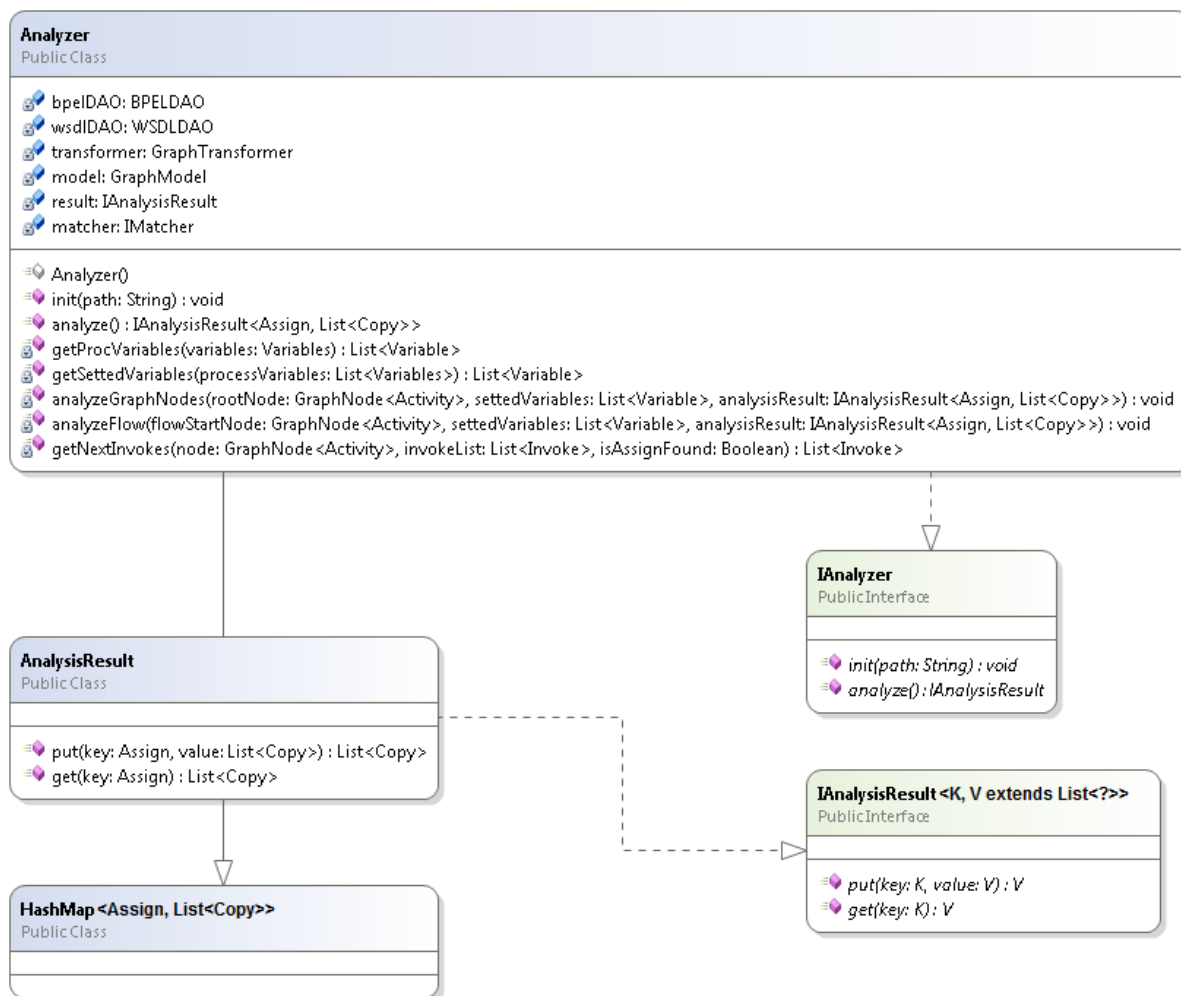


Rysunek 6.6 Różnica w sposobie iteracji po potomkach aktywności strukturalnych.

Model grafu utworzony w wyniku opisanej transformacji jest gotowy do użycia przez analizator, w celu odnalezienia możliwych dopasowań wśród zmiennych procesu.

## 6.2. Analiza grafu procesu.

Kluczowym etapem działania zaprojektowanej wtyczki jest analiza modelu grafu procesu w celu znalezienia dopasowań pomiędzy istniejącymi zmiennymi procesu. Wynikiem działania analizatora jest lista aktywności przepisania wartości (*Assign*) zmapowanych na listy instrukcji kopiujących wartości pomiędzy odnalezionymi dopasowaniami. Na rysunku Rysunek 6.7 zaprezentowano diagram klas, na którym przedstawiona została para interfejsów *IAnalysisResult* oraz *IAnalyzer* zaimplementowanych odpowiednio przez klasy *AnalysisResult* oraz *Analyzer*. Zwracany wynik analizy jest mapą zawierającą listę instrukcji *Assign* jako kluczy, które wskazują listy instrukcji *Copy* utworzonych w procesie analizy.



Rysunek 6.7 *Analyzer* – diagram klas.

Inicjowanie procesu analizy grafu rozpoczyna się od utworzenia dwóch list. Pierwsza z nich zawiera wszystkie zmienne (*Variable*) występujące w analizowanym procesie – zarówno zmienne typu złożonego jak i prostego – poprzez wyszukanie w wejściowym procesie wszystkich obiektów stanowiących instancję typu *org.eclipse.bpel.Variable*. Druga z inicjowanych list zawiera zestaw zmiennych procesu z ustawioną wartością oraz dodatkowo zmienna stanowiąca parametr wywołania procesu – wartości dostarczane do procesu przez konsumenta.

Rozpoczyna się iteracja po elementach grafu startując z węzła będącego węzłem głównym. Iterowane są tylko elementy nieodwiedzone. W każdej iteracji sprawdzane są trzy podstawowe warunki weryfikujące rodzaj aktywności zawartej w iterowanym węźle grafu – w przypadkach gdy wierzchołek zawiera aktywność *Flow*, *Assign* oraz *Invoke* wykonywane są dodatkowe funkcje wchodzące w skład algorytmu analizującego. Natrafienie na węzeł aktywności przepływu algorytm – jako, że jest to aktywność strukturalna – rozpoznawany jest jego rodzaj:

- otwierający,
- zamykający.

```

procedure analyzeGraphNodes(node rootNode, list settedVariables,
                             map analysisResult)
begin
    node current := rootNode
    while current.hasNext and not current.state == VISITED do
        begin
            processingActivity := current.data
            bool isFlow := processingActivity instanceof Flow
            if isFlow then
                begin
                    if rootNode.hasPrevious and
                        processingActivity == rootNode.previous.data then
                        begin
                            break
                        end
                    else
                        begin
                            for nodeElem : processingActivity.nextList do
                                begin
                                    analyzeGraphNodes(nodeElem, settedVariables, analysisResult)
                                end
                                current.state := VISITED
                                current := closingFlow
                            end
                        end
                    else if processingActivity instanceof Assign then
                        begin
                            list followingInvokes := new list
                            getNextInvokes(current.next, followingInvokes, FALSE)
                            list copyBlocks := matcher(settedVariables, followingInvokes)
                            if not (null == copyBlocks or copyBlocks.isEmpty) then
                                begin
                                    analysisResult.put(processingActivity -> copyBlocks)
                                end
                            else
                                begin
                                    if processingActivity instanceof Invoke then
                                        begin
                                            invokeActivity := processingActivity
                                            if not (null == invokeActivity) then
                                                begin
                                                    settedVariables.add(invokeActivity.outputVariable)
                                                end
                                            end
                                        end
                                    end
                                end
                                current.state := VISITED
                                current := current.next
                                if current == UNVISITED then
                                    begin
                                        current := PROCESSING
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```

Listing 6.3 Algorytm analizy grafu procesu – pseudokod.

Na listingu 6.3 przedstawiono algorytm wykorzystany do analizy grafu procesu zapisany w pseudokodzie.

W przypadku węzła otwierającego instrukcję *Flow* omawiany algorytm zostaje rekurencyjnie wywołany dla każdego z równoległych przebiegów (oznaczone kolorem żółtym na listingu 6.3 wywołanie funkcji *analizeGraphNodes*) składających się na instrukcję *Flow* traktując je jak podprocesy – podgrafy – iteruje po aktywnościach równoległych przebiegów. Dla węzłów zamykających iteracja jest przerywana – następuje zakończenie rekurencyjnego wywołania algorytmu. W sytuacji, w której iterowany węzeł grafu reprezentuje aktywność *Invoke* – sygnalizująca wywołanie zewnętrznej usługi w procesie – do listy zmiennych posiadających wartość dodana zostaje odpowiedź pochodząca z wywołanej usługi (ang. *Response*).

Kolejnym specyficznym przypadkiem jest węzeł reprezentujący aktywność przepisania wartości zmiennej/zmiennych (*Assign*). Trafiając na taki węzeł algorytm wyszukuje wszystkie instrukcje typu *Invoke*, których wywołanie w procesie następuje po wystąpieniu instrukcji *Assign* z aktualnie iterowanego wierzchołka grafu procesu – wyszukiwanie wierzchołka *Invoke* rozpoczyna się z aktualnie procesowanego węzła grafu. Ważnym elementem wyszukiwania jest fakt przerywania w momencie natrafienia na inną instrukcję *Assign* – by uniknąć sytuacji, w której jedna instrukcja przepisywania wartości posiada instrukcję kopiującą wartości zmiennych do wywołań wszystkich usług zewnętrznych następujących po niej, aż do zakończenia procesu. W tej sytuacji mógłby powstać problem dotyczący wydajności procesu gdyż instrukcje kopiujące te same wartości byłyby powielone w innych instrukcjach *Assign* procesu. Wyszukiwanie instrukcji wywołania usług zewnętrznych jest także specyficzne dla aktywności przepływu, w tym wypadku każdy z przepływów podlega równoległemu wyszukiwaniu. Tak powstaje lista instrukcji *Invoke* zawierająca rezultat wyszukiwania wywołań usług zewnętrznych.

W celu zbadania możliwości dopasowań - i na ich podstawie stworzenia instrukcji kopiujących - pomiędzy listą zmiennych zainicjowanych w procesie oraz listą zmiennych będących parametrami wywołania usług zewnętrznych został użyty dodatkowy algorytm w projekcie oddzielony od implementacji analizatora jako zewnętrzny mechanizm. *Matcher*, służący do badania dopasowań pomiędzy zestawami zmiennych został oddzielony od analizatora. Algorytm porównujący bazując na dostarczonych listach zmiennych zainicjowanych oraz zmiennych użytych jako parametry wywołań usług zewnętrznych porównuje po dwa elementy pochodzące z obu list rozpatrując sytuacje przedstawione w tabeli 7.1. Opis algorytmu wyszukiwania dopasowań – przypadek dla par zmiennych typu złożonego (*Złożona-Złożona* z tabeli 7.1) - w postaci pseudokodu został przedstawiony na listingu 6.4. Kolorem żółtym na listingu oznaczono porównanie nazw oraz typów zmiennych typów prostych stanowiących elementy badanego typu złożonego.

