



A G H

**AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

Projekt dyplomowy

*Projekt i implementacja oprogramowania
generującego modele procesów i decyzji
na podstawie diagramów zależności między atrybutami*

*Design and Implementation of Software for Process and Decision
Models Generation based on Attribute Relationship Diagrams*

Autor: *Karol Grzesiak*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Krzysztof Kluza*

Kraków, 2020

Chciałbym serdecznie podziękować doktorowi Krzysztofowi Kluzie za profesjonalizm oraz wsparcie merytoryczne podczas pisania niniejszej pracy. Dodatkowo kieruję podziękowania dla całej rodziny oraz przyjaciół za cierpliwość w tym okresie.

Spis treści

1. Wprowadzenie	7
1.1. Motywacja	7
1.2. Cel pracy	8
1.3. Struktura pracy	8
2. Metody reprezentacji procesów i decyzji.....	9
2.1. Business Process Model and Notation	9
2.2. Decision Model and Notation.....	11
2.3. Attribute Relationship Diagrams	14
2.3.1. Hekate Markup Langauge	15
3. Projekt aplikacji.....	19
3.1. Podstawowe przypadki użycia.....	19
3.2. Architektura systemu.....	20
3.2.1. Architektura N-tier.....	20
3.2.2. Architektura klient-serwer	22
3.3. Schemat projektu	25
3.4. Komponenty	25
3.5. Schemat interakcji	28
4. Implementacja.....	31
4.1. Stos technologiczny	31
4.1.1. Technologie front-end	31
4.1.2. Technologie back-end	32
4.1.3. Zewnętrzny System.....	34
4.2. Schemat bazy danych	34
4.3. Schemat klas	36
5. Ewaluacja.....	39
5.1. Model ARD	39
5.2. Interfejs użytkownika	39

5.3. Camunda.....	43
5.4. Podsumowanie wyników.....	49
6. Podsumowanie.....	51
6.1. Wnioski.....	51
6.2. Możliwe kierunki rozwoju aplikacji.....	52

1. Wprowadzenie

BPMN (Business Process Model and Notation [1]) oraz *DMN (Decision Model and Notation [2])* są powszechnie przyjętymi standardami służącymi do modelowania, opisywania oraz zarządzania procesami biznesowymi i decyjami. Notacja BPMN służy głównie do graficznej reprezentacji procesów, natomiast DMN skupia się na enkapsulacji logiki decyzyjnej (reguł organizacji).

Model procesu zbudowany przy użyciu tych dwóch standardów posiada dużą siłę ekspresji i może przynieść wiele korzyści dla firm używających systemów zarządzania wiedzą. Problem pojawia się jednak przy modelowaniu. Zajmują się tym głównie analitycy biznesowi, którzy, korzystając z wiedzy biznesowej, używają swojego doświadczenia oraz umiejętności. Niestety sam proces pozyskiwania modeli bywa trudny do dokładnego zdefiniowania, przez co może być odbierany jako sławny efekt *ATAMO*¹. Innymi słowy, proces prototypowania modeli procesów nie jest łatwym zadaniem.

Jednym z możliwych rozwiązań jest wykorzystanie *ARD (Attribute Relationship Diagrams [3])* – metody reprezentacji wiedzy dla ustrukturyzowanej specyfikacji systemu – do stworzenia prototypów modeli procesów, a następnie na tej podstawie wygenerowania odpowiednich diagramów w notacjach *BPMN* oraz *DMN* [4].

1.1. Motywacja

Mając na uwadze szybki rozwój zarządzania procesami biznesowymi oraz związane z tym problemy i proponowane rozwiązania, tematyka generowania modeli jest aktualna, zatem próby implementacji generatorów modeli mogą być użyteczne w badaniach naukowych i przemyśle.

Metoda *ARD* służy do prostego prototypowania modeli systemów, lecz standardy *BPMN* oraz *DMN* nadal królują jako podstawowa oraz najbardziej zrozumiała forma reprezentacji procesów i decyzji. Użytecznym narzędziem byłby system przekształcający prototypy *ARD* zapisane w plikach *HML*² (*Hekate Markup Language [5]*) do reprezentacji w standardach *BPMN* oraz *DMN*, który następnie wdroży takie modele do zewnętrznego systemu posiadającego silniki procesowe oraz decyzyjne, co umożliwi ich uruchomienie oraz ewaluację.

¹ „And then a miracle occurs“ – fraza spopularyzowana przez kreskówki Sidney Harris, często używana w pracach związanych z BPM aby opisać działania, które występują ale są trudne do zdefiniowania.

² Mówiąc o plikach HML, będę miał na myśli wersję *ARDML*, jednak z opcjonalnym fragmentem *TPH*. Więcej pod adresem: <https://ai.ia.agh.edu.pl/wiki/hekate:hml1>.

1.2. Cel pracy

Celem pracy jest zaprojektowanie i zaimplementowanie aplikacji internetowej „DAR”³, będącej generatorem modeli procesów biznesowych w notacji *BPMN* oraz decyzji w notacji *DMN* na bazie specyfikacji w postaci plików *HML*, opisujących diagramy zależności między atrybutami *ARD*. Docelowy system powinien zostać zintegrowany z wybranym środowiskiem wspierającym zarządzanie procesami biznesowymi oraz ewaluację decyzji, dzięki czemu umożliwia wygenerowanie modelu, a następnie wdrożenie go na opisywanym środowisku. Dodatkowo powinien umożliwiać opcje uzupełnienia niezbędnych informacji, aby następnie możliwe było uruchomienie wybranego procesu, analiza jego działania i obserwacja wyników.

1.3. Struktura pracy

Praca składa się z następujących rozdziałów:

- **Rozdział 2** – dokładniej opisuje pojęcia *BPMN*, *DMN*, *ARD* oraz wprowadza podstawowe koncepcje związane z dziedziną procesów biznesowych.
- **Rozdział 3** – przedstawia architekturę systemu w ujęciu abstrakcyjnym. Skupia się na schemacie działania, wydzielając odpowiednie komponenty.
- **Rozdział 4** – pokazuje aplikację od strony implementacyjnej. Omawia wybrane technologie, prezentuje stos technologiczny, przybliża strukturę występującą w bazie danych oraz schemat najważniejszych klas aplikacji.
- **Rozdział 5** – prezentuje działanie całej aplikacji na wybranym przykładzie, omawiając jednocześnie przebieg jej działania.
- **Rozdział 6** – kończy całą pracę, podsumowując działanie systemu i opisując zebrane spostrzeżenia oraz wnioski. Przedstawia możliwe kierunki rozwoju aplikacji.

³Anagram akronimu *ARD*.

2. Metody reprezentacji procesów i decyzji

Rozdział dokładniej opisuje pojęcia używane w tej pracy, które związane są z procesami biznesowymi. Główny nacisk położony jest na trzy notacje do reprezentacji procesów, decyzji i cech systemu.

2.1. Business Process Model and Notation

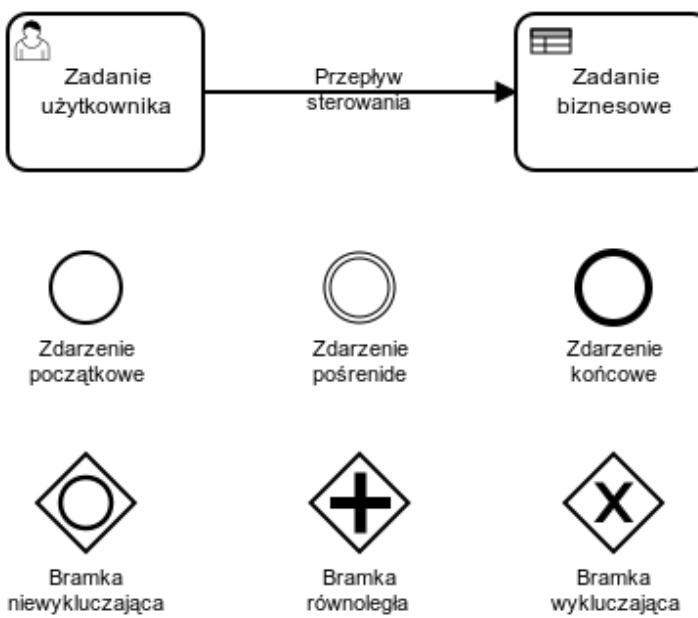
Proces biznesowy to seria działań lub zadań, których wynikiem jest określony rezultat. *BPMN (Business Process Model and Notation)* jest notacją stworzoną przez OMG (*Object Management Group*¹) w celu graficznej reprezentacji procesów biznesowych za pomocą diagramów stworzonych przy użyciu schematów blokowych. Motorem napędowym do stworzenia tej notacji była potrzeba medium zrozumiałego dla wszystkich użytkowników biznesowych. Tym sposobem notacja *BPMN* jest zrozumiała dla wszystkich interesariuszy, poczynając od analityka biznesowego tworzącego pierwsze szkice procesu, aż po programistów implementujących technologię odpowiedzialną za wykonywanie tych procesów.

BPMN 2.0 [1] jest w tym momencie najnowszą wersją notacji i zapewnia ona cztery typy diagramów, aby w pełni opisać różne aspekty procesów biznesowych:

- **Diagram Kooperacji,**
- **Diagram Współpracy,**
- **Diagram Choreografii,**
- **Diagram Procesów.**

Z perspektywy tej pracy, najbardziej interesujący jest ostatni typ diagramów. Jest on najbardziej elementarnym typem i reprezentuje przepływ sterowania procesu – w jakiej kolejności będą wykonywane zadania, podział procesu na osobne przepływy i sytuacje, które mogą się wydarzyć podczas wykonywania procesu. Poniżej opisane zostały podstawowe elementy, z których może składać się taki diagram [6], natomiast rysunek 2.1 przedstawia graficzną reprezentację opisanych elementów:

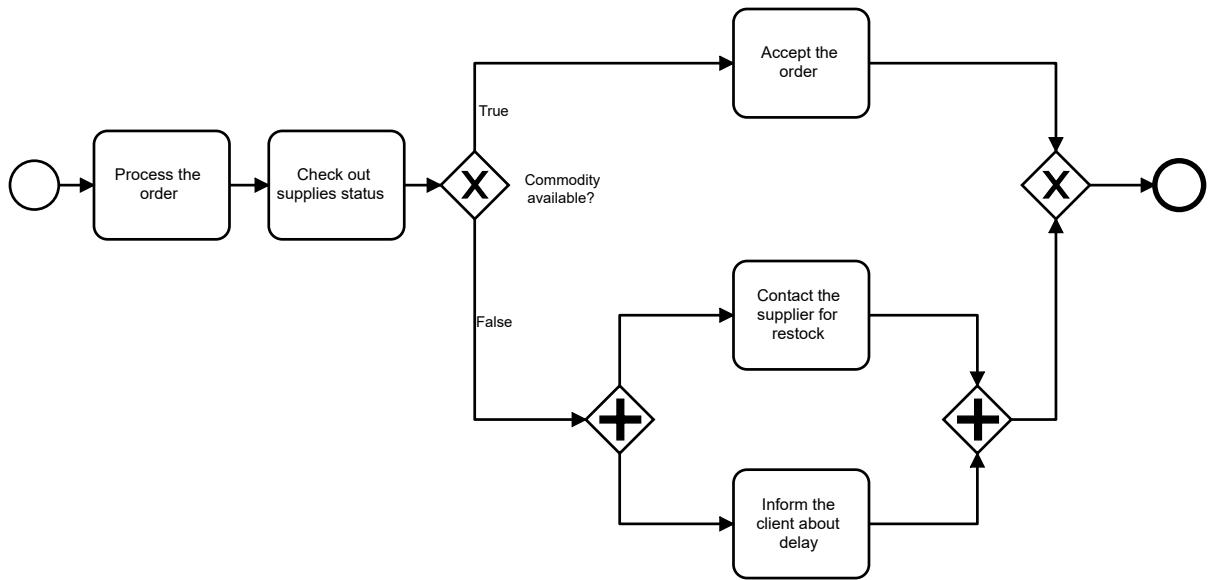
¹Zobacz: <https://www.omg.org/>.



Rys. 2.1. Graficzna reprezentacja podstawowych elementów BPMN

- **Zdarzenia** – reprezentują zdarzenia podczas procesu, np. rozpoczęcie procesu. Występują trzy główne typy zdarzeń: początkowe, pośrednie oraz końcowe.
- **Zadania** – reprezentują prace wykonywaną podczas procesu. Może to być przykładowo zadanie regułowe odpowiedzialne za podjęcie decyzji (połączone z tabelą decyzyjną w notacji DMN) albo zadanie użytkownika związanego z podaniem pewnych danych.
- **Bramki** – determinują rozwidlenie i łączenie przepływu w procesie. Ich reprezentacja graficzna posiada dodatkowe znaczniki, mówiące o tym, w jaki sposób przepływ jest kontrolowany:
 - **Bramka równoległa** – odpowiednik funkcji „AND”.
 - **Bramka niewykluczająca** – odpowiednik funkcji „OR”.
 - **Bramka wykluczająca** – odpowiednik funkcji „XOR”.
- **Przepływy sterowania** – obrazują relację (połączenia między obiektami).

Rysunek 2.2 pokazuje przykładowy model procesu w notacji BPMN. Jest on generalnie zrozumiały i łatwy do analizowania. Ilustruje mały wycinek procesu związanego obsługą złożonego zamówienia przez klienta. Zdarzenie startowe rozpoczyna cały proces. Następnie wykonywane są po kolej i dwa zadania. W zależności od danych, które dostarczą te elementy, bramka wykluczająca rozdziela przepływ na dwa. Przepływ górnego posiada jedno zadanie, natomiast na przepływie dolnym znowu dochodzi do podziału, tym razem za pomocą bramki równoległej, gdzie wszystkie następujące elementy są wykonywane równolegle. Finalnie przepływy łączą się i proces jest zakończony przez zdarzenie końcowe.



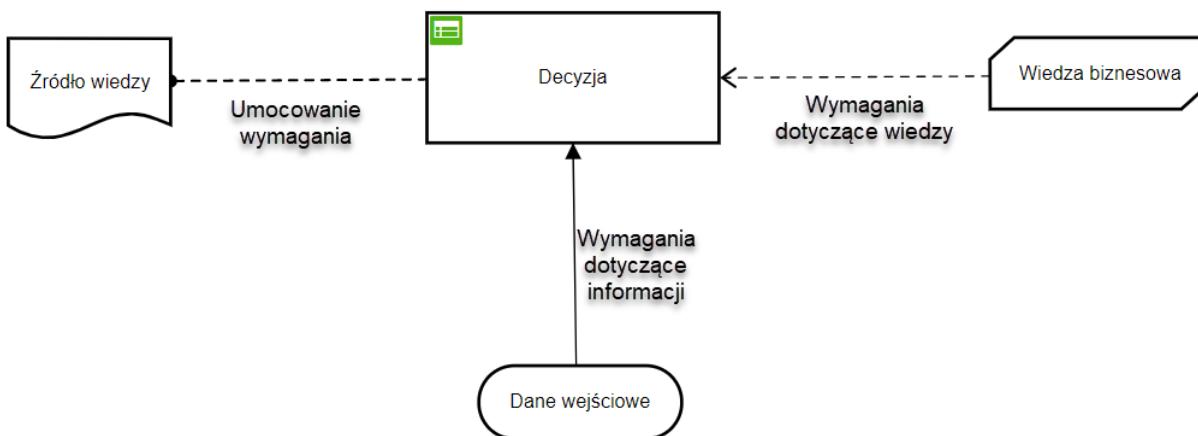
Rys. 2.2. Przykładowy proces BPMN

2.2. Decision Model and Notation

DMN (Decision Model and Notation) jest notacją stworzoną przez OMG w celu prostego opisu oraz modelowania reguł występujących w procesach i organizacjach. Tak samo jak w przypadku *BPMN* główną zaletą oraz założeniem tego standardu było umożliwienie prostej komunikacji użytkowników z dziedzin biznesu oraz IT. Dzięki niemu bezproblemowa staje się współpraca pomiędzy ludźmi biznesu monitorującymi i zarządzającymi decyzjami, analitykami biznesowymi, którzy szkicują wstępne wymagania decyzyjne wraz z logiką decyzyjną oraz technicznymi programistami odpowiedzialnymi za implementację systemów, które te decyzje podejmują. Standard ten może być używany jako samodzielny, jednak najczęściej towarzyszy on notacji *BPMN*. *BPMN* definiuje zadanie regułowe, które może być połączone z odpowiednim obiektem w notacji *DMN*. Dzięki temu reguły biznesowe mogą być wcielone do wykonywalnych procesów i wraz z działaniem przetwarzają dane oraz odpowiednio ewaluować decyzje.

DMN 1.2 [2] jest w tym momencie najnowszą wersją notacji i definiuje on cztery elementy:

- **Diagram wymagań decyzyjnych** – diagram pokazujący zależności między elementami w notacji *DMN*. Określa on wymagania dla logiki decyzyjnej.
- **Kontekst biznesowy** – kontekst decyzji, np. jak duży wpływ mają decyzje na wskaźniki wydajności lub jaka jest ich rola.
- **FEEL (Friendly Enough Expression Language)** – język służący do określania reguł biznesowych w tabelach decyzyjnych lub innych logicznych formatach.
- **Logika decyzyjna** – logika będąca wewnętrz obiektów występujących w *diagramie wymagań decyzyjnych*. W kontekście tej pracy głównie będzie się to odnosić do tabel decyzyjnych.



Rys. 2.3. Graficzna reprezentacja elementów diagramu wymagań decyzyjnych

Diagram wymagań decyzyjnych jest zbudowany bardzo podobnie do diagramu w notacji *BPMN*. Podstawowe obiekty, z których się składa opisane zostały poniżej [7], natomiast rysunek 2.3 przedstawia ich graficzną reprezentację:

– Elementy

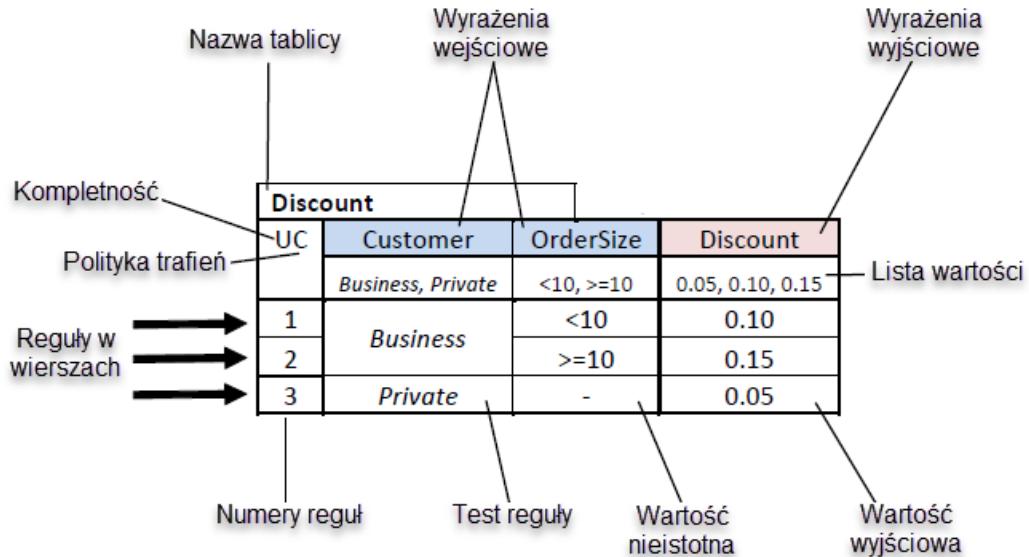
- **Decyzje** – elementy determinujące wynik na podstawie pewnej ilości danych poprzez aplikowanie logiki decyzyjnej.
- **Dane wejściowe** – zewnętrzne dane, niepochodzące z kontekstu biznesowego.
- **Źródła wiedzy** – źródła określające reguły podejmowania decyzji. Przykładem może być tutaj polityka firmy lub regulacja.
- **Wiedza biznesowa** – pewne funkcje, które enkapsulują logikę decyzyjną (reguły biznesowe, tablice decyzyjne). Powinny one być łatwe do wielokrotnego użycia.

– Wymagania

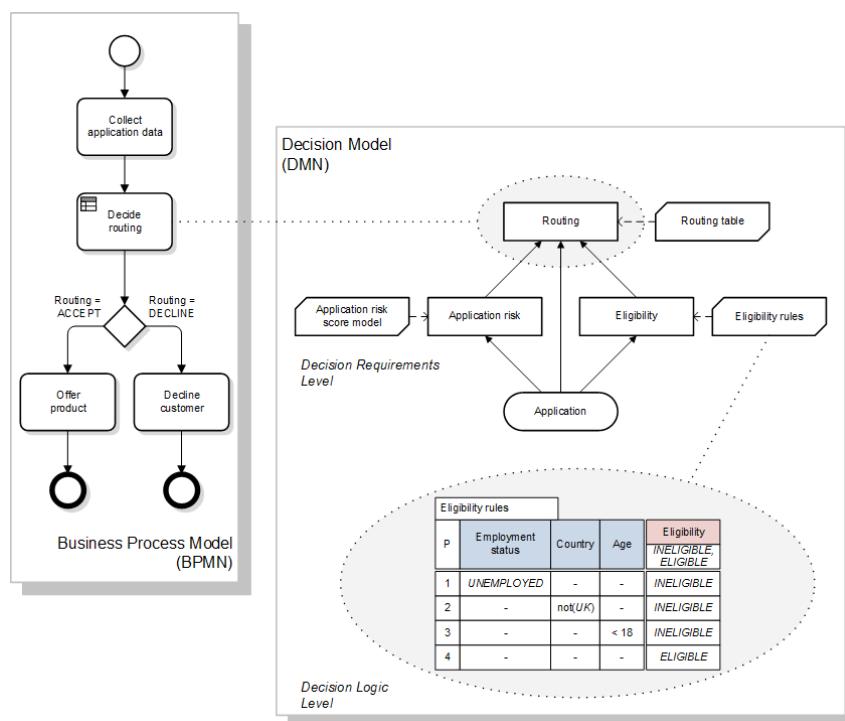
- **Wymagania dotyczące informacji** – połączenia wskazujące dane wejściowe oraz decyzje.
- **Wymagania dotyczące wiedzy** – połączenia wskazujące jaki model wiedzy biznesowej będzie wykorzystywany w procesie podejmowania decyzji.
- **Umocowanie wymagania** – połączenia wskazujące, które z elementów są źródłami wiedzy.

Rysunek 2.4 opisuje przykładową tablicę decyzyjną. Warto zauważyć użycie *FEEL* – przykładowo w kolumnie OrderSize można w prosty sposób uzależnić zwracaną wartość od tego, czy dostarczone dane będą mniejsze lub większe od liczby dziesięć.

Rysunek 2.5 prezentuje jak w praktyce wygląda integracja opisanych standardów. Zadanie regułowe w procesie zbudowanym za pomocą notacji *BPMN* jest połączone z decyzją w notacji *DMN*. Ta z kolei posiada pewną logikę – reguły biznesowe wyrażone za pomocą tabeli decyzyjnej i na tej podstawie, podczas działania procesu, ewaluuje decyzję.



Rys. 2.4. Tablica decyzyjna w notacji DMN [8]



Rys. 2.5. Integracja modelu BPMN oraz DMN [2]

2.3. Attribute Relationship Diagrams

ARD (Attribute Relationship Diagrams) to metoda, której celem jest przedstawienie relacji pomiędzy pewnymi atrybutami danego systemu. Wspomniane atrybuty są elementami branymi pod uwagę w przypadku rozważania logiki biznesowej. Z początku metoda *ARD* była zaproponowana jako mechanizm prototypowania baz wiedzy, głównie wykorzystywanych w silnikach regułowych [9]. Działanie miało być podobne do generowania struktur relacyjnych baz danych na podstawie diagramów ER² (*Entity-Relationship Diagrams*). W kontekście tej pracy *ARD* jest odpowiedzią na problem związany z prototypowaniem modeli procesów biznesowych. Oferuje ona prosty sposób na szybkie tworzenie szkiców, z których następnie system potrafi stworzyć bardziej skomplikowane struktury w notacjach *BPMN* oraz *DMN*. Proces tworzenia modelu *ARD* jest prosty i iteracyjny, co przyczynia się do zwiększenia szczegółowości modelu z każdym kolejnym krokiem. Całość opiera się na przechodzeniu od ogólnej koncepcji do bardzo szczegółowego modelu³, jednocześnie zachowując funkcjonalne zależności pomiędzy poszczególnymi elementami.

Diagram *ARD* jest zbiorem pewnych własności i zależności między nimi. Elementy, z których jest zbudowany to:

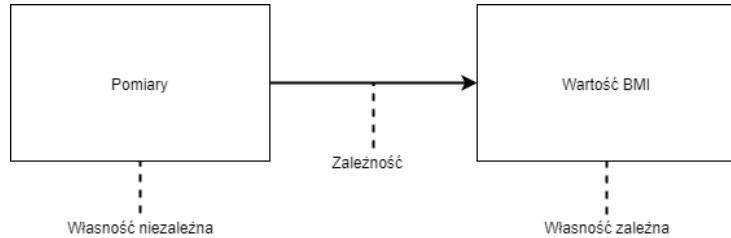
- **Własność** – własność, która reprezentuje pewne atrybuty. Właściwości można podzielić na proste i złożone. Każda zależność może być zależna (zależeć od innej właściwości) lub niezależna.
- **Prosta** – własność prosta reprezentuje jeden atrybut. Takiej właściwości nie da się podzielić na pewien zbiór właściwości, jest atomiczna. Przykładem takiej właściwości może być wiek lub wzrost, są to właściwości reprezentujące dokładnie jeden atrybut.
- **Złożona** – własność złożona, reprezentuje zbiór atrybutów. Da się ją podzielić na zbiór prostych właściwości. Przykładem takiej właściwości mogą być „pomiary” – jest to właściwość, która wewnętrznie może kryć wiele atrybutów, takich jak np. wzrost, waga.
- **Zależność** – zależność łączy ze sobą właściwości i pokazuje relację między nimi.

Rysunek 2.6 prezentuje prosty diagram *ARD*. Występują na nim dwie właściwości. „Pomiary” jest właściwością niezależną. Możliwym byłoby zastąpienie jej wartościami typu waga oraz wzrost. „Wartość BMI” jest właściwością zależną, więc nie jest możliwe ewaluowanie jej bez poprzedniego określenia właściwości źródłowych.

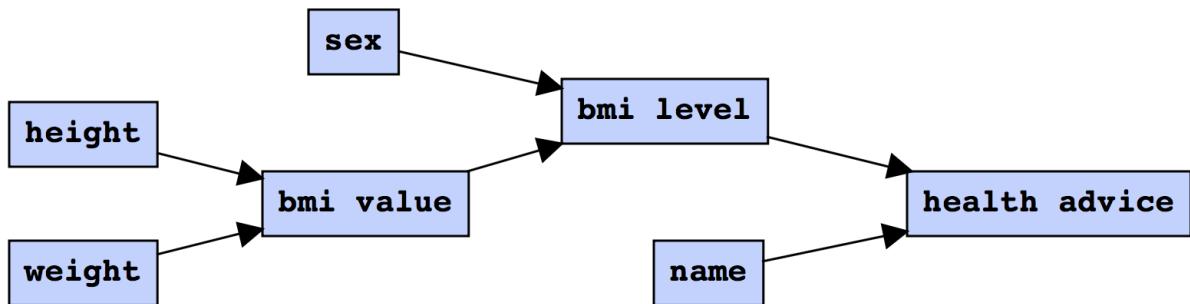
Jasno widać, że nietrudnym zadaniem byłoby rozbudowanie takiego diagramu o kolejne właściwości i to jest główną zaletą tej metody – prostota i iteracyjna natura. Aby lepiej zobrazować proces tworzenia diagramu *ARD* warto pochylić się nad konkretnym przykładem, wykorzystując jednocześnie diagram z rysunku 2.6. Niech przykładową sytuacją będzie potrzeba naszkicowania, przez analityka biznesowego, prototypu modelu związanego z poradami zdrowotnymi. Model ma opierać się na BMI (*Body*

²Więcej na temat *ERD*: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.526.369&rep=rep1&type=pdf>.

³Rozumowanie dedukcyjne – „od ogólnego do szczegółu”.



Rys. 2.6. Przykład prostego diagramu ARD



Rys. 2.7. Diagram ARD dla modelu związanego z poradą zdrowotną [4]

Mass Index⁴). Wykorzystując wcześniej wspomniany iteracyjny proces, finalną właściwością byłaby porada zdrowotna i z każdą kolejną iteracją analityk wprowadzałby coraz to nowsze i bardziej szczegółowe właściwości, budując w taki sposób diagram zależności. Porada potrzebowałaby imienia pacjenta oraz jego poziomu *BMI*. Do obliczenia poziomu *BMI* potrzebna by była płeć pacjenta oraz wartość *BMI*. Natomiast sama wartość byłaby obliczana na podstawie wagi oraz wzrostu. Finalny wykres i opisane rozumowanie obrazuje rysunek 2.7.

Wykorzystując dalej przykład z rysunku 2.7 oraz mając na uwadze wcześniej opisane standardy BPMN oraz DMN, można pokusić się o przeniesienie tego prototypu do wspomnianych notacji. Rysunek 2.8 prezentuje koncepcję takiej operacji. Wystarczyłoby odpowiednio pogrupować właściwości i ich zależności, przedstawić każdą właściwość prostą jako zadanie użytkownika, w którym użytkownik musi podać dane, a każdą właściwość złożoną jako zadanie biznesowe, czyli zadanie połączone z decyzją (gdzie wykorzystywana jest tablica decyzyjna). Na wspomnianym rysunku przykładem zadania użytkownika jest „Enter Measures”, które dostarcza „height” oraz „width” do tablic decyzyjnych, a przykładem zadania biznesowego jest „Determine bmi level”.

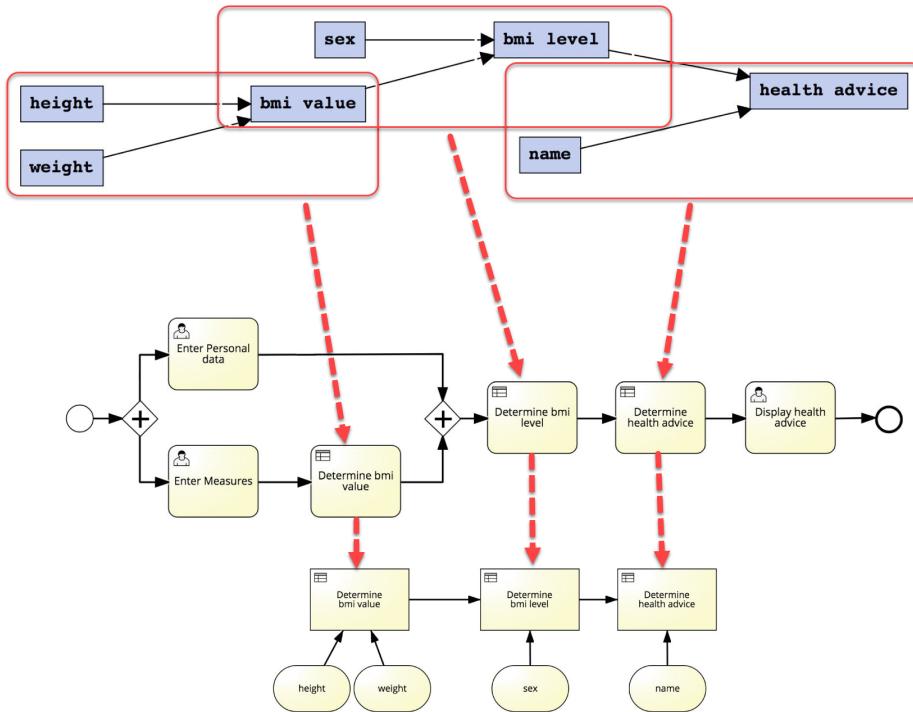
2.3.1. Hekate Markup Langauge

HML (Hekate Markup Language) to język stworzony do reprezentacji bazy reguł *HeKatE*⁵, zapisanych formacie HMR (*Hekate Meta Representation*⁶). Język *HML* posiada trzy podzbiorystyczne:

⁴Więcej na temat *BMI*: https://www.researchgate.net/publication/276444598_Body_Mass_Index.

⁵Więcej na temat *HeKatE*: <https://ai.ia.agh.edu.pl/hekate:start>.

⁶Więcej na temat *HMR*: <https://ai.ia.agh.edu.pl/hekate:hmr>.



Rys. 2.8. Koncept transformacji diagramu *ARD* do modelu w notacji *BPMN* oraz *DMN* [4]

- **attml** – Attribute Markup Language opisujący atrybuty reguł,
- **ardml** – Attribute Relationship Markup Language służący do ekspresji prototypów w *ARDplus* (rozszerzeniu *ARD*),
- **xttm** – XTT2 Rule Markup Language służący do reprezentacji ustrukturyzowanych reguł XTT2⁷.

W zależności od potrzeb, możliwe jest różnorakie wykorzystanie opisanych podjęzyków:

- **attml** – tylko definicje atrybutów (minimalistyczny przypadek),
- **attml + ardml** – atrybuty oraz zależności *ARD*,
- **attml + ardml + xttm** – atrybuty, zależności oraz reguły *XTT2*,
- **attml + xttm** – atrybuty i reguły (bez prototypu *ARD*).

⁷Więcej na temat XTT2: <https://ai.ia.agh.edu.pl/hekate:xtt2>.

Listing 2.1 prezentuje strukturę pliku *HML* wykorzystującego wszystkie wspomniane podjęzyki.

Listing 2.1. Przykład struktury pełnego pliku *HML*

```
<hml>
  <!-- attml zaczyna się tutaj -->
  <types>
    ...
  </types>
  <attributes>
    ...
  </attributes>
  <!-- attml kończy się tutaj -->

  <!-- ardml zaczyna się tutaj -->
  <properties>
    ...
  </properties>
  <tph>
    ...
  </tph>
  <ard>
    ...
  </ard>
  <!-- ardml kończy się tutaj -->

  <!-- xttml zaczyna się tutaj -->
  <xtt>
    ...
  </xtt>
  <!-- xttml kończy się tutaj -->

  <!-- dodatkowe specyfikacje systemu -->
  <states>
    ...
  </states>
</hml>
```

Składnia wymaga, aby wszystkie ważne elementy posiadały atrybut „id”, będący unikatowym identyfikatorem. Dodatkowo każdy element powinien zawierać odpowiedni przedrostek:

- `types` – identyfikator powinien zaczynać się od „`tpe`”,
- `attributes` – identyfikator powinien zaczynać się od „`att`”,
- `properties` – identyfikator powinien zaczynać się od „`prp`”,
- `type groups` – identyfikator powinien zaczynać się od „`tgr`”,
- `attribute groups` – identyfikator powinien zaczynać się od „`agr`”,
- `dependencies` – identyfikator powinien zaczynać się od „`dep`”,
- `ARD history` – identyfikator powinien zaczynać się od „`hst`”.

W kontekście niniejszej pracy pliki *HML*, które będą brane pod uwagę, powinny zawierać w sobie podzbiór językowy *ardml*, jednak z opcjonalnym fragmentem *TPH*, gdyż nie jest on wykorzystywany przy tworzeniu modelu procesu finalnego. Przykład wymaganego pliku prezentuje listing 2.2.

Listing 2.2. Przykład struktury pliku *HML* z podzbiorem *ardml*

```

<hml>
  <types>
    <type id="tpe_1" name="Temperature" base="numeric" length="5">
      <desc>Reprezentuje temperaturę</desc>
      <domain>
        <value from="1" to="5"/>
      </domain>
    </type>
    <type id="tpe_2" name="Integer" base="numeric" length="5">
      <desc>Reprezentuje numer</desc>
      <domain>
        <value from="-10000" to="10000"/>
      </domain>
    </type>
  </types>
  <attributes>
    <attr name="Thermostat" id="att_0" type="tpe_1"/>
    <attr name="Something" id="att_1" type="tpe_2"/>
  </attributes>
  <properties>
    <property id="prp_1">
      <attref ref="att_0"/>
    </property>
    <property id="prp_2">
      <attref ref="att_0"/>
      <attref ref="att_1"/>
    </property>
  </properties>
  <ard>
    <dep id="dep_01" independent="prp_1" dependent="prp_2"/>
    <dep id="dep_02" independent="prp_2" dependent="prp_3"/>
    <dep id="dep_03" independent="prp_1" dependent="prp_4"/>
    <dep id="dep_04" independent="prp_2" dependent="prp_4"/>
  </ard>
</hml>

```

Jest to koniec rozdziału opisującego metody reprezentacji procesów i decyzji. W tym rozdziale opisane zostały najważniejsze terminy, które będą często pojawiać się w dalszej części niniejszej pracy. W kolejnym rozdziale zostanie przedstawiony projekt aplikacji, która wykorzystuje opisane tutaj notacje *BPMN*, *DMN* oraz *ARD*.