```

procedure createCopyForMatchedVariables(list settedVariables,
                                         list followingInvokes)
begin
    list result := new list
    for it : followingInvokes do
        begin
            invokeInput := it.inputVariable
            for var : settedVariables do
                begin
                    if null != var.messageType or
                        null != invokeInput.messageType then
                        begin
                            list complexResult := new list
                            varComplexType := wsdlLoader.getMessage(varToCopyFrom.messageType)
                            invokeComplexType = wsdlLoader.getMessage(varToCopyTo.messageType)
                            for varElem : varComplexType.elements do
                                begin
                                    for invokeElem : invokeComplexType.elements do
                                        begin
                                            if varElem.name == invokeElem.name and
                                                varElem.type == invokeElem.type then
                                                begin
                                                    copyElem := new copy
                                                    copyElem.from := new from(varElem)
                                                    copyElem.to := new to(invokeElem)
                                                    complexResult.add(copyElem)
                                                end
                                            end
                                        end
                                    end
                                end
                                result.addAll(complexResult);
                            end
                        else if null != var.messageType and
                            null == invokeInput.messageType then
                            begin
                                ... // simple-complex type check
                            end
                        else if null == var.messageType and
                            null != invokeInput.messageType then
                            ... // complex-simple type check
                        end
                        else
                            begin
                                ... // simple-simple type check
                            end
                        end
                    end
                end
            end
        return result
    end

```

Listing 6.4 Wyszukiwanie dopasowań dla pary typów *Złożony-Złożony* – pseudokod.

W każdym przypadku, gdy choć jedna ze zmiennych badanych jest typu złożonego *Matcher* odpytuje obiekt *WSDL DAO* w celu uzyskania definicji typu złożonego badanej zmiennej.

Tabela 7.1 Rodzaje typów par zmiennych porównywanych.

Rodzaj typu.	
Zmienna zainicjalizowana.	Zmienna – parametr.
Złożony.	Złożony.
Złożony.	Prosty.
Prosty.	Złożony.
Prosty.	Prosty.

W przypadku, gdy typy obu badanych zmiennych są złożone i jednocześnie to te same typy następuje bezpośrednie utworzenie instrukcji kopiującej wartość zmiennej zainicjalizowanej do zmiennej wywołania. W sytuacji gdy typy różnią się między sobą do warstwy dostępu do danych zostaje wysłana prośba o podanie definicji obu typów. Otrzymane w odpowiedzi definicje każdego z typów zostają przetłumaczone przez obiekt *Resolver* na listę ciągów znakowych w formacie:

*MessagePartName.XSDElementTypeName.XSDElementName.*

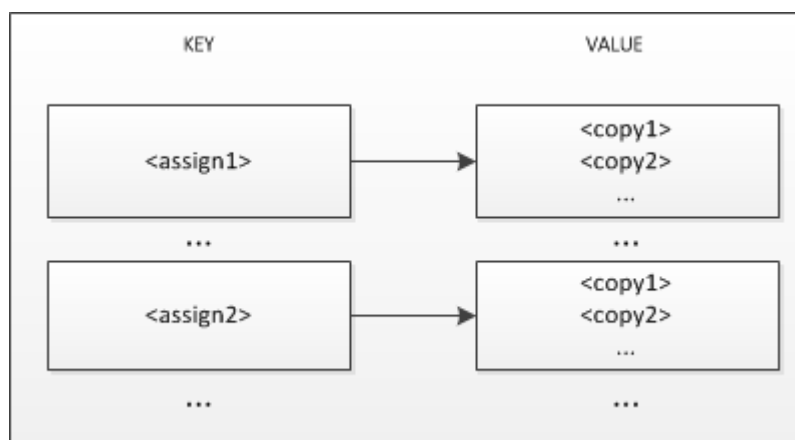
Obie listy „rozwiązanych” typów zostają porównywane pod względem zgodności nazw i typów elementów składowych, które tworzą typ złożony zmiennej biorącej udział w wyszukiwaniu dopasowań. Rezultatem procesu porównywania jest lista par – element typu złożonego z którego nastąpi przepisanie oraz element typu złożonego do którego nastąpi przepisanie wartości - ciągów znakowych w przedstawionym formacie. Tak utworzone dopasowania podlegają już tylko procesowi generacji instrukcji kopiujących, które są dodawane do listy wyjściowej mechanizmu wyszukiwania dopasowań.

Pary badanych zmiennych mogą również być zmiennymi o różnej złożoności typów (patrz tabela 7.1), czyli pary *złożony-prosty* oraz *prosty-złożony*. W takim przypadku algorytm działa przygotowuje zmienną typu złożonego na tej samej zasadzie co w sytuacji porównywania dwóch zmiennych typu złożonego, czyli pobiera definicję typu złożonego z warstwy dostępu do danych (*WSDL*) i tłumaczy elementy typu na ciągi w zaprezentowanym formacie. Następnie dokonuje porównań elementów typu złożonego ze zmienną typu prostego, jeżeli szereg porównań znalazł jakiegokolwiek dopasowania – typu i nazwy – następuje utworzenie instrukcji kopiującej i dodanie do listy instrukcji *Copy* będącej rezultatem działania mechanizmu poszukiwania dopasowań.

Mając do czynienia z dwoma typami prostymi algorytm dokonuje prostego porównania typu oraz nazwy i na podstawie jego wyniku generuje lub nie instrukcje

kopiującą. Wygenerowaną instrukcję *Copy* podobnie dodaje do wyniku działania mechanizmu.