3. Projekt aplikacji

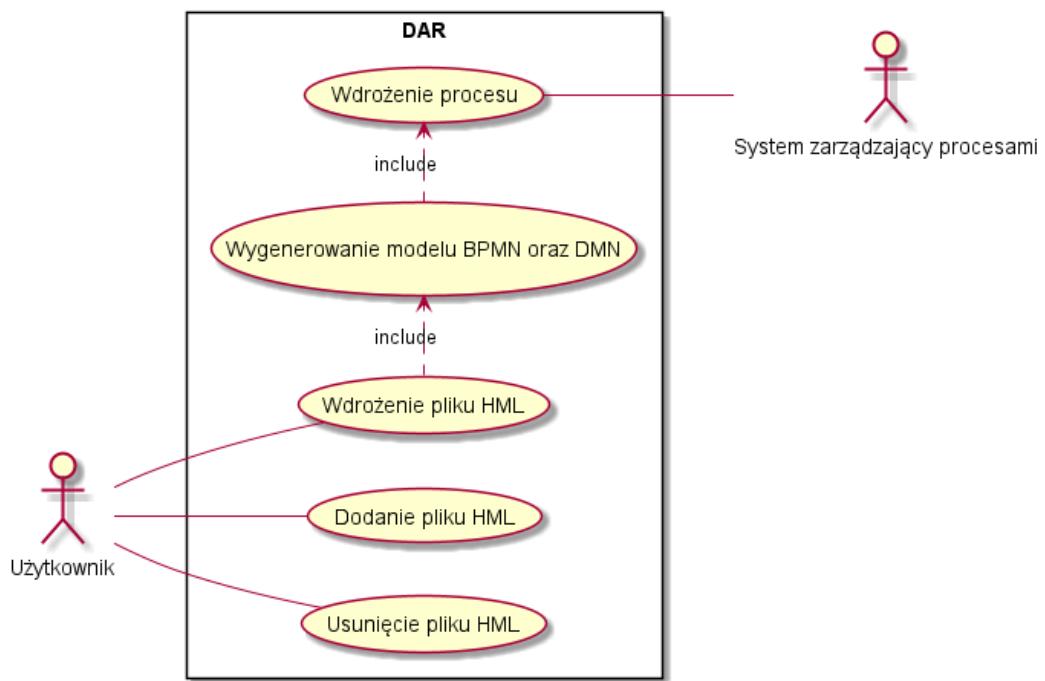
Rozdział opisuje architekturę systemu w ujęciu abstrakcyjnym. Nacisk położony jest na schemat działania oraz przedstawienie komponentów aplikacji. Dodatkowo wyjaśniane są pewne podstawowe pojęcia związane z inżynierią oprogramowania.

3.1. Podstawowe przypadki użycia

Aplikacja „DAR” jest z punktu widzenia funkcjonalności dość prostym systemem. Rysunek 3.1 prezentuje podstawowe przypadki użycia – warto mieć jednak na uwadze fakt, że aplikacja została zintegrowana z zewnętrznym systemem posiadającym silniki decyzyjne i procesowe, zatem wiele funkcjonalności niepokazanych tutaj, takich jak uruchamianie procesu, ewaluacja decyzji czy monitorowanie wydajności procesów, leży po jego stronie.

Poniżej opisane zostały funkcjonalności zaprezentowane na rysunku 3.1:

- **Wdrożenie procesu** – aplikacja przesyła dwa pliki do zewnętrznego systemu. Pierwszy z nich to plik reprezentujący proces w notacji *BPMN*, natomiast drugi to plik reprezentujący diagram decyzji w notacji *DMN*.
- **Wygenerowanie modelu BPMN oraz DMN** – aplikacja na podstawie dostarczonych danych generuje dwa modele – model w notacji *BPMN* oraz model w notacji *DMN*. Jednocześnie spaja te dwa modele poprzez odpowiednie ustawienia atrybutów.
- **Wdrożenie pliku HML** – aplikacja przesyła wybrany plik *HML* i rozpoczyna proces jego przetwarzania aby finalnie przesłać go do odrębnego systemu procesowego.
- **Dodanie pliku HML** – aplikacja zapisuje informację z plików *HML*.
- **Usunięcie pliku HML** – aplikacja usuwa wszystkie zapisane informacje z danego pliku *HML*.
- **Uruchamianie procesu, ewaluacja decyzji, monitorowanie procesu...** – wiele funkcjonalności nie należy stricte do ram aplikacji „DAR”, jednak poprzez integrację z zewnętrznym systemem, aplikacja pośredniczy w umożliwianiu takich działań, jak uruchamianie i śledzenie procesu, edycja tabel decyzyjnych, uzupełnianie danych czy ewaluacja decyzji.



Rys. 3.1. Graficzna reprezentacja podstawowych przypadków użycia

3.2. Architektura systemu

3.2.1. Architektura N-tier

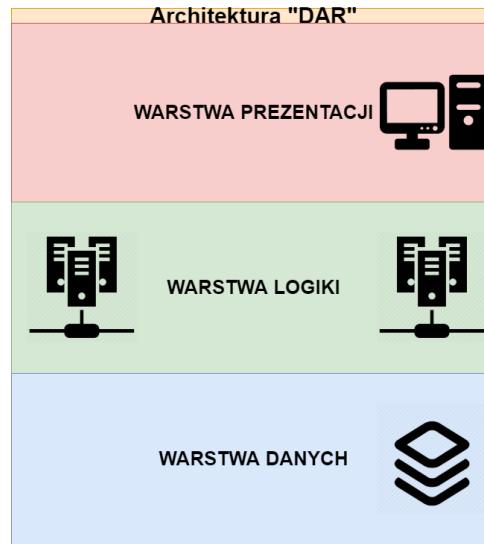
Aplikacja „DAR” została oparta na architekturze wielowarstwowej zwanej również architekturą N-tier. Jest to jedna z metod tworzenia oprogramowania, opierająca się na wydzieleniu warstw aplikacji – w sensie fizycznym oraz logicznym. Warstwy są narzędziem do odseparowania obowiązków¹ danych rejonów aplikacji oraz do zarządzania zależnościami. Każda warstwa posiada konkretną odpowiedzialność. Warstwy wyższe mogą używać usług z warstw niższych, jednak nie działa to w drugą stronę. Architektura wielowarstwowa może być:

- **Zamknięta** – tak jak jest to w przypadku opisywanego systemu. Dana warstwa może używać jedynie warstwy pod sobą.
- **Otwarta** – dana warstwa może używać dowolnej warstwy pod sobą.

„DAR” posiada trzy logiczne warstwy, zaprezentowane na rysunku 3.2. Poniżej przedstawiony został opis wydzielonych warstw:

- **Warstwa prezentacji** – najwyższa warstwa. Jej zadaniem jest wyświetlanie danych i komunikacja z warstwą niżej. Reprezentuje ona interfejs użytkownika i umożliwia interakcję z systemem.

¹Jest to ściśle związane z zasadą SoC (*Separation of Concerns*), czyli podstawową zasadą inżynierii oprogramowania, stanowiącą o tym, że każda część systemu powinna mieć silnie określone granice i adresować dokładnie odseparowaną kwestię.



Rys. 3.2. Warstwy aplikacji „DAR”

- **Warstwa logiki** – zwana również logiką biznesową, jest to główna część aplikacji, w której za-implementowane zostały wszystkie funkcjonalności. W przypadku „DAR” do dyspozycji są tutaj dwa serwisy, jeden będący serwerem „DAR”, natomiast drugi to zintegrowany zewnętrzny system do obsługi procesów. Warstwa ta przetwarza dane i przekazuje je do warstwy prezentacji.
- **Warstwa danych** – warstwa związana z dostępem do danych. W przypadku „DAR” jest to dostęp do relacyjnej bazy danych, gdzie przechowywane są informacje na temat dodanych plików *HTML*. Udostępnia ona dane dla warstwy logiki biznesowej.

Zaletami takiego rozwiązania są:

- **Skalowalność** – dzięki oddzieleniu odpowiedzialności bardzo łatwo zdiagnozować, gdzie aplikacja potrzebuje usprawnień i co najważniejsze, można to robić tylko w jednym konkretnym obszarze, co daje o wiele więcej możliwości skalowania.
- **Luźne powiązania** – architektura wymusza na programiście ograniczenie zależności, dzięki czemu zmiany w danej warstwie są bezbolesne z punktu widzenia innych warstw.
- **Bezpieczeństwo** – podział na warstwy niesie też ze sobą więcej możliwości ochrony przed atakami, ponieważ w każdej warstwie może występować inna metoda ochrony. Dodatkowo tak samo jak w przypadku skalowania można odpowiednio poświęcić środki na ochronę tylko newralgicznych punktów aplikacji (przykładem może być zwiększoną ochroną warstwy logiki biznesowej oraz danych niż warstwy prezentacji).
- **Łatwiejszy proces tworzenia aplikacji** – każdą warstwą może zajmować się inna osoba lub zespół, ze względu na luźne powiązania.
- **Większy nacisk na powtórne użycie** – dzięki modularności tego podejścia, komponenty są raczej projektowane w sposób generyczny, co ułatwia ich ponowne użycie.

3.2.2. Architektura klient-serwer

„DAR” jest aplikacją internetową działającą na zasadzie klient-serwer. Jest to metoda działania, która opiera się na podziale aplikacji na dwie części i na komunikacji tych modułów:

- **Front-end** – część kliencka, działająca na oprogramowaniu klienckim, w tym przypadku jest to przeglądarka użytkownika. Ta część reprezentuje interfejs użytkownika. W aplikacji „DAR” *front-end* został zrealizowany w formie SPA (*Single Page Application*²). Do tej części należy *warstwa prezentacji* opisywana wcześniej.
- **Back-end** – część serwerowa, działająca na osobnym serwerze. Do niej należą *warstwa logiki biznesowej* oraz *warstwa danych* opisane wcześniej. W przypadku systemu „DAR” została zrealizowana za pomocą internetowego REST³ API (*Application Programming Interface*).

W opisywanej aplikacji klient wysyła zapytanie *HTTP*⁴ przez sieć internetową do serwera, a ten przetwarza żądanie i zwraca odpowiedź, często z towarzyszącymi temu danymi. Komunikacja jest asynchroniczna, dzięki temu zwiększeniu jest odczuwalna szybkość działania aplikacji.

3.2.2.1. Protokół HTTP

Protokół *HTTP* (*Hypertext Transfer Protocol*) to bezstanowy protokół do przesyłania dokumentów typu hypermedia. Określa on zasady wymiany informacji, normalizuje i ujednolicia sposoby komunikacji pomiędzy urządzeniami. Jego głównym zastosowaniem jest umożliwienie komunikacji pomiędzy aplikacjami internetowymi (przeglądarkami), a serwerami (komputerami/chmurą). Wymiana informacji przebiega w następujący sposób:

- Klient (przeglądarka) wysyła zapytanie *HTTP* do sieci.
- Serwer internetowy otrzymuje zapytanie.
- Serwer uruchamia aplikację, aby przetworzyć zapytanie.
- Serwer zwraca odpowiedź *HTTP* do przeglądarki.
- Klient otrzymuje odpowiedź.

Rysunek 3.3 obrazuje wymianę informacji w protokole *HTTP* pomiędzy klientem a serwerem.,

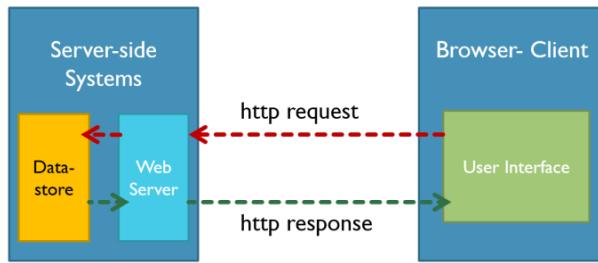
Klient, aby otrzymać odpowiedź od serwera, musi określić jego adres oraz metodę określającą jakiego typu jest zapytanie. Najważniejsze metody określone przez specyfikację *HTTP* [11] są następujące:

- **GET** – pobranie zasobu wskazanego za pomocą adresu.
- **PUT** – wysłanie danych, najczęściej aby zaktualizować pewne istniejące już dane.

²Więcej na temat SPA w rozdziale 3.2.2.3.

³Więcej na temat protokołu REST w rozdziale 3.2.2.2.

⁴Więcej na temat protokołu HTTP w rozdziale 3.2.2.1.



Rys. 3.3. Przykład komunikacji za pomocą protokołu *HTTP* [10]

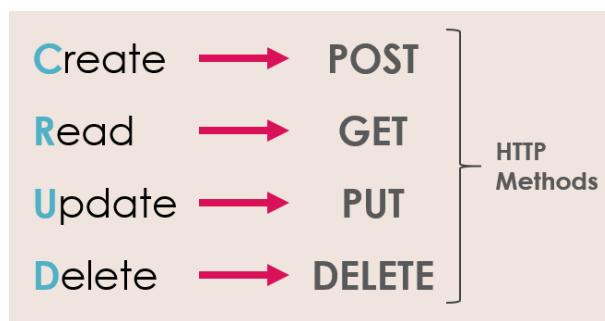
- **POST** – wysłanie danych, najczęściej aby stworzyć kompletnie nowy zasób.
- **DELETE** – usunięcie danych.

3.2.2.2. REST API

REST (Representational State Transfer) jest to styl architektury oprogramowania, bazujący na zbiorze określonych reguł, które określają, jak definiowane są zasoby, a co za tym idzie, jak można otrzymać do nich dostęp. *API (Application Programming Interface)* jest to zestaw pewnych zasad określających komunikację pomiędzy oprogramowaniem komputerowym. Innymi słowy, wiążąc te dwa pojęcia, *API* to reguły określające, jakie zasoby są dostępne i jak interesariusz może je uzyskać, natomiast *REST* to styl określający schemat oraz budowę *API*. Aby interfejs był w pełni zgodny z *REST*, czyli był *RESTful*, musi spełnić sześć podstawowych reguł z nim związanych:

- **Klient-serwer** – system powinien wspierać architekturę typu klient-serwer, jest to związane z poziomem odpowiedzialności. Odseparowując interfejs użytkownika od przechowywania danych zwiększa się modularność, przez co zyskuje na tym skalowalność.
- **Bezstanowość** – każda komunikacja na kanale klient-serwer jest w pełni samowystarczalna, posiada wszelkie niezbędne informacje. Żaden kontekst klienta nie jest przechowywany na serwerze pomiędzy zapytaniami.
- **Zdolność wykorzystania cache'a** – odpowiedź z *REST API* musi być jasno określona jako zdolna lub niezdolna do wykorzystania cache'a.
- **Zunifikowany interfejs** – punkt końcowy, czyli adres danego zasobu powinien być jednoznaczny. Użytkownik zawsze powinien wiedzieć do jakiego zasobu się odwołuje. Polega to głównie na odpowiedniej budowie adresów.
- **System warstwowy** – system powinien wprowadzać separację warstw.
- **Kod na żądanie** – zasada opcjonalna polegająca na udostępnianiu klientowi appletów i skryptów.

Główną kwestią, na którą kładzie nacisk *REST* są zasoby. Każda informacja, która może być nazwana, może być zasobem. Zasób jest to obiekt z typem, danymi, relacjami do innych zasobów oraz



Rys. 3.4. Główne metody wykorzystywane w REST API i geneza terminu *CRUD* [12]

z zestawem metod pozwalających na nim operować. Można go przyrównać do obiektu w programowaniu obiektowym. Zasoby mogą być oczywiście grupowane w kolekcję. Stan danego zasobu w dowolnym czasie jest znany jako reprezentacja zasobu. Zawiera ona dane, metadane opisujące zasób oraz linki typu hypermedia, które pomagają w podejmowaniu kolejnych akcji. Jeśli chodzi o same dane, które zawiera reprezentacja, przeważnie reprezentowane są one w formacie *JSON*⁵ (jednak występuje tutaj pełna dobrotność, nic nie stoi na przeszkodzie, aby użyć formatu *XML*).

Mówiąc o metodach, które zasób udostępnia, należy wrócić do wcześniej opisanego protokołu *HTTP*, ponieważ to właśnie tego protokołu używa *REST* do komunikacji. Konsumenti *API* uzyskują dostęp do zasobów poprzez wykorzystanie odpowiedniego internetowego adresu *URI* (*Uniform Resource Identifier*⁶) oraz odpowiedniej metody *HTTP*. Innymi słowy komunikacja opiera się na wysyłaniu zapytania *HTTP* z określona metodą, pod odpowiedni adres i otrzymaniu odpowiedzi *HTTP* z określonym rezultatem i towarzyszącą mu reprezentacją zasobu. Głównie wykorzystywane są cztery metody, przez co wiele prostych aplikacji będących *RESTful* jest nazywanych aplikacjami *CRUD* – rysunek 3.4 tłumaczy znaczenie tego terminu i przedstawia wspomniane metody.

3.2.2.3. SPA

SPA (*Single Page Application*) to podejście do tworzenia klienckiego interfejsu użytkownika w przeglądarce, a dokładniej wcześniej wspomnianego już front-endu, bardzo mocno zyskujące na popularności w ostatnich latach, opierające się na dynamicznym wyświetlanie danych i przebudowie tylko części widoku. W podejściu klasycznym czyli *MPA* (*Multipage Application*) za każdym razem, kiedy użytkownik nawigował się po stronie internetowej, przeglądarka wysyłała żądanie do serwera i w odpowiedzi otrzymywała nową stronę *HTML*⁷, którą następnie wyświetlała. *SPA* inaczej podchodzi do tematu nawigacji, zamiast zawsze prosić o nową stronę *HTML*, cały front-end jest ładowany do przeglądarki na samym początku uruchomienia strony, a wszelkie zapytania do serwera skutkują jedynie zwróceniem konkretnych danych (najczęściej w formacie *JSON*). Dzięki temu, to po stronie klienckiej leży odpowiedzialność za

⁵Więcej na temat *JSON*: <https://www.json.org/json-en.html>.

⁶Więcej na temat *URI*: <https://www.w3.org/Addressing/URL/uri-spec.html>.

⁷Więcej na temat *HTML*: <https://developer.mozilla.org/pl/docs/Web/HTML>.

odpowiednie wyświetlanie danych. Innymi słowy nie następuje przekierowanie na kolejną stronę, wyświetlana jest cały czas tylko jedna (co zresztą sugeruje nazwa *SPA*), ale to co na niej się znajduje jest dynamicznie zmieniane.

Warstwa prezentacji w opisywanej aplikacji została stworzona właśnie w podejściu *SPA*. Front-end komunikuje się z serwerem za pomocą *REST API*, otrzymując w odpowiedzi dane w formacie *JSON*, a następnie odpowiednio je przetwarza. Korzyści jakie przynosi ze sobą takie podejście są następujące:

- **Wydajność** – aplikacja jest szybsza, ponieważ o wiele mniej danych zostaje przetransportowane przez sieć internetową.
- **Mniejsze obciążenie serwera** – serwer może skupić się tylko i wyłącznie na przetwarzaniu danych, a nie na konstrukcji widoku.
- **Większe możliwości prezentacji treści** – przez brak ciągłego odświeżania strony, umozliwione są efekty, które wcześniej zarezerwowane były jedynie dla aplikacji natywnych dla urządzeń mobilnych lub desktopowych.
- **Zwiększenie odczuwalnej szybkości działania aplikacji** – aplikacja sprawia wrażenie szybszej, ponieważ niepotrzebne są całościowe przeładowania strony.