Utworzona w ten sposób lista instrukcji kopiujących trafia do obiektu *AnalysisResult* mającego strukturę mapy przedstawionej na rysunku 6.8.

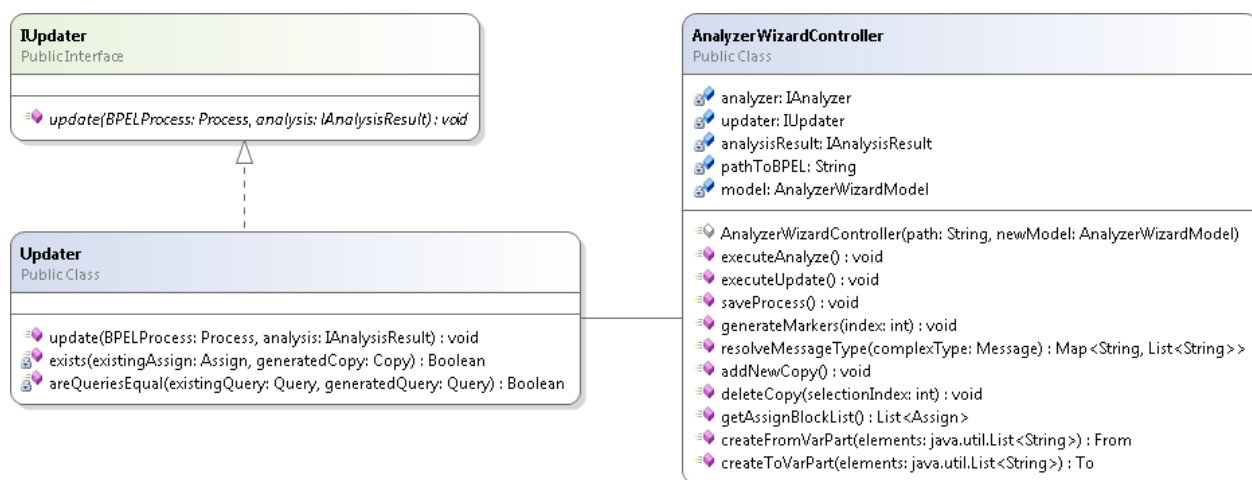


Rysunek 6.8 Schemat struktury rezultatu analizy dopasowań.

Każdemu przeprocesowanemu węzłowi grafu procesu ustawiany jest na „odwiedzony” (*VISITED*), do dalszej analizy pobrany zostaje następny element z grafu, któremu zostaje ustawiony stan „procesowany” (*PROCESSING*). Po zakończeniu działania algorytmu produkt analizy jest gotowy do użycia go w celu zaktualizowania procesu BPEL poprzez dodanie nieistniejących do tej pory instrukcji kopiujących odpowiednim blokom przepisywania wartości zmiennych (*Assign*).

### 6.3. Aktualizacja instrukcji kopiujących

Wynik analizy procesu zawierający zbiór instrukcji kopiujących, które będą dodane do odpowiednich elementów *Assign* przetwarzanego procesu BPEL. Na rysunku 6.9 przedstawiono diagram klas opisujący mechanizm aktualizujący proces.



Rysunek 6.9 *Updater* procesu BPEL – diagram klas.

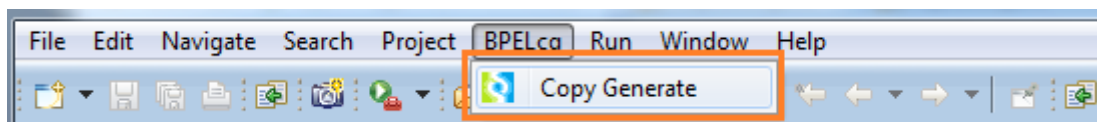
Aktualizacja procesu BPEL polega na dodaniu instrukcji kopiujących do wybranych bloków *Assign*. Iterując po elementach procesu sprawdzany jest ich typ, dla obiektów będących instancjami klasy implementującej interfejs *org.eclipse.bpel.mode.Assign* w pętli dodawane są kolejno elementy pochodzące z listy instrukcji kopiujących, na które wskazuje referencja procesowanego obiektu *Assign* – korzystając z mapy będącej wynikiem uprzedniej analizy procesu. Dodane zostają te instrukcje kopiujące, których aktualizowany blok nie zawiera.

Zaktualizowany proces jest gotowy do zaprezentowania użytkownikowi w postaci panelu konfiguracyjnego instrukcji *Assign*.

## 6.4. Graficzny interfejs użytkownika.

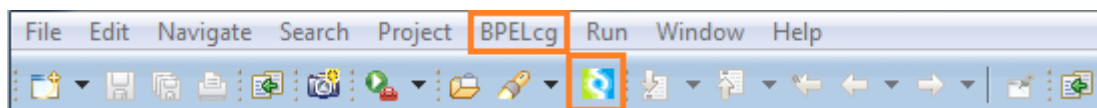
Wtyczka generująca automatycznie instrukcje kopiujące dostarcza interfejs użytkownika na który składa się:

- Menu (*BPELcg*) znajdujące się w Menu Bar – głównym menu środowiska Eclipse – umożliwiające uruchomienie generatora instrukcji kopiujących – rysunek 6.10.



Rysunek 6.10 Menu *BPELcg* – menu bar.

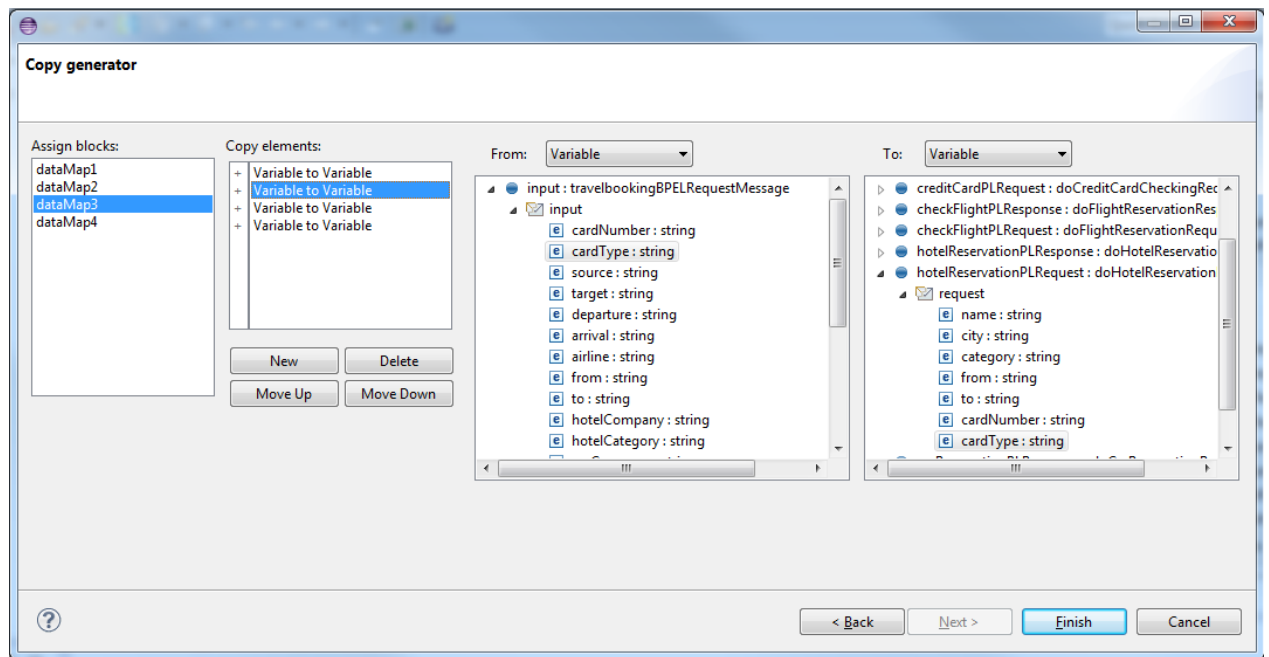
- Przycisk Toolbar – podobnie jak menu *BPELcg* – aktywujący akcję uruchomienia generatora instrukcji kopiujących w procesie BPEL – rysunek 6.11.



Rysunek 6.11 Przycisk *BPELcg* – toolbar.

- Wizard umożliwiający przegląd wszystkich instrukcji kopiujących – zarówno tych wygenerowanych, jak również dodanych przed uruchomieniem generatora - dla każdej instrukcji *Assign* w procesie oraz manualną edycję konfiguracji bloków przepisania wartości zmiennych w procesie – rysunek 6.11.





Rysunek 6.11 Wizard umożliwiający przegląd oraz edycję instrukcji kopiujących.

- Okno komunikatu wyświetlane przy próbie uruchomienia generatora, gdy aktywny plik w edytorze Eclipse nie jest procesem BPEL – Rysunek 6.12.

Rysunek 6.12 Okno komunikatu – plik aktywny w edytorze nie jest procesem BPEL.

Okno wizarda stworzone zostało w celu umożliwienia wygodnego przeglądu rezultatów działania generatora dla wszystkich instrukcji przepisywania wartości zmiennych procesu. List *Assign blocks* zawiera listę instrukcji *Assign* projektowanego procesu BPEL, po zaznaczeniu jednej z wylistowanych instrukcji na liście *Copy elements* pojawiają się wszystkie instrukcje kopiujące danego bloku przepisania wartości. Każda instrukcja kopiująca posiada również znacznik informujący o tym, czy została wygenerowana – obecność znaku + w lewej kolumnie list obok elementu *Copy*. Wybranie instrukcji kopiującej odświeża pola typu *Combobox* uzupełniając je wartościami definiującymi typ elementu z i do którego wartość jest kopiowana. Na rysunku 6.11 przedstawiono instrukcję kopiującą z elementu zmiennej typu złożonego do także elementu typu złożonego. Prezentowany wizard daje również możliwość manualnej edycji instrukcji *Assign* poprzez dodawanie nowych instrukcji kopiujących, usuwanie istniejących oraz zmianę kolejności instrukcji kopiujących znajdujących się na liście. Znajdujące się pod *Combobox*'ami listy zmiennych prezentowane w strukturze drzewiastej – typy złożone, typy proste nie mogą być rozwinięte służą do wyboru konkretnych elementów pomiędzy którymi ma dojść do skopiowania wartości. Poza możliwością przeglądu wyników generacji instrukcji kopiujących, wizard stanowi także ciekawą i wygodną alternatywę dla dotychczas używanych widoków konfiguracyjnych instrukcji *Assign* ze względu na listę wszystkich bloków przepisania wartości zmiennych występujących w procesie – szybki dostęp nie

zmuszający użytkownika do szukania – w dużych procesach często mozolnego – bloku *Assign*, którego konfigurację użytkownik życzy sobie zmodyfikować.

## 6.5. Konfiguracja wtyczki.

Zaprojektowana wtyczka wprowadza rozszerzenie do *Menu* środowiska Eclipse dodając elementy w *Menu Bar* oraz *Tool Bar* niezbędne do uruchomienia automatycznej analizy oraz generacji instrukcji kopiujących edytowanego procesu BPEL. Punktem rozszerzenia (ang. *Extension Point*) środowiska Eclipse użytym w projekcie jest *org.eclipse.ui.actionSets* – Listing 6.3.

```
<plugin>
  <extension point="org.eclipse.ui.actionSets">
    ...
  </extension>
</plugin>
```

Listing 6.3 Punkt rozszerzenia środowiska Eclipse w projektowanej wtyczce.

Konfiguracja wtyczki została także wzbogacona o warunkowe ustawienie widoczności rozszerzonego menu – dodane opcje widoczne w menu tylko dla perspektywy BPEL – Listing 6.4.

```
...
<visibleWhen checkEnabled="true">
  <with variable="activeWorkbenchWindow.activePerspectives">
    <equals value="BPEL" />
  </with>
</visibleWhen>
...
```

## 7. Testy funkcjonalne.

Na potrzeby testów funkcjonalnych przygotowany został proces *travelbooking* stworzony na podstawie [3] - przedstawiony w rozdziale 5 części poświęconej *Przykładowemu procesowi BPEL*. W tabeli 7.1 przedstawiono listę zmiennych procesu wraz z rozwinięciem zmiennych o typach złożonych.

Tabela 7.1 Lista zmiennych procesu *travelbooking* wraz z rozwinięciem typów złożonych.

<i>Variable : Message</i>	<i>Message parts</i>	<i>Message part elements</i>
<i>input</i> <i>:travelbookingBPELRequestMessage</i>	<i>input</i>	<i>cardNumber:string</i> <i>cardType:string</i> <i>source:string</i> <i>target:string</i> <i>departure:string</i> <i>arrival:string</i> <i>airline:string</i> <i>from:string</i> <i>to:string</i> <i>hotelCompany:string</i> <i>hotelCategory:string</i> <i>carCompany:string</i> <i>carCategory:string</i>
<i>output</i> <i>:travelbookingBPELResponseMessage</i>	<i>output</i>	<i>information:string</i>
<i>creditCardPLResponse</i> <i>:doCreditCardCheckingResponse</i>	<i>response</i>	<i>doCreditCardCheckingReturn:boolean</i>
<i>creditCardPLRequest</i> <i>:doCreditCardCheckingRequest</i>	<i>request</i>	<i>cardNumber:string</i> <i>cardType:string</i>
<i>checkFlightPLResponse</i> <i>:doFlightReservationResponse</i>	<i>response</i>	<i>doFlightReservationReturn:boolean</i>
<i>checkFlightPLRequest</i> <i>:doFlightReservationRequest</i>	<i>request</i>	<i>airline:string</i> <i>source:string</i> <i>target:string</i> <i>departure:string</i> <i>arrival:string</i> <i>cardNumber:string</i> <i>cardType:string</i>
<i>hotelReservationPLResponse</i> <i>:doHotelReservationResponse</i>	<i>response</i>	<i>doHotelReservationReturn:boolean</i>
<i>hotelReservationPLRequest</i> <i>:doHotelReservationRequest</i>	<i>request</i>	<i>name:string</i> <i>city:string</i> <i>category:string</i> <i>from:string</i> <i>to:string</i> <i>cardNumber:string</i> <i>cardType:string</i>
<i>carReservationPLResponse</i> <i>:doCarReservationResponse</i>	<i>response</i>	<i>doCarReservationReturn:boolean</i>
<i>carReservationPLRequest</i>	<i>request</i>	<i>company:string</i>

<i>:doCarReservationRequest</i>		<i>category:string from:string to:string cardNumber:string cardType:string</i>
<i>confirmOutput:confirmResponse</i>	<i>confirmResponse</i>	<i>travelConfirmation:boolean</i>
<i>confirmInput:confirmRequest</i>	<i>confirmRequest</i>	<i>carConfirmation:Boolean hotelConfirmation:Boolean cardConfirmation:Boolean flightConfirmation:Boolean</i>

Tabela 7.2 Instrukcje *Invoke* procesu wraz z parametrami we/wy.

Instrukcja <i>Invoke</i> : operacja	Parametry wejściowe/wyjściowe
<i>checkCreditCard:doCreditCardChecking</i>	<i>IN: creditCardPLRequest</i>
	<i>OUT: creditCardPLResponse</i>
<i>checkHotelReservation:doHotelReservation</i>	<i>IN: hotelReservationPLRequest</i>
	<i>OUT: hotelReservationPLResponse</i>
<i>doHotelReservation:hotelReservationPLResponse</i>	<i>IN: hotelReservationPLRequest</i>
	<i>OUT: hotelReservationPLResponse</i>
<i>checkCarReservation:doCarReservation</i>	<i>IN: carReservationPLRequest</i>
	<i>OUT: carReservationPLResponse</i>
<i>Confirmation: confirm</i>	<i>IN: confirmInput</i>
	<i>OUT: confirmOutput</i>

Rozpatrywany proces zawiera cztery bloki przepisywania danych (*Assign*). Wśród przetestowanych scenariuszy pojawiły się między innymi dwa przykładowe przypadki testowe (*ang. Test Case – TC*):

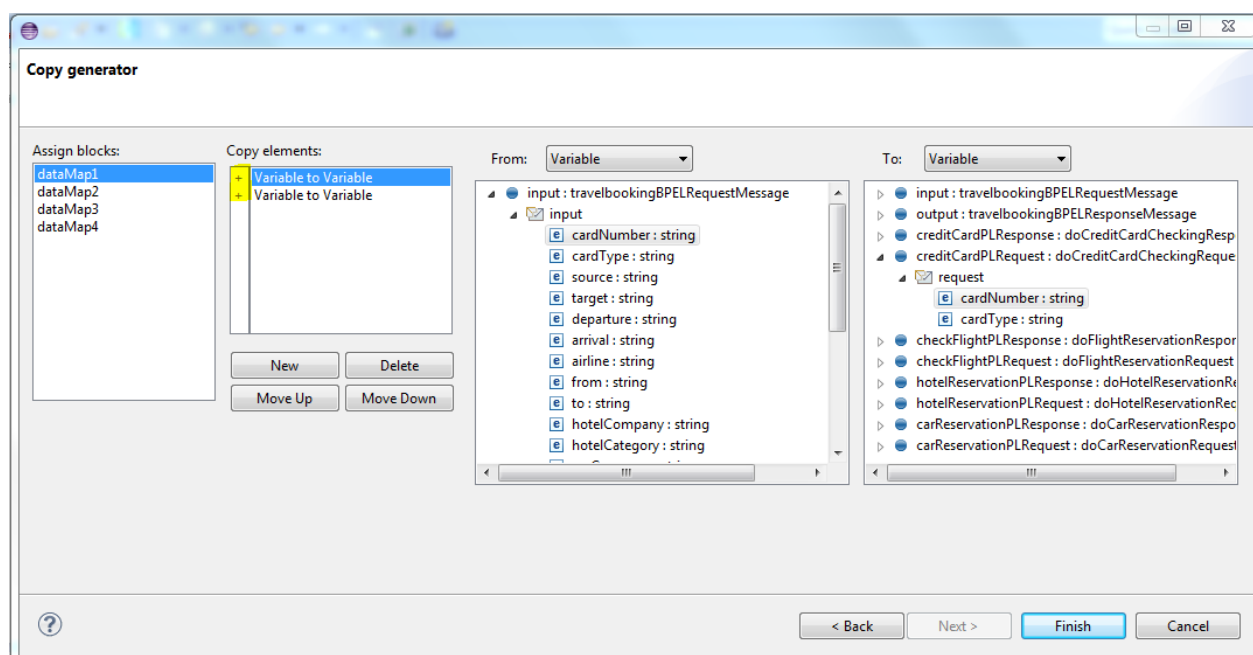
- [TC1] generowanie instrukcji kopiujących, instrukcje *Assign* w procesie nie zawierają żadnych instrukcji kopiujących.
- [TC2] dodanie nowej operacji kopiowanej przez wizarde testowanej wtyczki do generowania instrukcji kopiujących.

[TC1] W przypadku testów 1 instrukcje *Assign* procesu były puste – nie istniały żadne przepisania wartości w procesie. Na listingu 7.1 przedstawiono wszystkie bloki *Assign*.