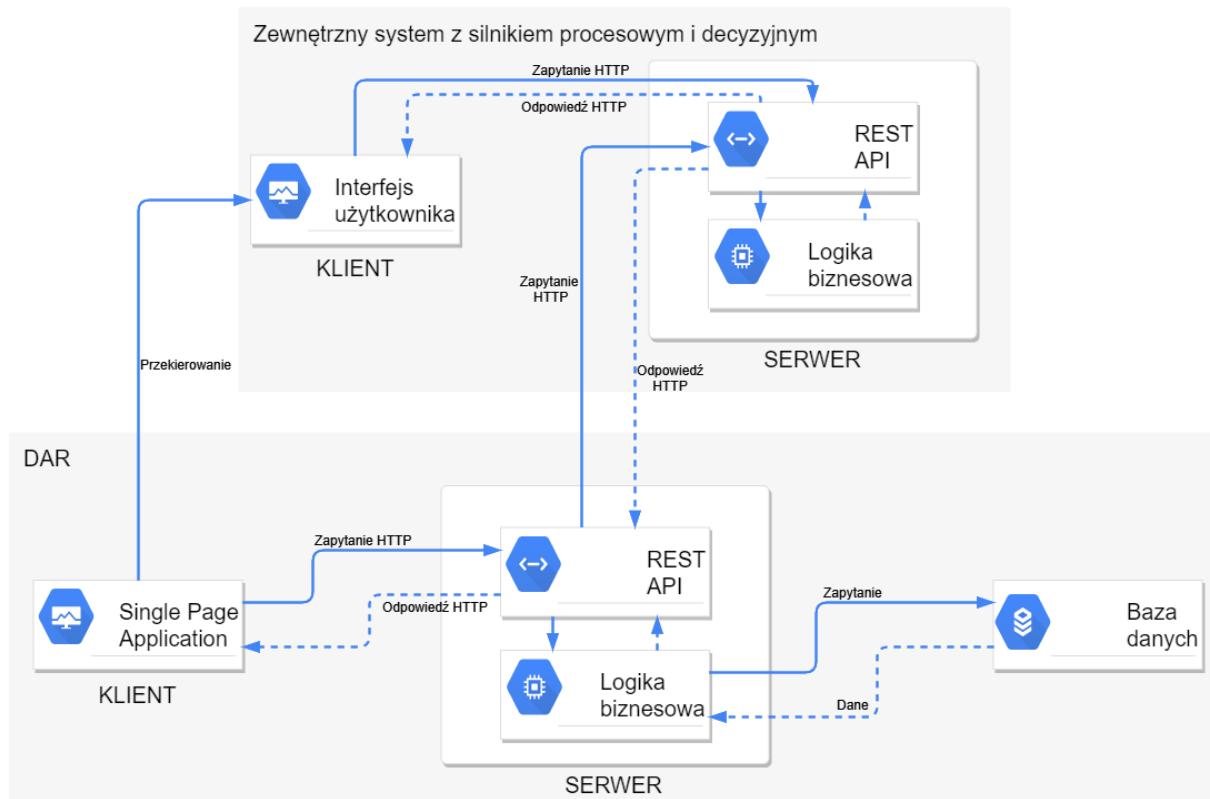
3.3. Schemat projektu

Rysunek 3.5 przedstawia generalny schemat projektu. Jak widać w projekcie dochodzi do integracji dwóch systemów. „DAR” udostępnia aplikację *SPA* po stronie klienckiej, która komunikuje się z serwisem za pomocą *REST API*, wysyłając odpowiednie zapytanie *HTTP*. Serwis po otrzymaniu żądania wykonuje zaimplementowaną logikę biznesową, korzystając przy tym z bazy danych, aby finalnie zwrócić odpowiedź. Ma on również możliwość komunikacji z serwisem zewnętrznego systemu, który również udostępnia *REST API*. Front-end aplikacji „DAR” ma możliwość przekierowania użytkownika do interfejsu wspomnianego zewnętrznego systemu.

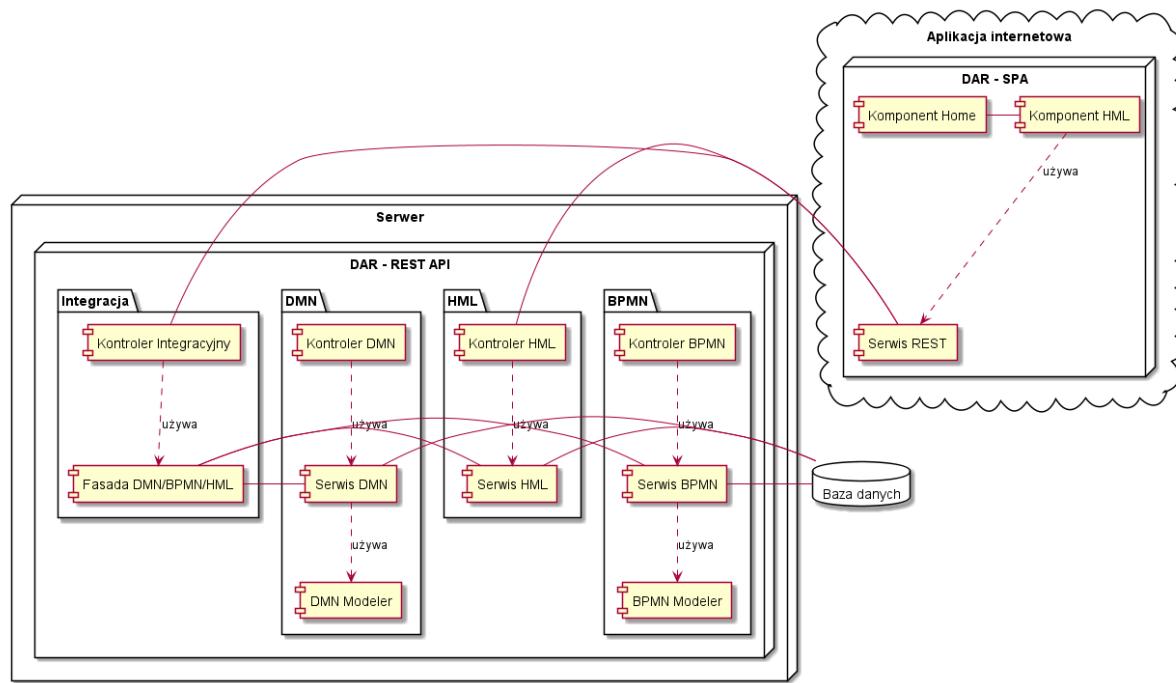
Podsumowując schemat projektu, występują w nim dwa zintegrowane ze sobą serwisy, dwa interfejsy użytkownika oraz baza danych. Sercem całego projektu jest back-end aplikacji, ponieważ to on integruje ze sobą resztę modułów i to on odpowiedzialny jest za pracę związaną z generacją modeli procesów i wdrażaniem ich do zewnętrznego systemu.

3.4. Komponenty

Nawiązując do rysunku 3.5, warto przyjrzeć się bliżej części „DAR”. Rysunek 3.6 prezentuje wyodrębnione komponenty aplikacji. Należy jednak zaznaczyć, że jest to zbliżenie tylko na wspomnianą wcześniej część, nie pojawiają się tam elementy zewnętrznego systemu, z którymi wiele z tych komponentów jest zintegrowanych.



Rys. 3.5. Schemat projektu „DAR”



Rys. 3.6. Komponenty aplikacji „DAR”

Podział komponentów na załączonym rysunku jest następujący:

– **Serwer**

– **HML**

- * **Kontroler HML** – część odpowiedzialna za komunikację *HTTP* i określająca adresy funkcjonalności związanych z plikami *HML*.
- * **Serwis HML** – część odpowiedzialna za logikę biznesową związaną z plikami *HML*. Zapewnia komunikację z bazą danych.

– **BPMN**

- * **Kontroler BPMN** – część odpowiedzialna za komunikację *HTTP* i określająca adresy funkcjonalności związanych z modelami *BPMN*.
- * **Serwis BPMN** – część odpowiedzialna za logikę biznesową związaną z modelami *BPMN*. Zapewnia komunikację z bazą danych.
- * **Modeler BPMN** – część odpowiedzialna za tworzenie modeli *BPMN*.

– **DMN**

- * **Kontroler DMN** – część odpowiedzialna za komunikację *HTTP* i określająca adresy funkcjonalności związanych z modelami *DMN*.
- * **Serwis DMN** – część odpowiedzialna za logikę biznesową związaną z modelami *DMN*. Zapewnia komunikację z bazą danych.
- * **Modeler DMN** – część odpowiedzialna za tworzenie modeli *DMN*.

– **Integracja**

- * **Kontroler integracyjny** – nazwany integracyjnym, ponieważ jego głównym zadaniem jest integracja z zewnętrznym systemem posiadającym silniki decyzyjne oraz procesowe i komunikację *HTTP*, określając adresy funkcjonalności związanych z tym systemem.
- * **Fasada DMN/BPMN/HML** – fasada wyżej opisanych serwisów, implementująca logikę biznesową związaną z wdrożeniem modeli procesów do zewnętrznego systemu.

– **Aplikacja internetowa**

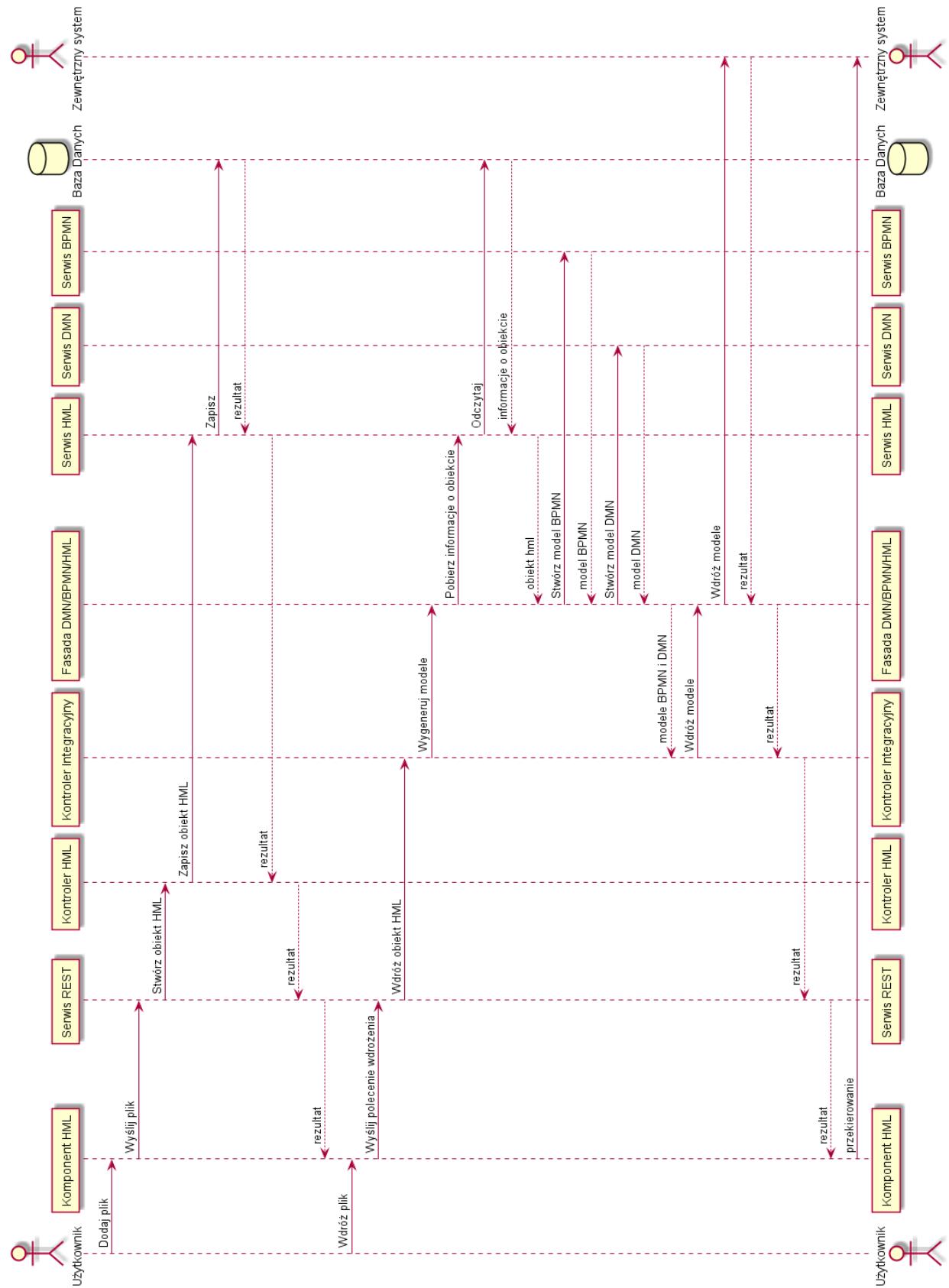
- **Komponent Home** – część domyślna aplikacji klienckiej, pozwala na nawigację do kolejnych elementów.
- **Komponent HML** – serce części klienckiej, gdzie znajduje się główny widok i to on odpowiada za umożliwienie użytkownikowi podstawowych interakcji typu dodanie i wdrożenie pliku *HML*.
- **Serwis REST** – serwis do komunikacji za pomocą protokołu *HTTP*, wykorzystywany przez *Komponent HML* do wysyłania zapytań *HTTP*.
- **Baza danych** – część odpowiedzialna za przechowywanie danych.

3.5. Schemat interakcji

Rysunek 3.7 prezentuje główny schemat interakcji pomiędzy komponentami na przykładzie procesu wgrywania pliku *HML* przez użytkownika oraz wdrażania tego pliku do zewnętrznego systemu posiadającego silniki decyzyjne oraz procesowe.

Zaczynając użytkownik dodaje pewien plik, w założeniu powinien to być plik *HML*. Następnie *Komponent HML* przesyła ten plik poprzez *Serwis REST*. Aplikacja po stronie serwera otrzymuje plik w *Kontrolerze HML*. Następnie używając *Serwisu HML* tworzy odpowiedni obiekt i zapisuje go do bazy danych. W przypadku wdrożenia sytuacja początkowa jest podobna – *Komponent HML* przesyła za pomocą *Serwisu REST* identyfikator odpowiedniego pliku *HML* wraz z poleceniem wdrożenia. Polecenie trafia do *Kontrolera integracyjnego*, który przekazuje identyfikator do fasady serwisów, która używa *Serwisu HML*. *Serwis HML* pobiera informację o danym pliku identyfikując go dostarczonym identyfikatorem i zwraca obiekt *HML*. Fasada następnie na podstawie otrzymanego rezultatu używa *Serwisu BPMN* oraz *Serwisu DMN*. Te zwracają odpowiednie modele *BPMN* oraz *DMN*. Fasada po odpowiedzi od serwisów zwraca modele do kontrolera, który komunikuje się z zewnętrznym systemem wysyłając żądanie wdrożenia otrzymanych z fasady obiektów. Finalnie komponent po stronie klienckiej, po otrzymaniu pozytywnego rezultatu, przekierowuje użytkownika do zewnętrznego systemu.

Jest to koniec rozdziału opisującego projekt aplikacji. W tym rozdziale została przedstawiona architektura systemu będącego głównym tematem niniejszej pracy. W kolejnym rozdziale zostanie opisana implementacja przedstawionego tutaj projektu, jakie technologie zostały użyte oraz w jaki sposób zaimplementowane zostały najważniejsze, wspomniane tutaj komponenty aplikacji.



Rys. 3.7. Schemat interakcji w aplikacji „DAR”

4. Implementacja

Rozdział opisuje technologie użyte w pracy oraz przedstawia strukturę bazy danych i implementacje wydzielonych komponentów. Aplikacja prezentowana jest od strony implementacyjnej.

4.1. Stos technologiczny

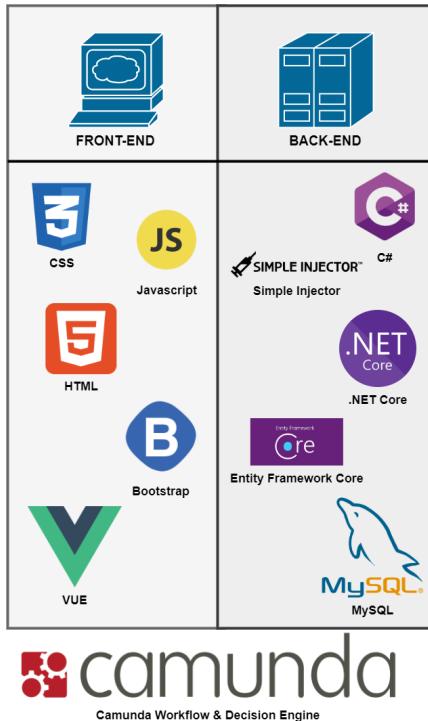
Rysunek 4.1 prezentuje technologie użyte podczas implementacji projektu. Stos przedstawiony na rysunku został podzielony na część związaną z interfejsem użytkownika oraz na część związaną z serwerem. Dodatkowo pokazana jest technologia, która została wybrana jako zewnętrzny system i zintegrowana z aplikacją „DAR”. Kolejne podrozdziały pogłębiają temat wybranych technologii.

4.1.1. Technologie front-end

Przedstawione technologie po stronie lewej rysunku 4.1 są związane z interfejsem użytkownika. Występuje tutaj podział na języki programowania oraz szkielety aplikacyjne/biblioteki:

- **Języki programowania/Języki komputerowe**
 - **Javascript** – interpretowany, dynamiczny język programowania wykorzystywany jako główny język skryptowy na stronach internetowych. Według ankiety [13] z 2019 roku, przeprowadzonej na serwisie *StackOverflow*¹ jest to obecnie najpopularniejszy język świata, bezsprzecznie królujący w świecie przeglądarek internetowych.
 - **HTML** – *HyperText Markup Language* to język komputerowy stworzony w celu tworzenia stron internetowych. Umożliwia tworzenie nagłówków, połączeń, ustrukturyzowanych sekcji, paragrafów i wiele innych elementów...
 - **CSS** – *Cascading Style Sheets* to język związany ze stylem dokumentów. Umożliwia on specyfikację, jak pliki są prezentowane użytkownikom – określa ich styl, ułożenie elementów oraz inne atrybuty. Najczęściej używany razem z językiem *HTML*.

¹Jeden z najpopularniejszych serwisów dla programistów: <https://stackoverflow.com/>.



Rys. 4.1. Stos technologiczny aplikacji „DAR”

– Szkielety aplikacyjne/Biblioteki

- **Bootstrap** – otwarte oprogramowanie, będące biblioteką CSS, nad którą pieczę sprawują programiści *Twittera*². Bootstrap usprawnia tworzenie aplikacji internetowych, udostępniając wiele szablonów HTML oraz CSS, które upiększają stronę.
- **Vue** – Szkielet aplikacyjny front-end napisany w JavaScript, do tworzenia wszelkiego rodzaju interfejsów użytkownika w tym *Single Page Application*. Usprawnia tworzenie aplikacji internetowych poprzez udostępnienie wielu funkcjonalności, w tym chociażby manipulowanie drzewem obiektów dokumentu, co pozwala na dynamiczne podmienianie danych na stronie bez konieczności pełnego przeładowania strony. Jest on bardzo podobny do najpopularniejszych szkieletów aplikacyjnych, takich jak *React*³ lub *Angular*⁴. Według wspomnianej już ankiety [13], Vue zyskuje bardzo szybko na popularności.

4.1.2. Technologie back-end

Przedstawione technologie po stronie prawej rysunku 4.1 są związane z warstwą logiki oraz danych. Występuje tutaj podział na języki programowania, szkielety aplikacyjne/biblioteki oraz oprogramowania do zarządzania bazą danych:

²Jeden z najpopularniejszych portali społecznościowych: <https://twitter.com/>.