```
<process>
...
<sequence name="travelBooking">
...
  <assign name="dataMap1" validate="yes">
  </assign>
...
  <flow>
    ...
    <bpel:assign name="dataMap2" validate="no">
    </bpel:assign>
    ...
    <bpel:assign name="dataMap3" validate="no">
    </bpel:assign>
    ...
    <bpel:assign name="dataMap4" validate="no">
    </bpel:assign>
    ...
  </flow>
</sequence>
</process>
```

Listing 7.1 wygląd instrukcji *Assign* przed uruchomieniem generatora instrukcji kopiujących.

Po uruchomieniu generatora instrukcji *Copy* został zaprezentowany *wizard* z możliwością przejrzania wyników działania. Widok *wizarda* dla instrukcji *Assign* *dataMap1* pokazano na rysunku 7.1. Kolorem żółtym zaznaczone są *markery* oznaczające instrukcje kopiujące dodane w wyniku uruchomionej generacji.



Rysunek 7.1 Wizard z rezultatem działania generatora dla przypadku [TC1] – instrukcja *Assign* – *dataMap1*.

W efekcie działania generatora do instrukcji *Assign* procesu dodane zostały wygenerowane instrukcje kopiujące. Wygląd bloków przepisywania wartości dla testowanego procesu został przedstawiony na listingu 7.2 – kolorem żółtym oznaczono instrukcje kopiujące wygenerowane przez omawiane narzędzie dla instrukcji *Assign* *dataMap1*.

```
<process>
...
<sequence name="travelBooking">
...
  <assign name="dataMap1" validate="yes">
    <copy>
      <from part="input" variable="input">
        <query queryLanguage="...xpath1.0">
          <![CDATA[tns:cardNumber]]>
        </bpel:query>
      </from>
      <to part="request" variable="creditCardPLRequest">
        <query queryLanguage="...xpath1.0">
          <![CDATA[ns0:cardNumber]]>
        </query>
      </to>
    </copy>
    <copy>
      <from part="input" variable="input">
        <query queryLanguage="...xpath1.0">
          <![CDATA[tns:cardType]]>
        </query>
      </from>
      <to part="request" variable="creditCardPLRequest">
        <query queryLanguage="...xpath1.0">
          <![CDATA[ns0:cardType]]>
        </query>
      </to>
    </copy>
  </assign>
...
  <flow>
    ...
    <assign name="dataMap2" validate="no">
    </assign>
    ...
    <assign name="dataMap3" validate="no">
    </assign>
    ...
    <assign name="dataMap4" validate="no">
    </assign>
    ...
  </sequence>
</process>
```

Listing 7.2 Wygląd instrukcji *Assign dataMap1* po zakończeniu działania generatora instrukcji kopiujących.

Tabela 7.3 Lista instrukcji kopiujących w procesie *travelbooking*.

Oczekiwane instrukcje kopiujące w procesie <i>travelbooking</i> .	
<i>From (variable.part.element)</i>	<i>To (variable.part.element)</i>
<i>input.input.cardNumber</i>	<i>creditCardPLRequest.request.cardNumber</i>
<i>input.input.cardType</i>	<i>creditCardPLRequest.request.cardType</i>
<i>input.input.airline</i>	<i>checkFlightPLRequest.request.airline</i>
<i>input.input.source</i>	<i>checkFlightPLRequest.request.source</i>
<i>input.input.target</i>	<i>checkFlightPLRequest.request.target</i>
<i>input.input.departure</i>	<i>checkFlightPLRequest.request.departure</i>
<i>input.input.Arrival</i>	<i>checkFlightPLRequest.request.arrival</i>
<i>input.input.cardNumber</i>	<i>checkFlightPLRequest.request.cardNumber</i>
<i>input.input.cardType</i>	<i>checkFlightPLRequest.request.cardType</i>