³Więcej na temat React: <https://pl.reactjs.org/>.

⁴Więcej na temat Angular: <https://angular.io/>.

– Języki programowania

– **C#** – wysokopoziomowy, obiektowy język programowania. Był on odpowiedzią *Microsoftu*⁵ na język *Java*⁶, ściśle zintegrowany z platformą *Microsoftu .NET*. Wcześniej używany głównie do tworzenia aplikacji na systemy *Windows*, ale odkąd powstał wieloplatformowy *.NET Core* sytuacja uległa zmianie. Często używany do tworzenia internetowych *API* oraz aplikacji mobilnych dzięki technologii *Xamarin*. Przykładem aplikacji stworzonej przy użyciu C# jest portal *StackOverflow*.

– Szkielety aplikacyjne/Biblioteki

– **.NET Core** – a właściwie *ASP .NET Core*, jest to otwarte oprogramowanie, będące wieloplatformowym szkieletem aplikacyjnym rozwijanym przez *Microsoft* do tworzenia wysoko wydajnych, nowoczesnych, opartych na technologii chmurowej, internetowo połączonych aplikacji. Cały kod tego projektu dostępny jest na platformie *GitHub*⁷. Dodatkowo dzięki wieloplatformowości umożliwia on tworzenie oraz działanie na platformach, takich jak *Linux* czy *macOS*, a oprócz tego jest on stworzony z myślą o konteneryzacji. *ASP .NET Core* wspiera najlepsze metodyki tworzenia aplikacji poprzez promowanie modularności, wbudowane mechanizmy wstrzykiwania zależności czy wsparcie dla rozwiązań chmurowych oraz *gRPC*⁸. Szkielet aplikacyjny *.NET Core* w cytowanej już ankcie [13] portalu *StackOverflow* z roku 2019, zyskał pierwsze miejsce w rankingu najlepszych szkieletów aplikacyjnych. Natomiast pod względem wydajności plasuje się w czołówce (miejsce 10) według rankingu *TechEmpower* [14], wyprzedzając takie szkielety aplikacyjne jak *Django* (miejsce 213), *Spring* (miejsce 245), czy *Ruby on Rails* (miejsce 215).

– **Entity Framework Core** – jest to ORM (*Object-Relational Mapper*), czyli narzędzie pozwalające programistom działać z danymi w stylu obiektowym. Jego głównym zadaniem jest mapowanie obiektów zdefiniowanych w aplikacji w wybranym języku programowania, do danych przechowywanych w relacyjnej bazie danych. Narzędzie to bardzo ułatwia pracę z bazą danych, ukrywając za abstrakcjami połączenia z bazą danych, wszelkie konwersje do języka *SQL* oraz tworzenie struktur bazy danych.

– **Simple Injector** – biblioteka służąca jako kontener wstrzykiwania zależności. Szkielet aplikacyjny *.NET Core* posiada wbudowane narzędzie do wstrzykiwania zależności jednak *Simple Injector* działa opierając się na nim i rozbudowując jego możliwości. Jego głównym zadaniem jest pilnowanie oraz zarządzanie cyklem życia obiektów w aplikacji, poprzez zastosowanie odwrócenia sterowania [15].

⁵Jedna z największych korporacji świata: <https://www.microsoft.com>.

⁶Jeden z najbardziej popularnych, obiektowych języków programowania na świecie: <https://www.java.com>.

⁷Zobacz: <https://github.com/aspnet/AspNetCore>.

⁸Więcej na temat gRPC: <https://grpc.io/>.

- **Oprogramowanie do zarządzania bazą danych**
 - **MySQL** – jest to otwarte oprogramowanie, będące systemem do zarządzania relacyjnymi bazami danych, opierającym się na modelu klient-serwer. Odpowiada on za stworzenie bazy danych i odpowiednie przechowywanie oraz zarządzanie informacjami. Komunikacja opiera się na języku SQL (*Structured Query Language*), służącym do manipulowania danymi.

4.1.3. Zewnętrzny System

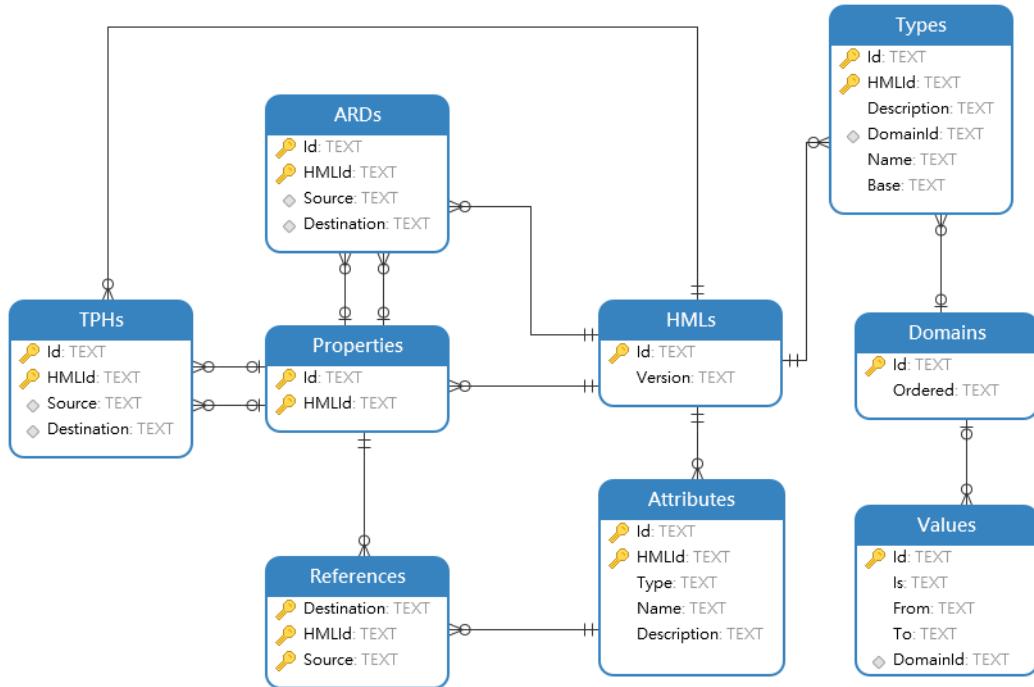
Camunda została wybrana jako zewnętrzny system do zarządzania modelami procesów oraz decyzji. Jest to otwarte oprogramowanie napisane w języku *Java*, pozwalające na uruchomienie wykonywalnych modeli procesów, ewaluację decyzji oraz monitorowanie całego przepływu działań procesu. Najważniejsze komponenty, które udostępnia *Camunda* w kontekście niniejszej pracy to:

- **Silnik**
 - **Process Engine** – silnik napisany w języku *Java*, umożliwiający uruchamianie procesów w notacji *BPMN* oraz decyzji w notacji *DMN*.
- **Aplikacje internetowe**
 - **REST API** – *API* umożliwiające dostęp do wspomnianego wcześniej silnika oraz platformy zarządzającej procesami i decyzjami. Głównie wykorzystywane w celu wdrożenia modelów *BPMN* oraz *DMN* do platformy.
 - **Camunda Tasklist** – część platformy do zarządzania procesami i decyzjami. Umożliwia uruchomienie procesu oraz wykonywanie zadań użytkownika, np. podanie odpowiednich danych, potrzebnych do działania procesu.
 - **Camunda Cockpit** – część platformy do zarządzania procesami i decyzjami. Umożliwia obserwację i edycję modeli *BPMN* oraz *DMN*, oraz monitorowanie wykonywanych procesów i ewaluowanych decyzji.

Pomimo że *Camunda* jest w wersji podstawowej otwartym oprogramowaniem, w kontekście niniejszej pracy wymagana jest edycja typu *Enterprise* będąca wersją płatną. Wynika to z faktu, że modele *DMN* tworzone przez aplikację *DAR* nie są wypełnione regułami – ze względu na brak takowych w diagramach *ARD*. W utworzonych tabelach decyzyjnych należy uzupełnić reguły, tak aby decyzje mogły być ewaluowane. Ta możliwość jest udostępniona tylko w wersji płatnej.

4.2. Schemat bazy danych

Dzięki wspomnianemu wcześniej *Entity Framework Core*, wszystkie klasy, które zostały zmapowane ze schematu pliku *HML* do kodu w języku *C#*, automatycznie zostały użyte do budowy tabel w relacyjnej bazie danych. Rysunek 4.2 przedstawia efekt tego zabiegu.



Rys. 4.2. Schemat bazy danych aplikacji „DAR”

Warto przyjrzeć się bliżej poszczególnym tabelom:

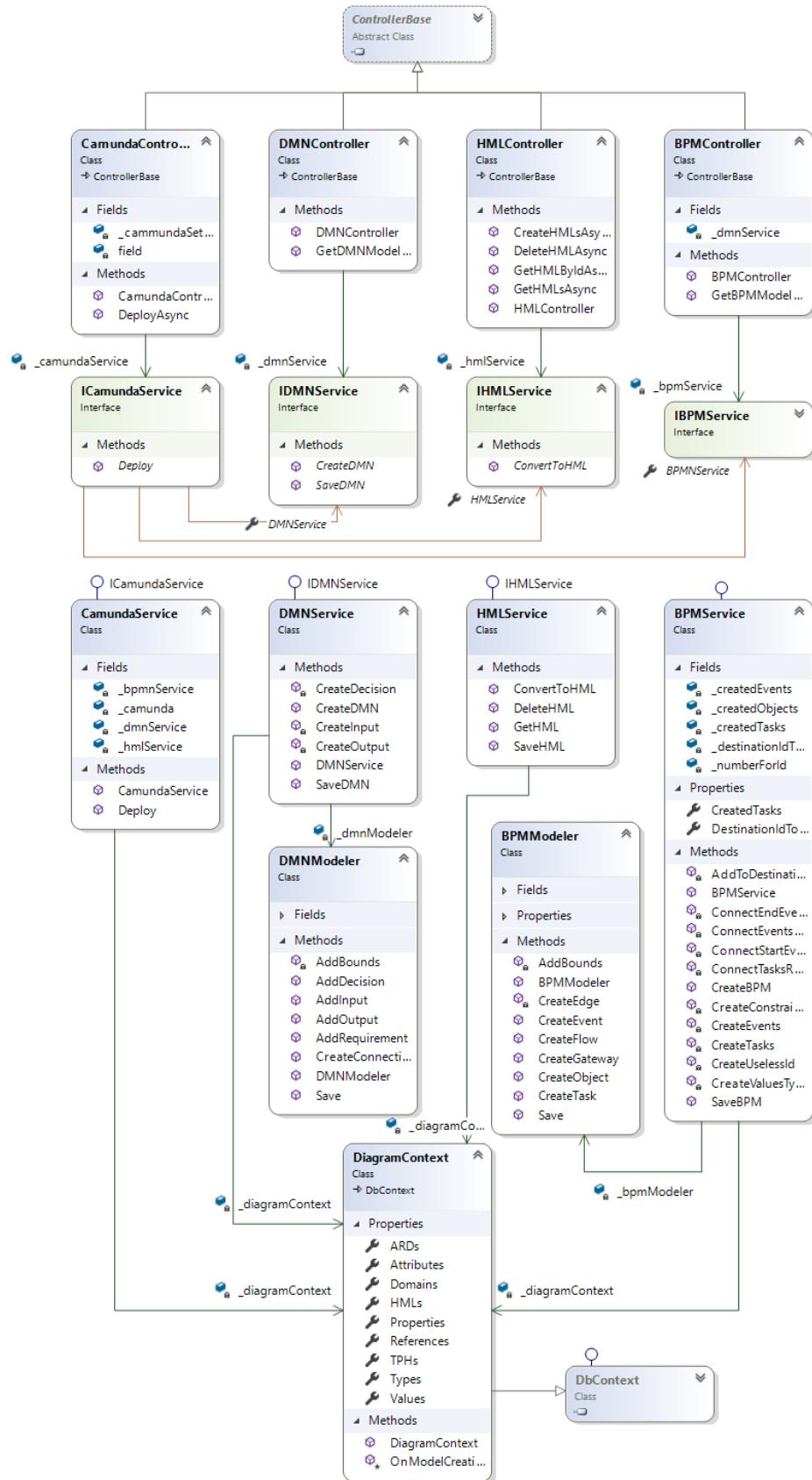
- **Tabela HMLs** – serce całego schematu. Posiada jeden klucz główny *Id*, identyfikujący poszczególne pliki *HML*. Dodatkowo określa wersję danego pliku.
- **Tabela Properties** – reprezentuje właściwości opisane w plikach. Posiada dwa klucze, jeden główny, będący złożeniem *Id* oraz *HMLId*, służący do identyfikacji, oraz drugi obcy *HMLId*, będący połączeniem do tabeli *HMLs*.
- **Tabela Attributes** – reprezentuje atrybuty opisane w plikach. Posiada dwa klucze, jeden główny, będący złożeniem *Id* oraz *HMLId*, służący do identyfikacji, oraz drugi obcy *HMLId*, będący połączeniem do tabeli *HMLs*. Zawiera dodatkowo informacje o atrybutach, takie jak nazwa, opis oraz typ danych.
- **Tabela References** – pośredniczy pomiędzy właściwościami, a atrybutami. Posiada trzy klucze. Jeden podstawowy będący złożeniem *Destination*, *Source* oraz *HMLId*, służący do identyfikacji oraz dwa obce: złożenie *Source* oraz *HMLId*, będące połączeniem do tabeli *Properties* i złożenie *Destination* oraz *HMLId*, będące połączeniem do tabeli *Attributes*.
- **Tabela Types** – reprezentuje typy danych opisane w plikach. Posiada trzy klucze. Jeden podstawowy *Id*, służący do identyfikacji oraz dwa obce: *HMLId* będący połączeniem do tabeli *HMLs* oraz *DomainId*, będący połączeniem do tabeli *Domains*. Dodatkowo określa takie właściwości, jak opis typu, nazwa oraz podstawa (przykładowo numeryczna).

- **Tabela Values** – reprezentuje możliwe wartości danego typu. Posiada dwa klucze. Jeden podstawowy *Id*, służący do identyfikacji oraz jeden obcy *DomainId*, będący połączeniem do tabeli *Domains*. Dodatkowo określa różne możliwości związane z wartościami, typu przedziału wartości, czy jest to wartość numeryczna lub konkretna wartość.
- **Tabela Domains** – reprezentuje dziedziny opisane w pliku i pośredniczy między tabelami *Types* and *Values*. Posiada jeden klucz główny *Id*, służący do identyfikacji.
- **Tabela ARDs** – reprezentuje zależności między właściwościami. Posiada cztery klucze: jeden podstawowy złożony z *Id* oraz *HMLId*, służący do identyfikacji i trzy obce: *HMLId*, będące połączeniem do tabeli *HMLs* oraz dwa złożenia: *HMLId* i *Source*, będące połączeniem do tabeli *Properties*, do właściwości niezależnej oraz złożenie *HMLId* i *Destination*, będące połączeniem do tabeli *Properties*, do właściwości zależnej.
- **Tabela TPHs** – właściwie niewykorzystywana, reprezentuje historię zależności między właściwościami. Posiada cztery klucze: jeden podstawowy złożony z *Id* oraz *HMLId*, służący do identyfikacji oraz trzy obce: złożenie *HMLId* i *Source*, będące połączeniem do tabeli *Properties*, do właściwości niezależnej, złożenie *HMLId* i *Destination*, będące połączeniem do tabeli *Properties*, do właściwości zależnej oraz *HMLId*, będące połączeniem do tabeli *HMLs*.

4.3. Schemat klas

Rysunek 4.3 przedstawia najistotniejsze klasy aplikacji. Pominięte zostały klasy związane z infrastrukturą, określające jak szkielet aplikacyjny *ASP .NET Core* ma obsługiwać przychodzące zapytania *HTTP*, którego serwera lub której bazy danych ma używać. Dodatkowo niezaprezentowane zostały enkleje, będące prostymi klasami bez żadnej logiki, zawierającymi jedynie pola, gdyż zostały wygenerowane one automatycznie na podstawie specyfikacji plików *ARD*, *BPMN* oraz *DMN*.

Na rysunku 4.3 widać, że wszystkie komponenty typu *Kontroler*, dziedziczą po bazowej klasie *ControllerBase*, należącej do *ASP .NET Core*. Dzięki temu, aplikacja może w łatwy sposób nawigować adresy w stylu *REST* do konkretnej klasy i metody. Kontrolery przyjmują zapytania *HTTP* oraz związane z nimi dane, walidują je, a następnie przekazują do komponentów implementujących logikę biznesową, w tym przypadku serwisów. Serwisy występujące w kontrolerach, opierają się na interfejsach – jest to związane z wspomnianym już wstrzykiwaniem zależności i odwróceniem sterowania. *ASP .NET Core* promuje programowanie do interfejsów. Na diagramie 4.3 widać, że każdy interfejs został zaimplementowany i to właśnie te implementacje są wstrzykiwane w czasie działania aplikacji do kontrolerów. Dodatkowo serwisy *BPM*, *DMN* oraz *HML* posiadają referencje do klasy *DiagramContext*, dziedziczącej po klasie *DbContext* – klasie należącej do opisywanego już *Entity Framework Core*. Umożliwia ona komunikację z bazą danych. Wszystkie pola występujące w klasie *DiagramContext* są odwzorowane w bazie danych jako tabele (rysunek 4.2) – serwisy korzystają z tej klasy, jak z repozytorium, używając



Rys. 4.3. Schemat klas wchodzących w skład REST API aplikacji „DAR”

wspomnianych pól aby, pobierać/edytować/usuwać/dodawać dane. Serwisy *BPMN* oraz *DMN* zawierają również referencję do odpowiednich klas zakończeniem „Modeler”. Są to klasy pomocnicze pozwalające na dodawanie diagramowych obiektów związanych z modelami *BPMN* oraz *DMN* do plików tych modeli – takie obiekty muszą posiadać wymiary oraz współrzędne.

Jest to koniec rozdziału opisującego implementacje aplikacji. W tym rozdziale przedstawione zostały technologie użyte w celu stworzenia systemu, a także schematy implementacji najważniejszych elementów aplikacji. W kolejnym rozdziale wykorzystane zostaną opisane tutaj elementy i przedstawione zostanie działanie zaimplementowanej aplikacji na konkretnym przykładzie.

5. Ewaluacja

Rozdział prezentuje działanie aplikacji na przykładzie procesu obliczania kwoty ubezpieczenia samochodowego. Przedstawione zostały wybrane najważniejsze funkcjonalności systemu.