<i>input.input.hotelCompany</i>	<i>hotelReservationPLRequest.request.name</i>
<i>input.input.target</i>	<i>hotelReservationPLRequest.request.city</i>
<i>input.input.hotelCategory</i>	<i>hotelReservationPLRequest.request.category</i>
<i>input.input.from</i>	<i>hotelReservationPLRequest.request.from</i>
<i>input.input.to</i>	<i>hotelReservationPLRequest.request.to</i>
<i>input.input.cardNumber</i>	<i>hotelReservationPLRequest.request.cardNumber</i>
<i>input.input.cardType</i>	<i>hotelReservationPLRequest.request.cardType</i>
<i>input.input.carCompany</i>	<i>carReservationPLRequest.request.company</i>
<i>input.input.carCategory</i>	<i>carReservationPLRequest.request.category</i>
<i>input.input.from</i>	<i>carReservationPLRequest.request.from</i>
<i>input.input.to</i>	<i>carReservationPLRequest.request.to</i>

<i>input.input.cardNumber</i>	<i>carReservationPLRequest.request.cardNumber</i>
<i>input.input.cardType</i>	<i>carReservationPLRequest.request.cardType</i>
<i>carReservationPLResponse.response. carConfirmation</i>	<i>confirmInput.confirmRequest.carConfirmation</i>
<i>hotelReservationPLResponse.response. doHotelReservationReturn</i>	<i>confirmInput.confirmRequest.hotelConfirmation</i>
<i>creditCardPLResponse.response. doCreditCardCheckingReturn</i>	<i>confirmInput.confirmRequest.cardConfirmation</i>
<i>checkFlightPLResponse.response. doFlightReservationReturn</i>	<i>confirmInput.confirmRequest.flightConfirmation</i>

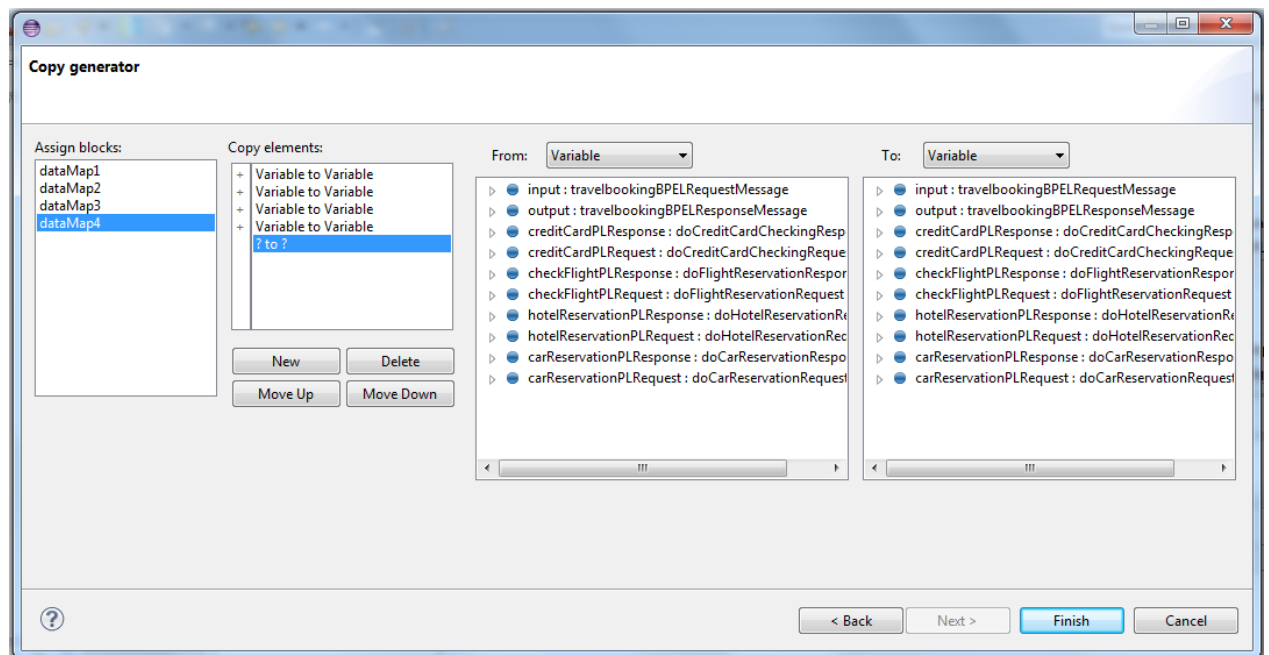
W tabeli 7.3 przedstawiono listę instrukcji kopiujących wymaganych w procesie *travelbooking* – jest to lista instrukcji, które projektant powinien uzupełnić, aby zapewnić w procesie BPEL komunikację – kopiowanie wartości z parametru wywołania procesu do parametrów wywołań poszczególnych usług zewnętrznych biorących udział w procesie.

Kolorem zielonym oznaczono instrukcje kopiujące wygenerowane przez zrealizowaną w ramach niniejszej pracy wtyczkę. Kolorem różowym oznaczono te instrukcje kopiujące, które nie zostały wygenerowane. Można zauważyć, że nazwy pól w instrukcjach niewygenerowanych mają różne nazwy. W tabeli brak jest instrukcji kopiujących, które nie zostały wygenerowane, a posiadają pola o tych samych nazwach – podstawowe wymaganie funkcjonalne zostaje zatem spełnione.

[TC2] Dodawanie nowej instrukcji kopiującej przy użyciu *wizarda* oferowanego przez dostarczoną wtyczkę.

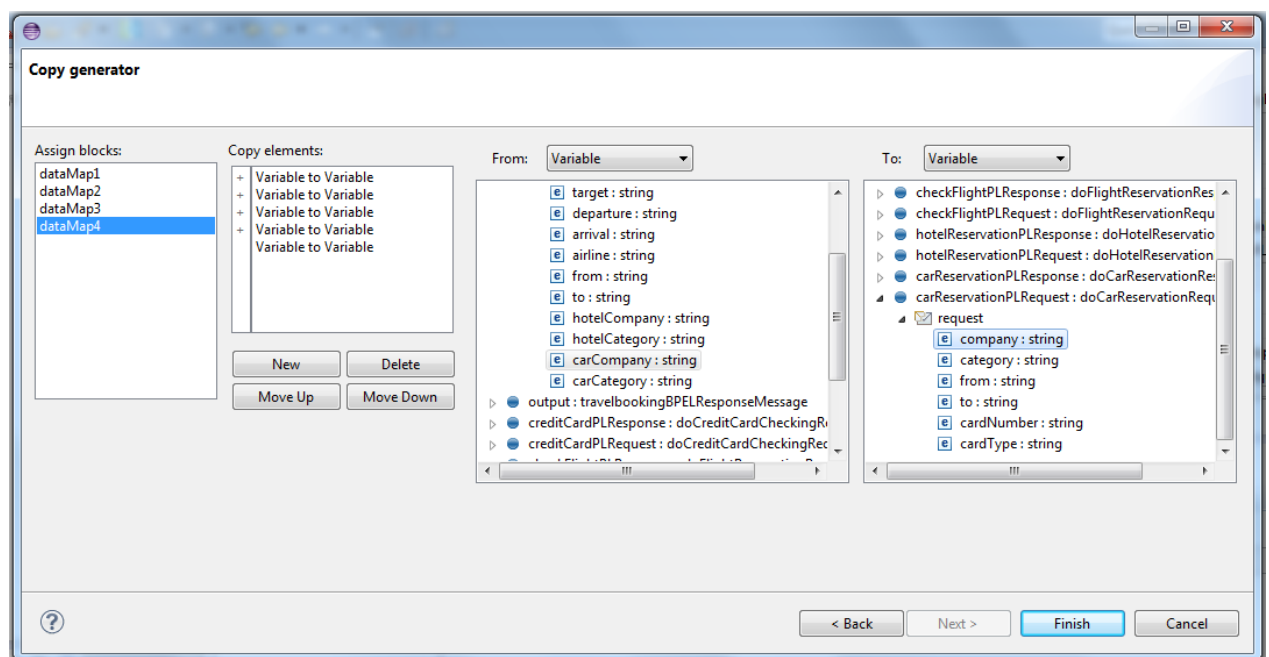
Po procesie wygenerowania instrukcji kopiujących na podstawie analizy procesu możliwa jest edycja wszystkich instrukcji *Assign* w procesie za pośrednictwem *wizarda*. TC2 ma za zadanie zweryfikować poprawność dodania instrukcji kopiującej do procesu BPEL.





Rysunek 7.2 Dodanie instrukcji kopiującej.

Przez kliknięcie przycisku „New” wtyczka dodaje nową instrukcję kopiującą, którą należy skonfigurować poprzez wybranie elementów składowych *From* oraz *To* z dostarczonej listy zmiennych procesu.



Rysunek 7.3 Konfiguracja nowo dodanej instrukcji kopiującej.

Po zakończonej konfiguracji oraz zakończeniu edycji procesu, zmiany zostają zapisane do pliku procesu. W efekcie dodana została instrukcja kopiująca przedstawiona na listingu 7.3.

Listing 7.3 Instrukcja *Copy* dodana przy użyciu *wizarda* – manualnie.