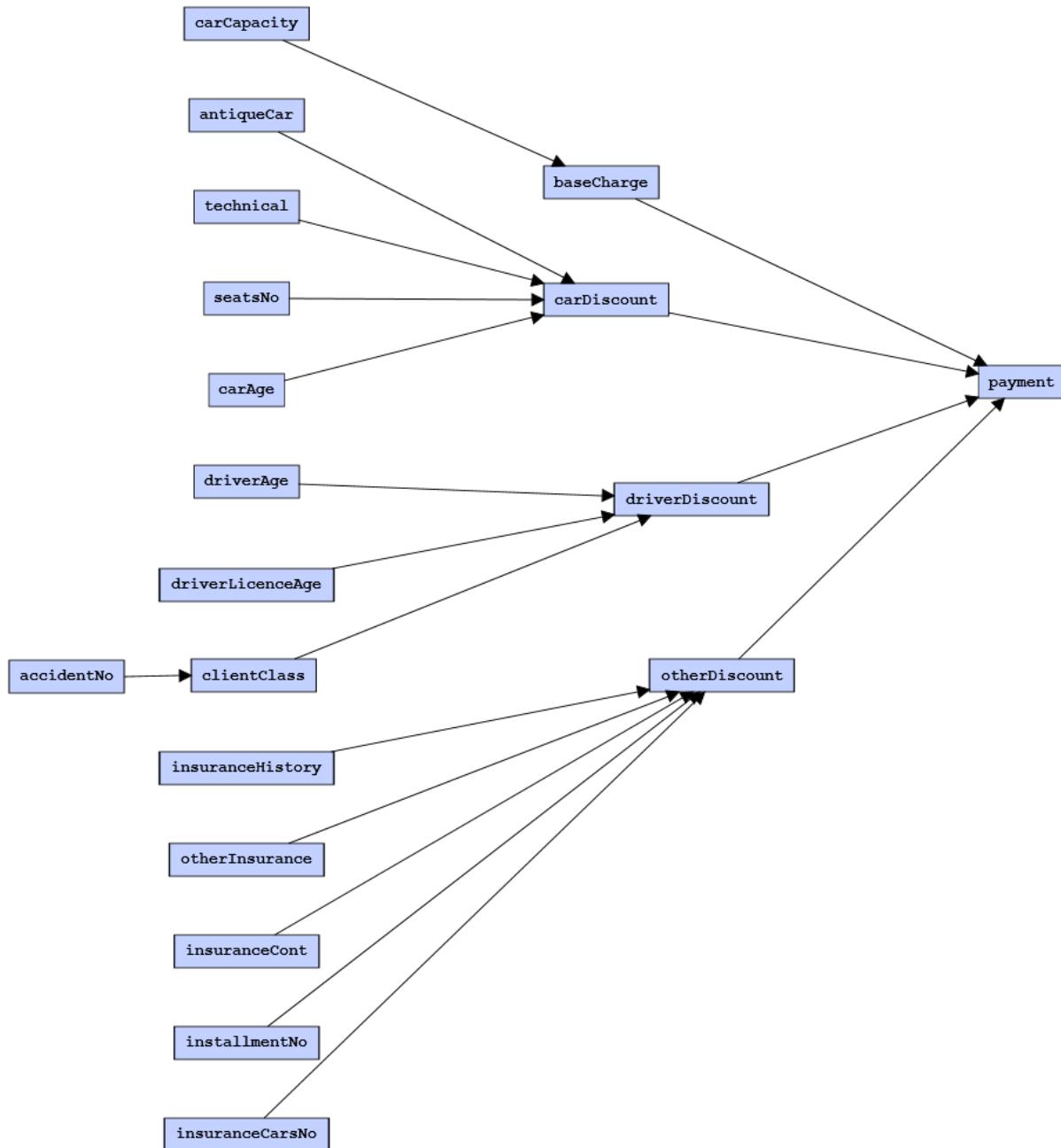
5.1. Model ARD

Przykład, na którym zostanie zaprezentowana aplikacja, obrazuje proces związany z obliczaniem kwoty płatności na rzecz ubezpieczenia samochodowego. Jak wiadomo kwota takiego ubezpieczenia zależy od wielu zmiennych, takich jak liczba wypadków, wiek kierowcy czy rodzaj samochodu. Rysunek 5.1 prezentuje diagram *ARD*, który mógłby zostać stworzony przez analityka biznesowego, jako prototyp takiego procesu. Tabela 5.1 opisuje wszystkie własności, przedstawione na diagramie *ARD* (rysunek 5.1). Podany jest ich typ, zakres oraz krótki opis. Na tej podstawie stworzony został plik *HML*. Zawartość pliku, ze względu na spory rozmiar, nie jest tutaj wylistowana.

5.2. Interfejs użytkownika

Po uruchomieniu aplikacji pierwszym widokiem, z którym ma do czynienia użytkownik jest domyślny ekran z omawianego w rozdziale 3 komponentu *Home*, co prezentuje rysunek 5.2.

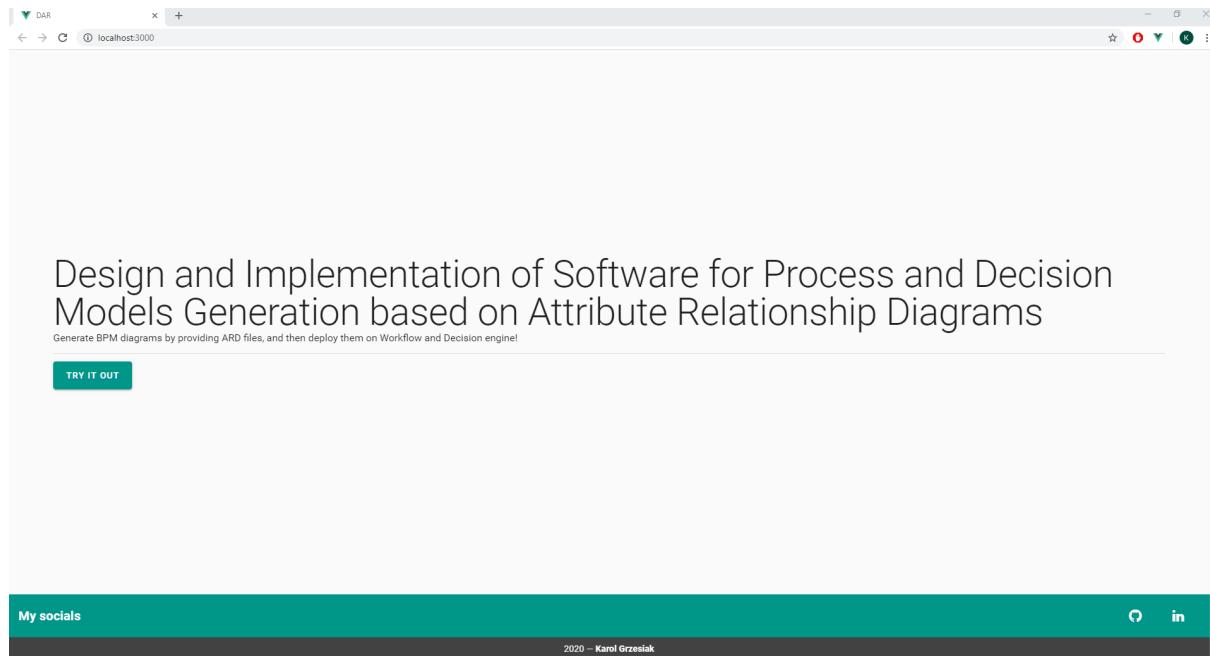
Po naciśnięciu przycisku „TRY IT OUT”, użytkownik zostaje przeniesiony do głównej części interfejsu użytkownika, czyli do widoku z komponentem *HML*. W tym miejscu prezentowane są wszystkie pliki *HML*, z których informacje znajdują się w bazie danych. Użytkownik ma możliwość dodania nowego pliku, a także wykonania akcji na plikach już dodanych. W przypadku naciśnięcia na któryś z wyświetlanych plików, aplikacja prezentuje dwa przyciski: „DEPLOY” oraz „DELETE”. Naciśnięcie pierwszego przycisku powoduje generację modeli *BPMN* oraz *DMN*, a następnie wdrożenie ich na platformę *Camunda*, aby finalnie przenieść użytkownika na wspomnianą platformę. Naciśnięcie drugiego przycisku powoduje usunięcie pliku z bazy danych. Rysunek 5.3 przedstawia opisany interfejs po wgraniu dwóch plików *HML*. Plik „example” to plik zawierający wcześniej opisywany model *ARD*. Dalsza część rozdziału opisuję scenariusz, w którym naciśnięty zostaje zielony przycisk pod plikiem „example” i plik zostaje wdrożony.



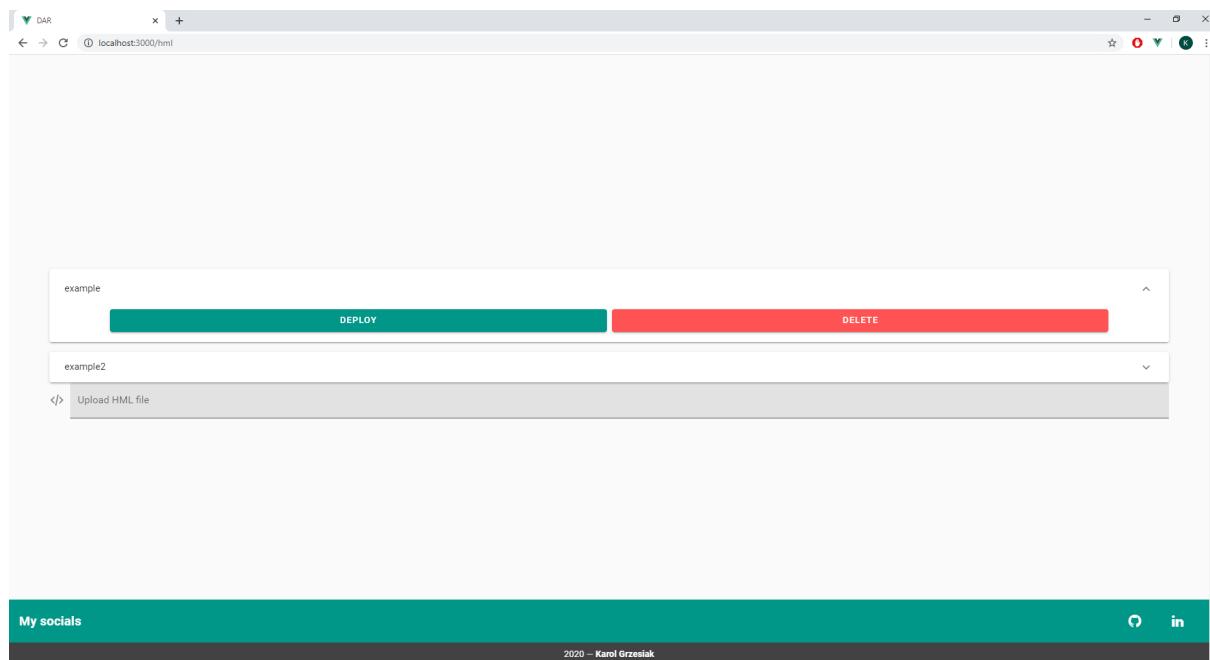
Rys. 5.1. Diagram ARD będący prototypem procesu obliczania kwoty ubezpieczenia samochodowego. Przykład opracowany na bazie studium przypadku przygotowanego w ramach ewaluacji metodyki Semantyczna Inżynieria Wiedzy [16]

nazwa	typ	zakres	opis
accidentNo	integer	[0;inf]	liczba kolizji spowodowanych w okresie ostatnich 12 miesięcy
clientClass	integer	[-1;9]	klasa jaką posiada klient
carCapacity	integer	[0;inf]	pojemność silnika [cm^3]
baseCharge	integer	[0;inf]	stawka podstawowa w [PLN]
driverAge	integer	[16;inf]	wiek właściciela pojazdu
driverLicenseAge	integer	[0;inf]	okres posiadania prawa jazdy przez właściciela pojazdu
driverDiscount	integer	—	suma zniżek driverAge i drLicAge
carAge	integer	[0;inf]	wiek samochodu
antiqueCar	boolean	[true; false]	pojazd historyczny
seatsNo	integer	[2;9]	liczba siedzeń w pojeździe
technical	boolean	[true; false]	aktualne badania techniczne
carDiscount	integer	—	suma zniżek za carAge, historiCar, noSeats i technical
installmentNo	integer	[1;2]	liczba rat
insuranceCont	boolean	[true; false]	kontynuacja ubezpieczenia
insuranceCarsNo	integer	[0;inf]	liczba ubezpieczonych samochodów
otherInsurance	boolean	[true; false]	inne ubezpieczenia
insuranceHistory	integer	[0;inf]	historia ubezpieczenia
otherDiscount	integer	—	suma zniżek za noRates, contIns, noCarsIns, otherIns, insHistory
payment	float	[0;inf]	ostateczna opłata za ubezpiecznie danego pojazdu w [PLN]

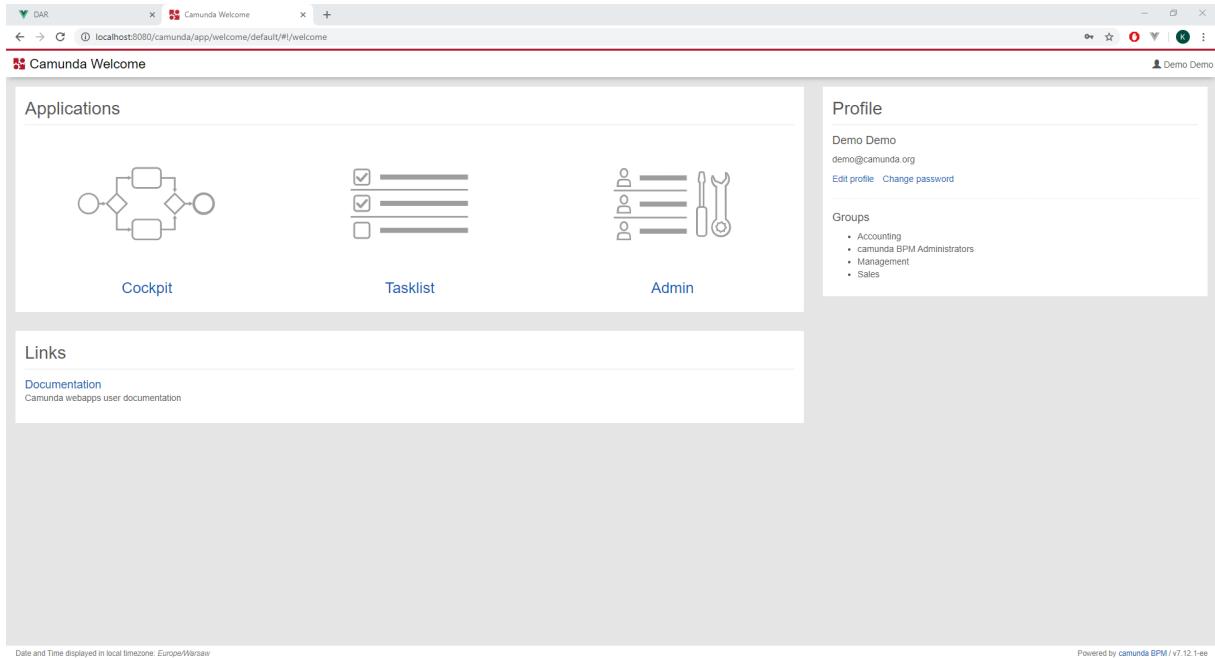
Tabela 5.1. Opis własności w prototypie procesu obliczania kwoty ubezpieczenia samochodowego. Przykład opracowany na bazie studium przypadku przygotowanego w ramach ewaluacji metodyki Semantycznia Inżynieria Wiedzy [16]



Rys. 5.2. Domyślny ekran aplikacji „DAR”



Rys. 5.3. Główny ekran interfejsu użytkownika aplikacji „DAR”



Rys. 5.4. Domyślny ekran platformy *Camunda* po wdrożeniu pliku *HTML*

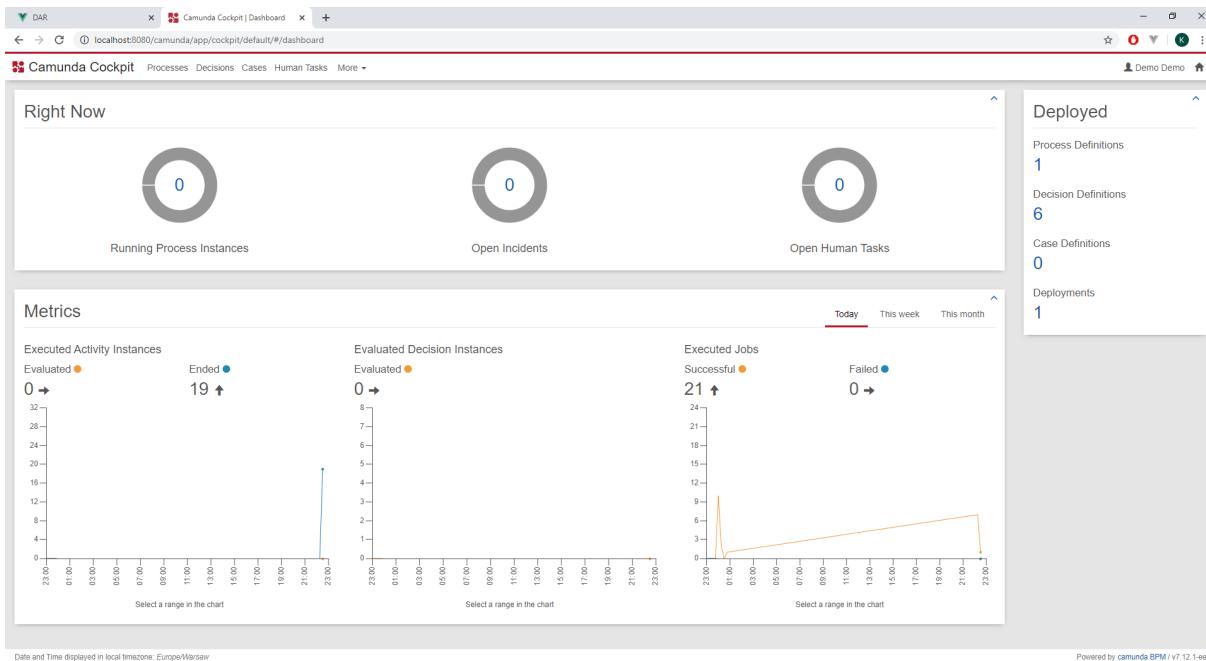
5.3. Camunda

Po wdrożeniu pliku zawierającego model opisany w podrozdziale 5.1 użytkownik zostaje przeniesiony na platformę *Camunda*. Domyślny widok, który zostaje mu zaprezentowany przedstawia rysunek 5.4. Do dyspozycji są trzy aplikacje, najważniejsze z nich to *Camunda Cockpit* oraz *Camunda Tasklist*. Były one opisywane w rozdziale 4.1.3.

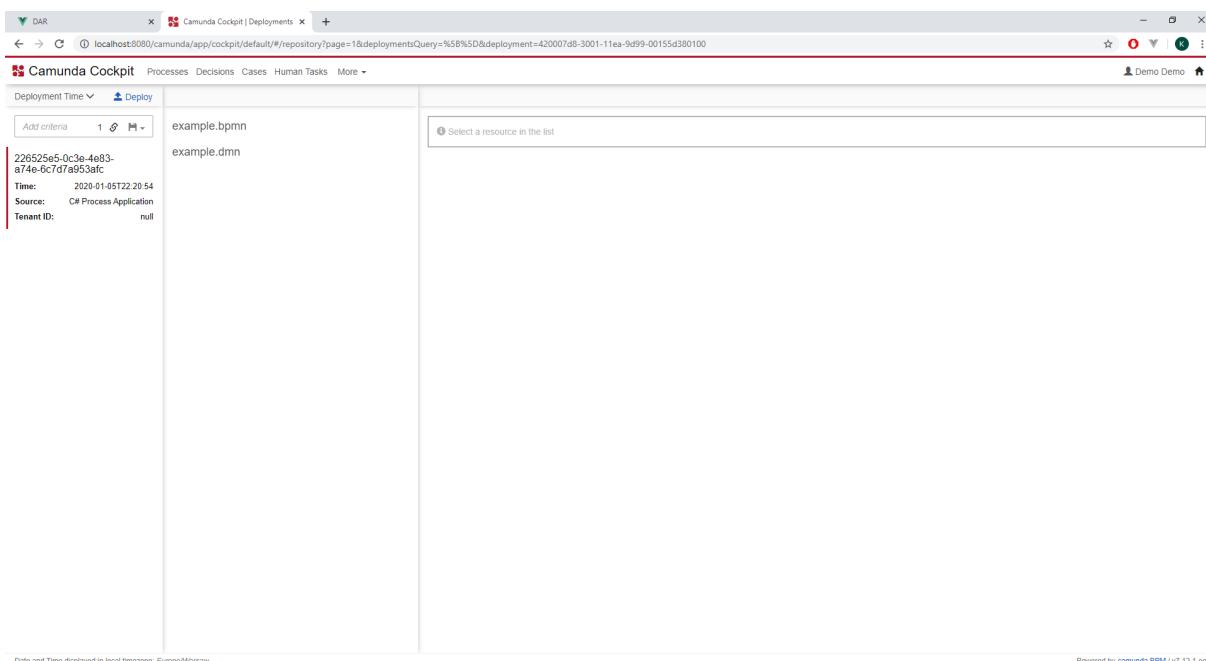
Rysunek 5.5 przedstawia domyślny ekran aplikacji *Camunda Cockpit*. Znajduje się na nim wiele metryk służących do analizy działań na platformie oraz informację o uruchomionych procesach. Przechodząc do zakładki „Deployments” („More“ → „Deployments“) prezentowane są wszystkie wdrożenia, które miały miejsce. Jak widać na rysunku 5.6 pojawiło się tutaj wdrożenie pliku „example”, a wraz z nim wygenerowane modele w notacjach *BPMN* oraz *DMN*. Rysunki 5.7 oraz 5.8 prezentują stworzone wspomniane modele.

Jak już zostało w pracy nadmienione diagramy *ARD* dostarczają informacje pozwalające na stworzenie modeli *DMN* oraz związanych z nimi tablic decyzyjnych, w których określona jest ilość atrybutów wejściowych i wyjściowych, ich nazwy oraz typy wartości. Reguły biznesowe jednak pozostają puste. Dlatego po wdrożeniu użytkownik musi sam edytować tablice decyzyjne i według własnego uznania wypełnić reguły. Rysunek 5.9 prezentuje edycję jednej z tablic decyzyjnych związanych z jedną z decyzji z rysunku 5.8. Na potrzeby przykładu dodane zostały dwa wiersze reguł. Każda z tablic decyzyjnych została w podobny sposób wypełniona, aby móc uruchomić proces.

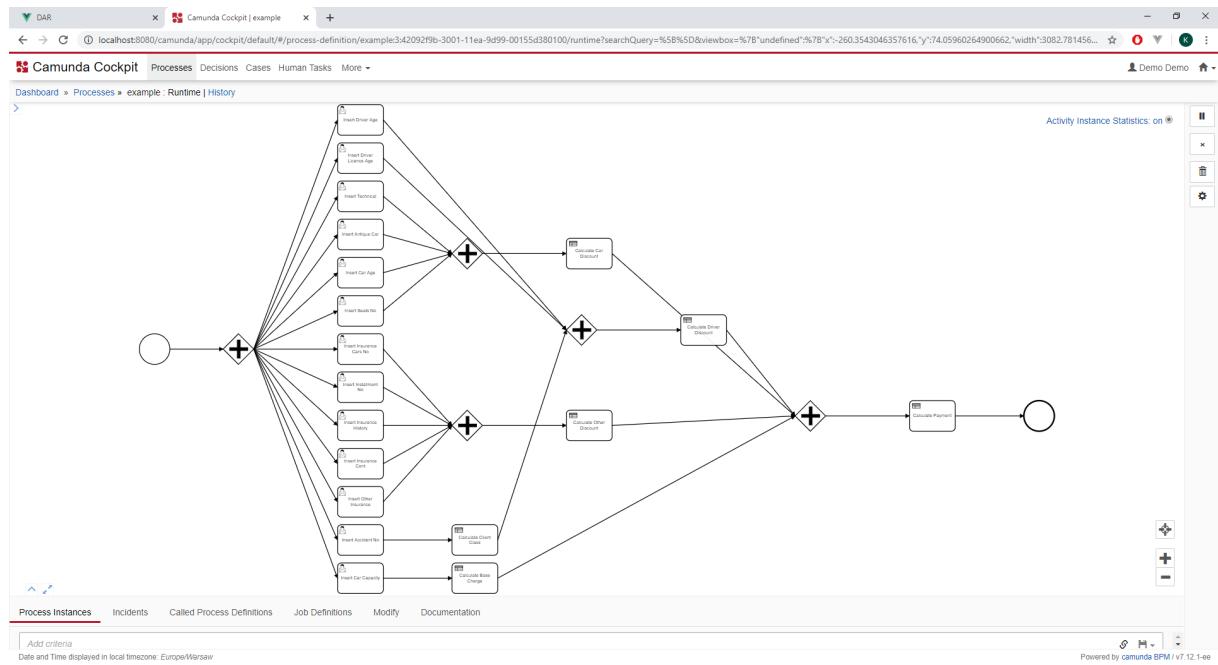
W tym momencie nic nie stoi na przeszkodzie, aby uruchomić proces. Żeby tego dokonać trzeba przejść do *Camunda Tasklist* (używając menu, w prawym górnym rogu interfejsu w kształcie domku). Widok z jakim spotyka się użytkownik jest przedstawiony na rysunku 5.10. Tutaj wyświetlane będą



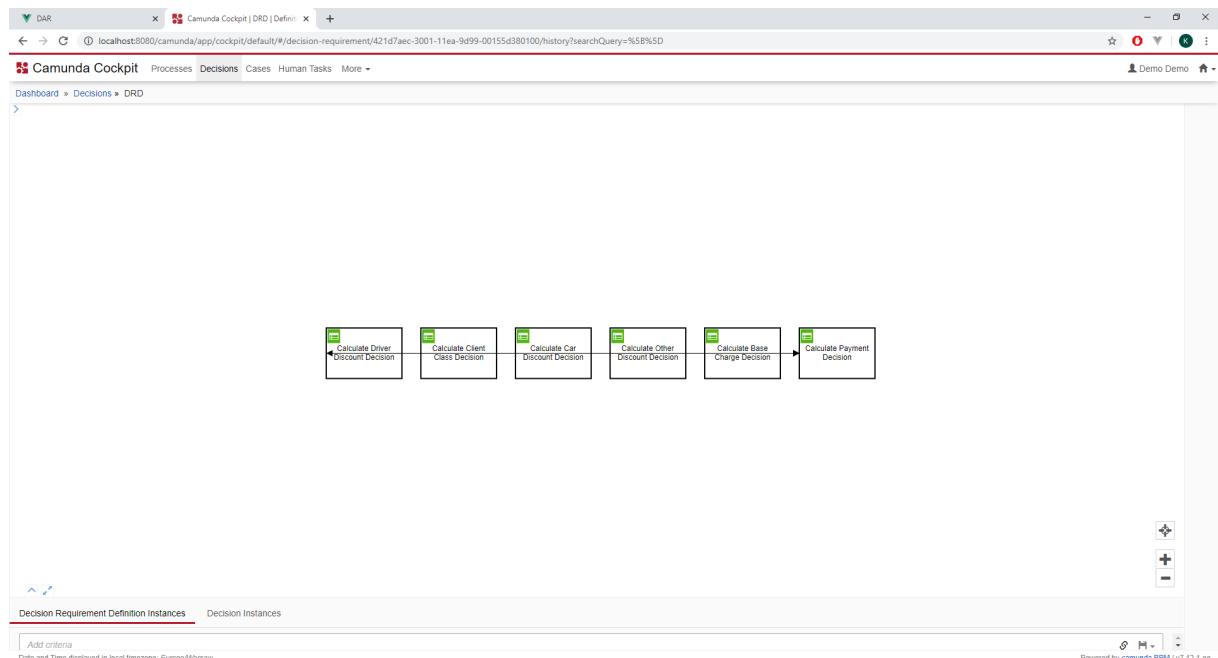
Rys. 5.5. Domyślny ekran Camunda Cockpit



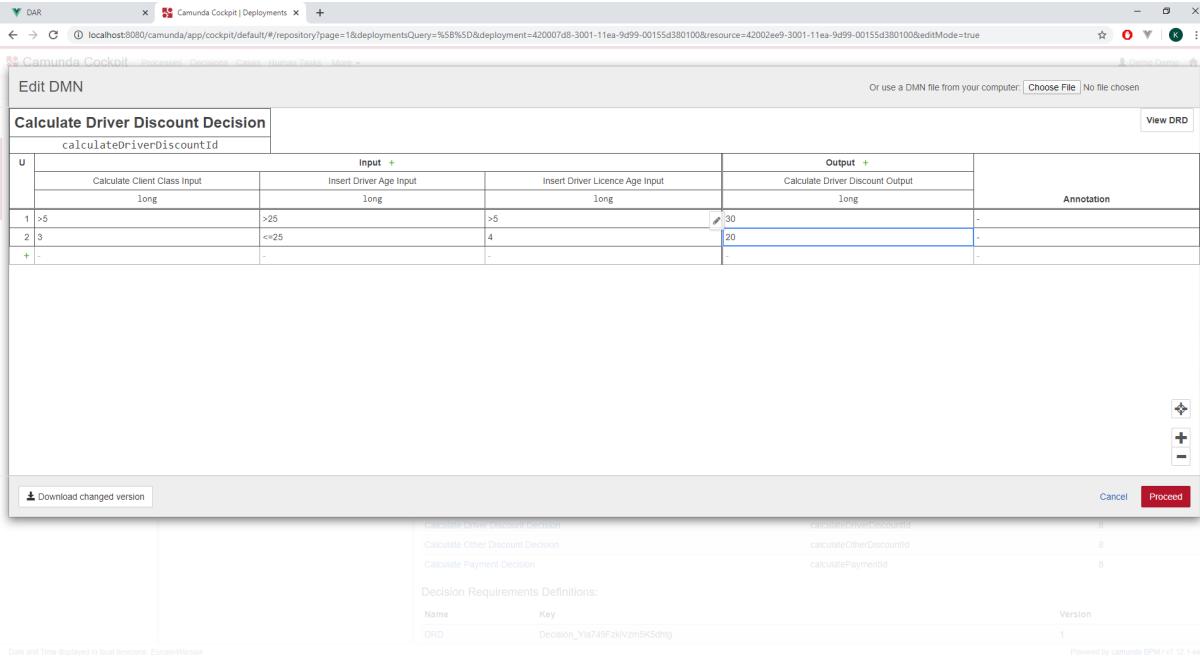
Rys. 5.6. Ekran wdrożeń Camunda Cockpit



Rys. 5.7. Model BPMN pliku „example”



Rys. 5.8. Model DMN pliku „example”

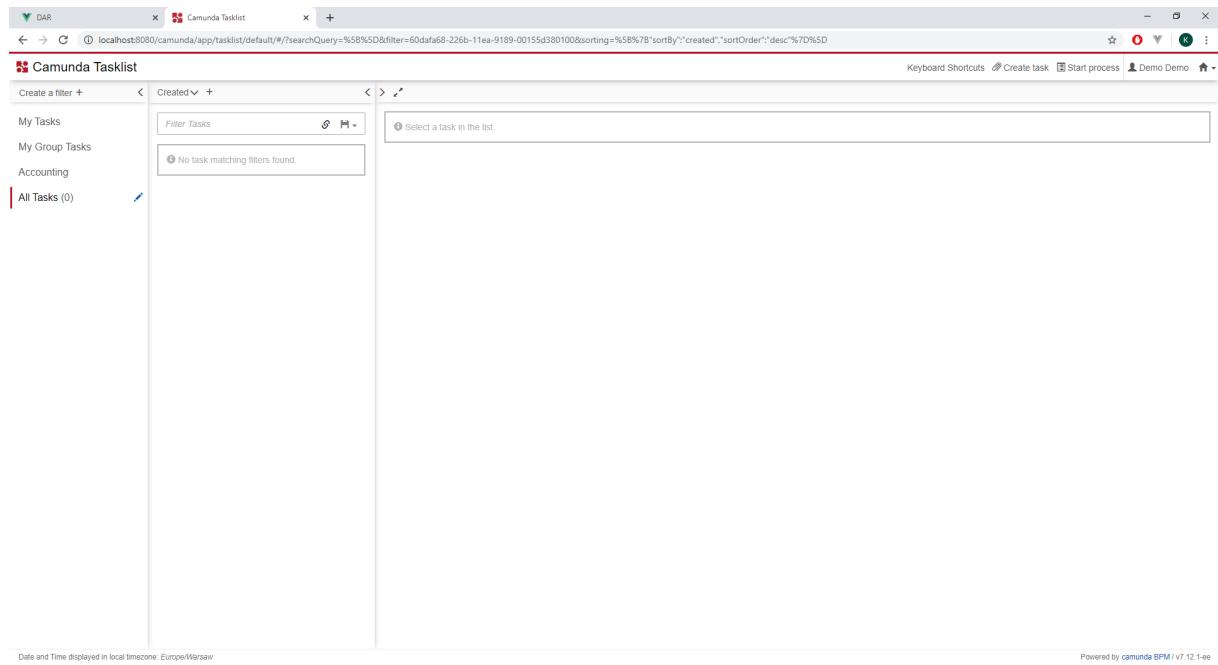


Rys. 5.9. Edycja tablicy decyzyjnej

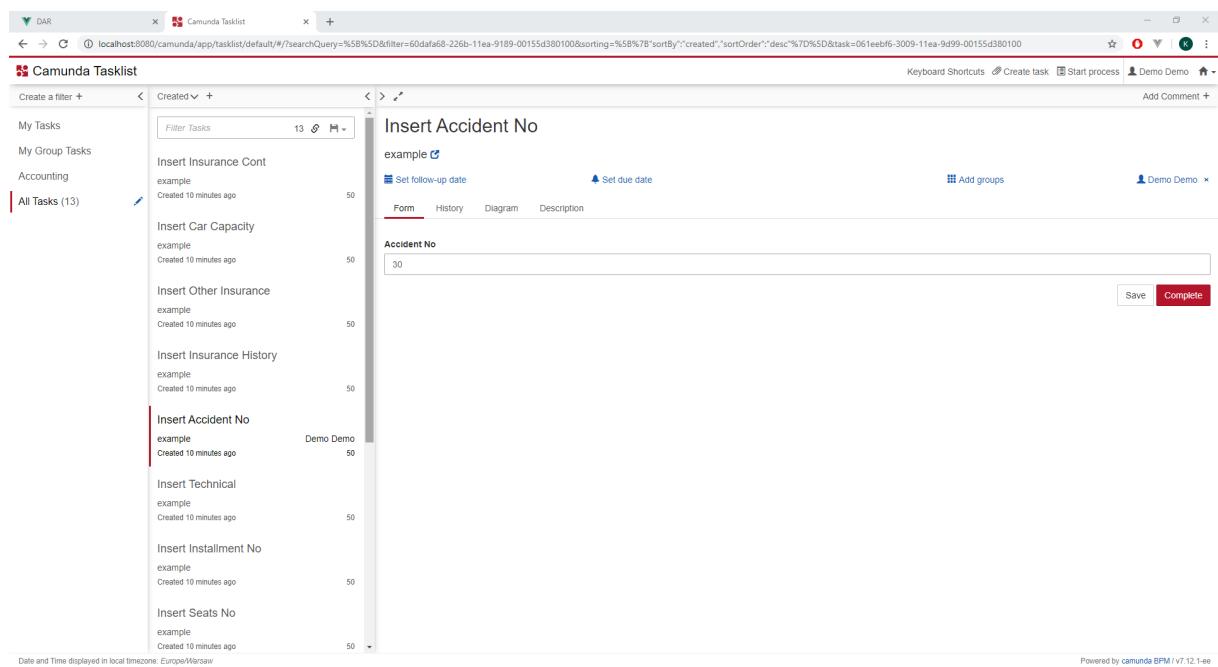
wszystkie zadania wykonywalne przez użytkownika potrzebne do działania procesu, np. zadania związane z podaniem danych wejściowych. Wyświetlone zadania można odpowiednio filtrować.

Aby uruchomić proces, należy wybrać przycisk „Start process” w prawym górnym rogu, wybrać odpowiedni model *BPMN* (w tym przypadku będzie to „example”), nadać tej instancji procesu pewien identyfikator, dla przykładu będzie to „ExampleBusinessKey” i nacisnąć przycisk „Start”. Spowoduje to pojawienie się adnotacji, że proces został uruchomiony. Można w tym momencie wrócić do modułu *Camunda Cockpit*, gdzie na stronie z rysunku 5.5 w metrykach nastąpi zmiana – pokazane zostanie, że pewien proces działa. W zakładce „Processes” wyświetlona będzie instancja procesu, która po naciśnięciu pokaże aktualny stan wykonywania procesu. Jednak przed przejściem do wspomnianego modułu warto odświeżyć *Camunda Tasklist*. Pojawi się lista wykonywalnych zadań związanych z procesem, który został uruchomiony. Są to dane wejściowe, które musi podać użytkownik, aby proces kontynuować działanie. Rysunek 5.11 prezentuje opisaną listę oraz wypełnienie jednego z takich zadań. Na przykładzie wszystkie zadania zostały wykonane.