```
<process>
...
<sequence name="travelBooking">
...
  <flow>
...
    <bpel:assign name="dataMap4" validate="no">
...
      <copy>
        <from part="input" variable="input">
          <query queryLanguage="...xpath1.0">
            <![CDATA[tns:carCompany]]>
          </query>
        </from>
        <to part="request" variable="carReservationPLRequest">
          <query queryLanguage="...xpath1.0">
            <![CDATA[ns3:company]]>
          </query>
        </to>
      </copy>
...
    </bpel:assign>
...
  </sequence>
</process>
```

## 8. Podsumowanie.

Stworzona w ramach niniejszej pracy wtyczka do zintegrowanego środowiska programistycznego Eclipse daje możliwość wygenerowania instrukcji kopiujących w procesie biznesowym projektowanym z użyciem Eclipse BPEL Designer. Dodatkowo umożliwia edycję wszystkich instrukcji *Assign* procesu na jednym widoku, czego Eclipse BPEL Designer wcześniej nie umożliwiał.

Prace nad stworzeniem narzędzia pozwoliły na zapoznanie się ze środowiskiem programistycznym Eclipse – był to mój pierwszy kontakt z tym *IDE* – oraz narzędziem Eclipse BPEL Designer stanowiącym rozszerzenie środowiska Eclipse w postaci wtyczki, które wspiera projektowanie procesów biznesowych w języku BPEL. Ponadto budując narzędzie automatyzujące przepływ danych w procesie BPEL nauczyłem się rozszerzać środowisko Eclipse oraz budować własne narzędzia bazujące na tym środowisku. Opracowana wtyczka wprowadza jednak pewne ograniczenia w stosunku do możliwości edycji w Eclipse BPEL Designer. W obecnej wersji wtyczka oferuje wyszukiwanie dopasowań jedynie pomiędzy zmiennymi procesu, <nie ma partner link reference blablabla>. Identyczne ograniczenie istnieje przy edycji procesu przy użyciu dostarczonego *wizarda* umożliwiającego przegląd i edycję instrukcji *Assign* procesu BPEL. Głównymi osiągnięciami podczas tworzenia narzędzia były:

- Odczyt i zapis procesu BPEL oraz wczytanie definicji typów złożonych biorących udział w komunikacji z usługami sieciowymi z dokumentów definiujących interfejsy usług – WSDL – przy użyciu stworzonych elementów dostępu do danych – DAO.
- Transformacja wczytanego procesu BPEL z postaci EMF do postaci grafu.
- Analiza i przeszukiwanie grafowej postaci procesu pod kątem dopasowań pomiędzy zmiennymi biorącymi udział w procesie BPEL.
- Enkapsulacja mechanizmu szukającego dopasowań – wprowadzając możliwość łatwego rozszerzenia wtyczki o bardziej zaawansowane metody wyszukiwania dopasowań.
- Stworzenie możliwości edycji wszystkich instrukcji *Assign* procesu BPEL w jednym miejscu – *wizard*.

Jak wykazały przeprowadzone testy funkcjonalne wtyczki, spełnia ona wymagania funkcjonalne przedstawione w rozdziale 2 przy nałożonych ograniczeniach. Również pozytywnym aspektem jest duży zysk czasowy związany z zastąpieniem części manualnej pracy automatyczną generacją, która jest znacznie szybsza.

### 8.1. Napotkane problemy.

Podczas fazy koncepcyjnej pracy inżynierskiej, problem stanowiło znalezienie propozycji rozwiązania wprowadzającego innowacje dla narzędzi służących do

projektowania BPEL. W czasie trwania projektu problematyczne również było uzyskiwanie pomocy ze strony społeczności Eclipse BPEL Designer, pytania często pozostawione bez odpowiedzi przez dłuższy czas – ostatecznie odpowiedzi udzielałem sam.

Na etapie implementacji funkcji transformującej proces BPEL z postaci EMF do grafu dosyć problematyczne okazało się transformowanie elementu *Flow* procesu. Specyficzne dla tej instrukcji języka BPEL równoległe przebiegi w grafie musiały być poprzedzane oraz zakończone tą samą parą węzłów otwierającego oraz zamykającego.

Nierozwiązany został problem dotyczący modelu EMF procesu BPEL wtyczki Eclipse BPEL Designer'a (*org.eclipse.bpel.model*) związany z implementacją instrukcji *From* oraz *To*. Elementy te posiadają pole (*partName*) przechowujące nazwę elementu *Part* typu złożonego zdefiniowanego w dokumencie WSDL. Dostęp do pola może odbywać się jednokierunkowo (*setPartName*) natomiast brakuje (*getPartName*). Z tego powodu w trakcie aktualizacji instrukcji *Assign* w procesie – jeśli wygenerowana instrukcja kopiująca już występuje w instrukcji *Assign* nie jest do niej dodawana – nie są one sprawdzane pod kątem zgodności elementu *PartName*, dlatego w obecnej wersji narzędzia nie zostaną dodane instrukcje kopiujące, których elementy *From* i *To* różnią się jedynie elementem *PartName*.

## 8.2. Możliwości rozwoju.

Wtyczka ze względu na swoje innowacyjne podejście w procesie projektowania BPEL ma duże szanse w przyszłości zostać wykorzystaną na szerszą skalę, dlatego powinna być dalej rozwijana. Oto kilka propozycji rozwoju opracowanego narzędzia, które pozytywnie wpłyną na jego funkcjonalność:

- Opracowanie bardziej zaawansowanych mechanizm dopasowania zmiennych – można to osiągnąć w łatwy sposób dzięki wprowadzonej w projekcie enkapsulacji tego mechanizmu.
- Opracowanie bardziej optymalnego algorytmu analizy procesu w postaci grafu dostarczonego przez *Transformer*.
- Rozszerzenie możliwości analizatora o wyszukiwanie innego rodzaju dopasowań – w obecnej wersji wyszukiwane są dopasowania *Variable-Variable* (zgodnie z widokiem konfiguracyjnym instrukcji *Assign* Eclipse BPEL Designer) – m.in. *Property of a Variable* – *Property of a Variable* lub *Partner Link Reference* – *Partner Link Reference*.
- Rozszerzenie wizarda opracowanej wtyczki o możliwość edycji/dodawania nowych instrukcji kopiujących pomiędzy elementami innymi niż *Variable-Variable*, np. *Property of a Variable* – *Property of a Variable* lub *Partner Link Reference* – *Partner Link Reference*.

## 9. Bibliografia.

- [1] <http://www.eclipse.org/bpel/>
- [2] Andrzej Ratkowski. Projektowanie transformacyjne procesów w architekturze usługowej, 2011.
- [3] <http://pic.dhe.ibm.com/infocenter/adiehelp/v5r1m1/index.jsp?topic=%2Fcom.ibm.etools.ctc.bpel.doc%2Fsamples%2Ftravelbooking%2FtravelBooking.html>
- [4] <http://www.slideshare.net/milliger/eclipse-bpel-designer-presentation-765528>
- [5] Nick Boldt and Dave Steinber. Introduction to the Eclipse Modeling Framework, eclipseCON 2006.
- [6] Siran Chen. Extraction of BPEL Process Fragments in Eclipse BPEL Designer. Stuttgart, 2009.
- [7] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First. Design Patterns. 2004 O'Reilly Media, Inc.
- [8] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 1.2. W3C, 2002.
- [9] Krzysztof Stencel WSDL (Web Services Description Language) at <http://stencel.mimuw.edu.pl/abwi/20020514.WSDL/>
- [10] A WSDL description of a "stock quote" service at <http://cs.au.dk/~amoeller/WWW/webservices/wsdlexample.html>
- [11] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn XML Schema Part 1: Structures Second Edition. W3C Recommendation 28 October 2004.
- [12] David C. Fallside, Priscilla Walmsley – Second Edition. XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004
- [13] OASIS. Web Services Business Process Execution Language Version 2.0 at <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>
- [14] Charlton Barreto, Vaughn Bullard, Thomas Erl, John Evdemon, Diane Jordan, Khanderao Kand, Dieter König, Simon Moser, Ralph Stout, Ron Ten-Hove, Ivana Trickovic, Danny van der Rijn, Alex Yiu. Web Services Business Process Execution Language Version 2.0 Primer. 9 May 2007.
- [15] Eclipse Modeling Framework Project (EMF) at <http://www.eclipse.org/modeling/emf/>
- [16] Eclipse documentation – Eclipse Juno at <http://help.eclipse.org/juno/index.jsp>

## 10. Załączniki.

- Płyta CD zawierająca opracowaną wtyczkę generującą instrukcje kopiujące w procesie BPEL (BPELcg).
- Instrukcja instalacji wtyczki BPELcg.
- Dokumentacja projektu wtyczki BPELcg.

### 10.1. Płyta CD.

Załączona płyta CD zawiera:

- Wersję instalacyjną wtyczki (katalog *install\* ).
- Kod źródłowy projektu (katalog *source\* ).

### 10.2. Instrukcja instalacji wtyczki BPELcg (*BPEL copy generator*).

// TODO

### 10.3. Dokumentacja wtyczki BPELcg (*BPEL copy generator*).

// TOGENERATE from JavaDoc