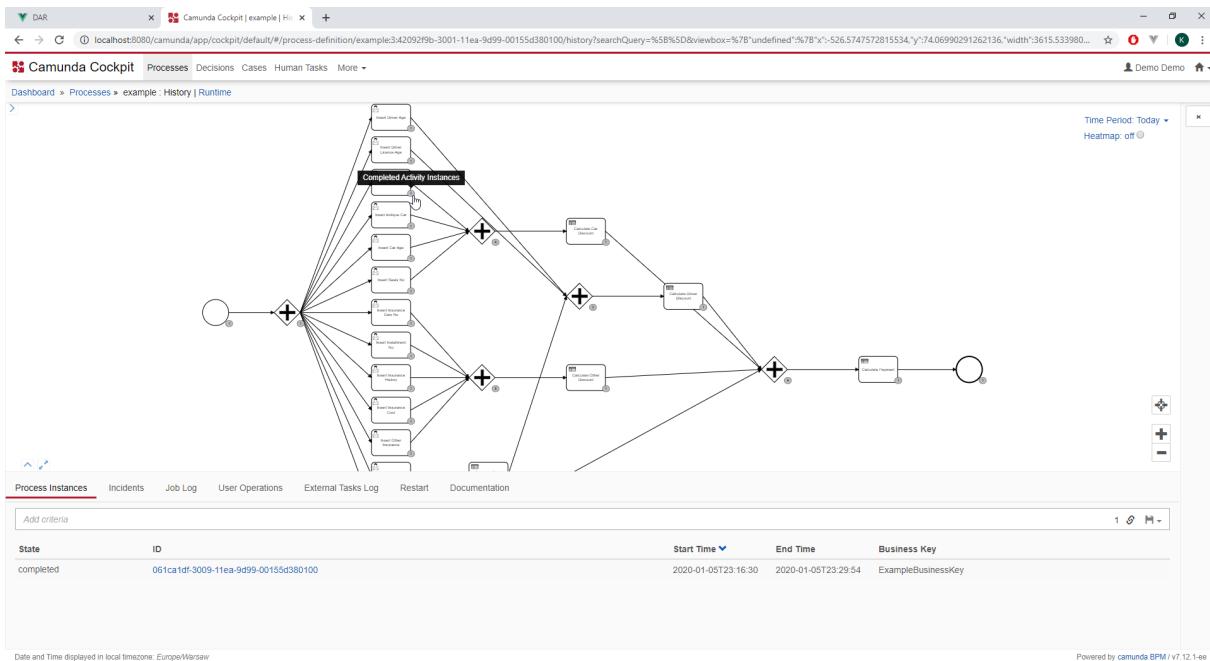
Po wykonaniu wszystkich zadań warto przejść do modułu *Camunda Cockpit* do zakładki „Processes”, wybrać tam odpowiedni proces i po wyświetleniu modelu *BPMN* nacisnąć przycisk „History”. Opcja „Runtime” w tym momencie nic nie wyświetla, ponieważ proces został zakończony, wszystkie decyzje zostały zewaluowane, jednak tak jak to było opisywane wcześniej, w przypadku wykonywania procesu to jest miejsce, gdzie można śledzić działanie procesu. Rysunek 5.12 prezentuje widok „History”. Dodatkowo można w tym miejscu sprawdzić jak rozkładała się praca procesu, aby ewentualnie go usprawniać. Rysunek 5.13 przedstawia przydatny do tego widok.



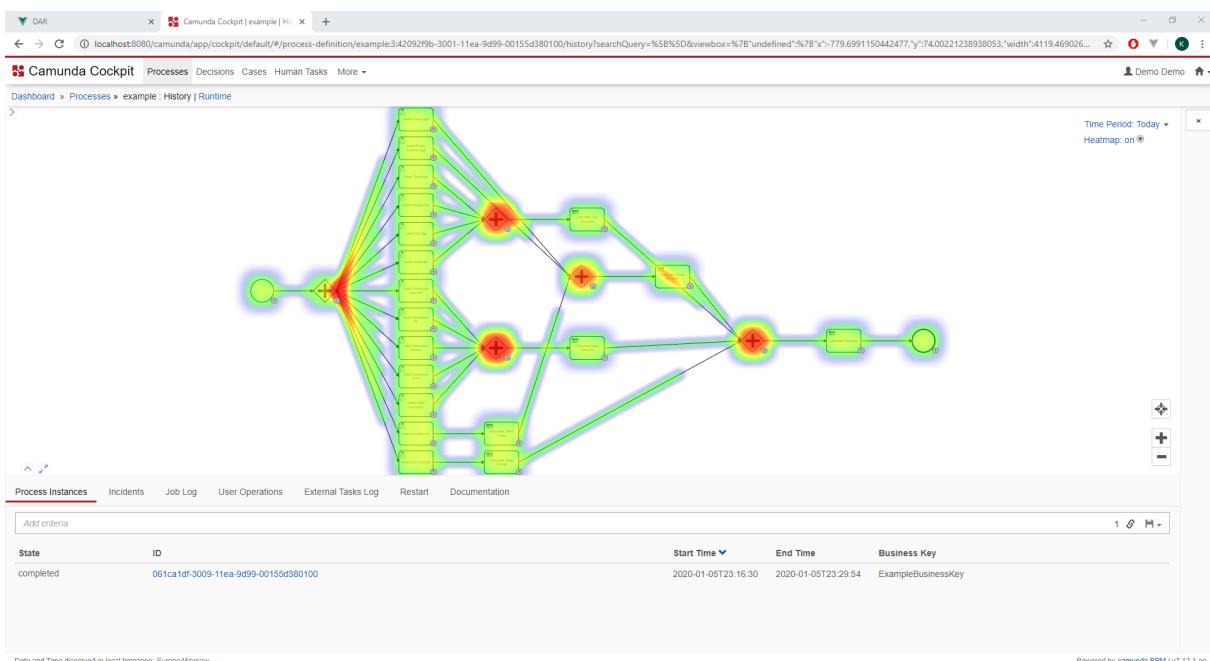
Rys. 5.10. Domyślny ekran Camunda Tasklist



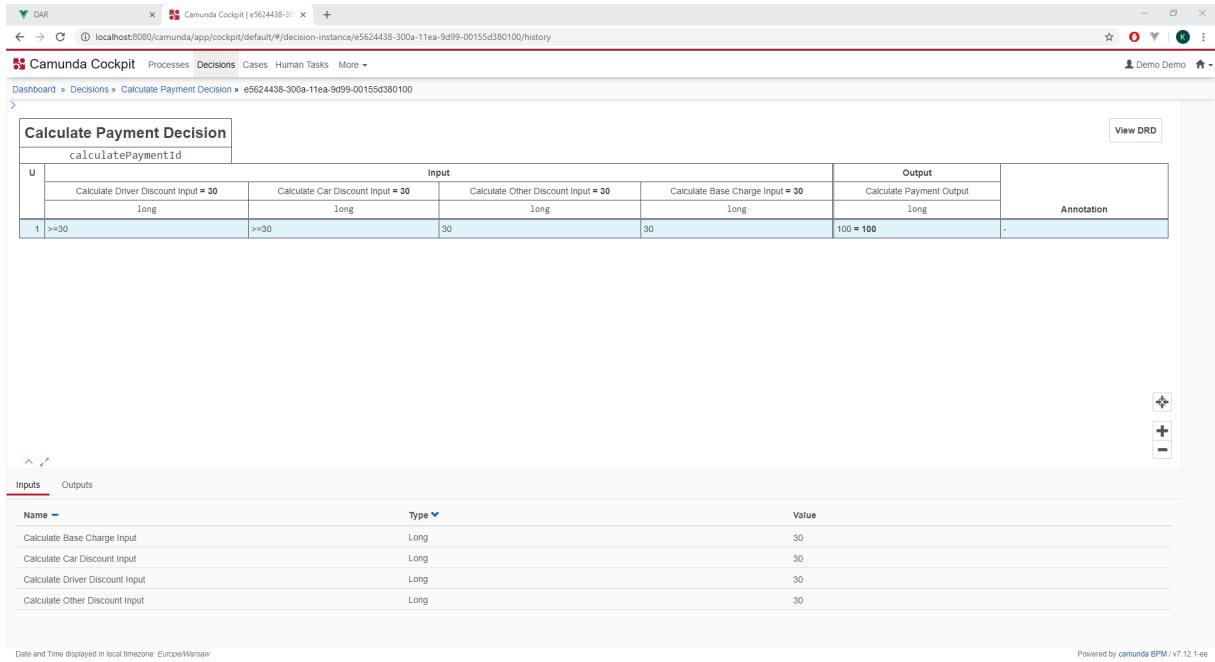
Rys. 5.11. Wypełnione zadanie wraz z listą oczekujących zadań



Rys. 5.12. Historia wykonywania procesu



Rys. 5.13. Rozkład pracy procesu



Rys. 5.14. Wynik procesu

W dolnej części rysunku 5.12 widać instancję zakończonego procesu. Po naciśnięciu „ID” tej instancji, widok modelu zostaje ten sam, jednak dochodzi wiele zakładek związanych z historią wykonywania tego procesu, np. jakie zadania zostały wykonane, jakie były wartości danych i jakie decyzje zostały ewaluowane. Aby zobaczyć wynik końcowy wystarczy przejść do ewaluowanych decyzji i wybrać finalną decyzję, czyli w prezentowanym przykładzie „Calculate Payment”. Po wybraniu tej decyzji zostaje tylko przejść do jej instancji. Rysunek 5.14 przedstawia wynik całego procesu i jakie były finalne wartości atrybutów – koszt ubezpieczenia w tym przypadku wyniósł 100 PLN. Oczywiście w ramach przykładu reguły są trywialne, w przedstawionej tablicy decyzyjnej występuje tylko jeden wiersz reguł, jednak w prawdziwym świecie może ich być dowolna liczba.

5.4. Podsumowanie wyników

Aplikacja spełnia wszystkie założone wymagania. Generowane modele w notacji *BPMN* oraz *DMN* poprawnie opisują diagram *ARD* zapisany w podanym pliku *HML*. Interfejs aplikacji „DAR“ jest intuicyjny i przejrzysty. Kod aplikacji w większości jest asynchroniczny co sprawia, że aplikacja jest płynniejsza w działaniu. Zintegrowany system *Camunda* poprawnie uruchamia wdrożone procesy. W przypadku decyzji, ewaluacja również jest poprawna, wymogiem jest tylko uzupełnienie reguł przez użytkownika. Sam wynik końcowy jest zadowalający i zgodny z oczekiwaniami.

Jest to koniec rozdziału przedstawiającego działanie aplikacji. W tym rozdziale przedstawione zostały najważniejsze funkcjonalności systemu na konkretnym przykładzie. W kolejnym, końcowym rozdziale przedstawione zostaną wnioski nasuwające się po zapoznaniu się z aplikacją oraz możliwe kierunki rozwoju aplikacji.

6. Podsumowanie

Rozdział kończy niniejszą pracę podsumowując działanie systemu oraz opisując zebrane wnioski. Dodatkowo przedstawia możliwe kierunki rozwoju aplikacji.

6.1. Wnioski

W ramach niniejszej pracy został osiągnięty cel, jakim było zaprojektowanie i zaimplementowanie aplikacji internetowej, będącej generatorem modeli w notacji *BPMN* oraz *DMN* na bazie plików *HML*, opisujących diagramy *ARD* oraz integrację z zewnętrznym systemem wspierającym zarządzanie procesami biznesowymi oraz ewaluację decyzji.

Praca całościowo opisuje proces tworzenia aplikacji „DAR“, wprowadzając najpierw najważniejsze pojęcia używane w pracy, by następnie przedstawić projekt aplikacji z odpowiednimi schematami i jego implementację wraz z opisem wykorzystanych technologii. Finalnie praca prezentuje przebieg działania aplikacji na konkretnym przykładzie, pokazując dokładnie interfejs użytkownika.

W projekcie wymagające dodatkowej implementacji okazało się tworzenie modeli *BPMN* oraz *DMN*, wszystko za sprawą nieaktualnych bibliotek do modelowania obiektów *BPMN* oraz *DMN* w środowisku .NET. Zadaniem prostym w implementacji było stworzenie odpowiednich definicji zadań lub przejść, jednak tworzenie na tej podstawie konkretnych obiektów na diagramie okazało się być wymagające. Sam algorytm do wyciągania informacji z pliku *HML* i odpowiednia transformacja informacji z diagramu *ARD* do *BPMN* był tworzony głównie z myślą o wydajności, mając na uwadze fakt, że w jednej chwili może być obsługiwanych wiele plików *HML*. Integracja z systemem *Camunda* również przysporzyła wiele problemów, ponieważ wymagała ona modyfikacji modeli *BPMN* oraz *DMN* o specyficzne dla tego systemu adnotacje i informacje w plikach wdrażanych na ten system. Organizacja tworząca system *Camunda* zakłada, że wszystkie modele wdrażane na platformę *Camunda* będą tworzone przez jej autorski edytor modeli. Ten edytor modeli nie udostępnia jednak *API*, dlatego zaszła potrzeba modyfikacji zaimplementowanych w aplikacji „DAR“ edytorów modeli, aby były zgodne ze standardem *Camunda*. Największym wyzwaniem było odkrycie tego problemu, ponieważ *API* wystawione przez *Camunda* służące do wdrożeń modeli nie zwracało żadnych konkretnych błędów.

6.2. Możliwe kierunki rozwoju aplikacji

Główną możliwością rozwoju aplikacji jest wzbogacenie procesu związanego z tworzenia modeli *BPMN* oraz *DMN*. W tym momencie wszystkie proste, niezależne własności z diagramu *ARD*, które potrzebne są jako dane wejściowe są umieszczane na samym początku modelu *BPMN* i dla każdego z nich przypisana jest jedna aktywność wykonywalna. Podobnie w przypadku skomplikowanych, zależnych własności – do każdej jednej takiej własności przypisana jest jedna decyzja z modelu *DMN*. Jest to przypadek głębokości poziomu zero (poziomy głębokości wy tłumaczone są w pracy [4]). Warto byłoby wprowadzić możliwość wyboru poziomu głębokości, aby proste niezależne własności były łączone w grupy o wielkości zależnej od głębokości i każdej takiej grupie byłoby przyporządkowywane odpowiednie zadania wykonywalne. Na tej samej zasadzie decyzje w zależności od poziomu głębokości mogą odpowiadać nie jednej skomplikowanej, zależnej własności, a przykładowo pewnej grupie.

Drugim z możliwych kierunków rozwoju jest wprowadzenie reguł decyzyjnych dla tablic decyzyjnych w modelu *DMN*. Gdyby plik *HML* rozszerzyć o takie reguły, aby mogły być one odpowiednio wydobyte i wdrożone do modelu *DMN* lub gdyby dostarczyć jakieś źródło takich reguł, tak aby użytkownik nie musiał ręcznie ich uzupełniać po wdrożeniu procesu, zaimplementowane rozwiązanie byłoby w pełni zautomatyzowane.

Pomysłem wartym rozważenia jest również rozbudowa aplikacji o wsparcie generacji modeli *ARD*. W tym momencie aplikacja potrafi przetwarzać pliki *HML*, opisujące diagram *ARD*, jednak nie jest w stanie użytkownikowi pokazać, jak taki diagram wygląda.

Kolejną sprawą jest część *TPH* w pliku *HML*. Zapisane w tej części informacje mogłyby być wykorzystane, tak jak w przypadku z poprzedniego akapitu, do zwrócenia diagramu *TPH*, aby użytkownik mógł go fizycznie zobaczyć. Dodatkowo wykorzystanie diagramów *TPH* pozwoliłoby ulepszyć dworzone modele *BPMN* oraz *DMN*.

Patrząc bardziej od strony technicznej – pierwszą rzeczą jest brak rozwiązania chmurowego. W tym momencie cała aplikacja jest uruchamiana lokalnie, potrzebny jest fizyczny komputer użytkownika wraz z serwerem bazy danych i trzema wolnymi portami odpowiednio dla aplikacji klienckiej, serwera „DAR” oraz serwera *Camunda*. Prostym rozwiązaniem jest wdrożenie całej aplikacji na chmurę. Dodatkowo dobrym pomysłem jest konteneryzacja systemu, szczególnie w przypadku gdy szkielet aplikacyjny *ASP .NET Core* jest bardzo dobrze przystosowany do konteneryzacji. To samo dotyczy się aplikacji klienckiej.

Finalnie możliwą zmianą jest zastąpienie systemu *Camunda* przez własną aplikację. Oczywiście byłoby to bardzo duże przedsięwzięcie, wymagające własnej implementacji silników procesowych i decyzyjnych oraz całej logiki z wykonywaniem procesów, a także odpowiednim prezentowaniu działań, z wyświetlaniem diagramów *BPMN* oraz *DMN* na czele.

Bibliografia

- [1] *Business Process Model and Notation (BPMN) Version 2.0.* Spraw. tech. Dostęp: 2019-12-20. Object Management Group (OMG), 2011.
- [2] *Decision Model and Notation (DMN) Version 1.2.* Spraw. tech. Dostęp: 2019-12-20. Object Management Group (OMG), 2019.
- [3] Grzegorz J. Nalepa i Igor Wojnicki. „Towards Formalization of ARD+ Conceptual Design and Refinement Method”. W: *FLAIRS-21: Proceedings of the twenty-first international Florida Artificial Intelligence Research Society conference: 15–17 may 2008, Coconut Grove, Florida, USA.* Red. David C. Wilson i H. Chad Lane. Menlo Park, California: AAAI Press, 2008.
- [4] Krzysztof Kluza i in. „From Attribute Relationship Diagrams to Process (BPMN) and Decision (DMN) Models”. W: *Knowledge Science, Engineering and Management.* Red. Christos Douligeris, Dimitris Karagiannis i Dimitris Apostolou. Cham: Springer International Publishing, 2019, s. 615–627. ISBN: 978-3-030-29551-6.
- [5] *Hekate Markup Language.* <https://ai.ia.agh.edu.pl/wiki/hekate:hml1>. Dostęp: 2019-12-20.
- [6] Szymon Drejewicz. *Zrozumieć BPMN. Modelowanie procesów biznesowych.* Wydawnictwo Helion, 2012. ISBN: 9788324634033.
- [7] *Modelowanie decyzji za pomocą notacji DMN.* <http://edusolution.pl/modelowanie-decyzji-za-pomoca-notacji-dmn-czesc-pierwsza/>. Dostęp: 2019-12-20.
- [8] *Drools Decision Tables in Spreadsheets.* https://training-course-material.com/training/Filip_Drools_Decision_Tables_in_Spreadsheets. Dostęp: 2019-12-20.
- [9] Grzegorz J. Nalepa i Antoni Ligęza. „Conceptual Modelling and Automated Implementation of Rule-Based Systems”. W: *Proceedings of the 2005 Conference on Software Engineering: Evolution and Emerging Technologies.* NLD: IOS Press, 2005, 330–340. ISBN: 1586035592.
- [10] Kereshmeh Afsari, Charles Eastman i Dennis Shelden. „Data Transmission Opportunities for Collaborative Cloud-Based Building Information Modeling”. W: *SIGraDi 2016, XX Congress of the Iberoamerican Society of Digital Graphics 9-11, November, 2016 -Buenos Aires, Argentina.* 2016, s. 907–913.
- [11] *Hypertext Transfer Protocol – HTTP/1.1.* <https://tools.ietf.org/html/rfc2616>. Dostęp: 2019-12-20.

- [12] *What is REST API?* <https://www.visual-paradigm.com/guide/development/what-is-rest-api/>. Dostęp: 2019-12-20.
- [13] *StackOverflow Developer Survey*. <https://insights.stackoverflow.com/survey/201>. Dostęp: 2019-12-20.
- [14] *Web Framework Benchmarks*. <https://www.techempower.com/benchmarks>. Dostęp: 2019-12-20.
- [15] Mark Seemann i Steven van Deursen. *Dependency Injection Principles, Practices, and Patterns*. Manning Publications, 2019. ISBN: 9781617294730.
- [16] Grzegorz J. Nalepa. *Semantic Knowledge Engineering. A Rule-Based Approach*. Kraków: Wydawnictwa AGH, 2011.