ReDSeeDS
Podręcznik Użytkownika

Lucyna Skrzypek, Piotr Łabęcki

Redakcja: Michał Śmiałek

Spis treści

1. Wprowadzenie

Głównym przeznaczeniem narzędzia ReDSeeDS jest wytwarzanie oprogramowania w znacznej części w oparciu o przypadki użycia. Projekt ten wciąż jest w fazie rozwoju, lecz efekty uzyskane na dzień dzisiejszy oraz przyjęte plany pozwalają z optymizmem patrzeć na realizację wyżej opisanej wizji w niedalekiej przyszłości.

Narzędzie ReDSeeDS oparte jest na platformie Eclipse. Sama platforma Eclipse jest powszechnie znana i używana, w związku z czym nie ma potrzeby szerszego jej omawiania. Kod generowany przez narzędzie ReDSeeDS jest oparty na frameworku Echo 2. W warstwie bazy danych zostanie opisana integracja z technologią H2.

W niniejszym rozdziale przedstawione zostaną istotne aspekty związane z pracą nad wymaganiami oprogramowania definiowanymi w narzędziu ReDSeeDS. Omówione zostaną takie kwestie jak: tworzenie nowego projektu, konwencja pisania scenariuszy przypadków użycia, dodawanie i zarządzanie pojęciami oraz proces walidacji efektów pracy i generowania kodu. W trakcie omawiania wyżej wymienionych zagadnień, autorzy będą przedstawiali fragmenty wykonanej przez siebie specyfikacji wymagań jako przykłady przedstawianych operacji.

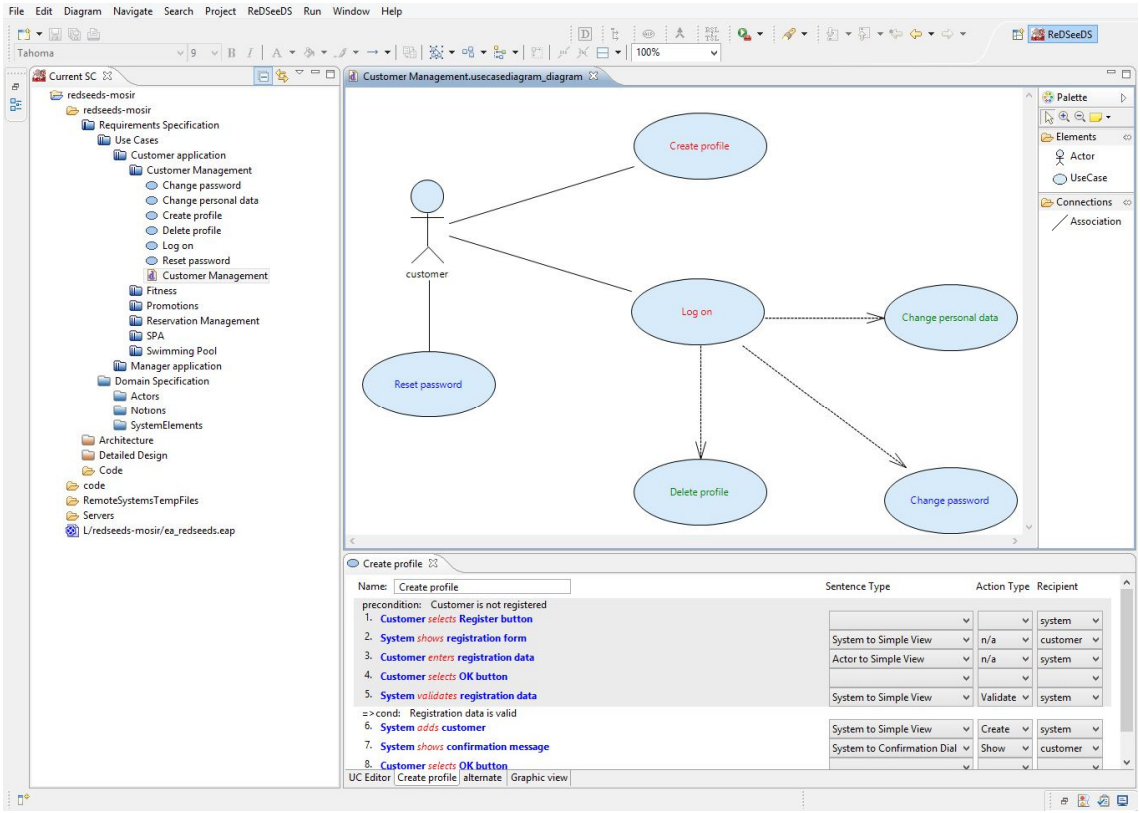
1.1. Co to jest ReDSeeDS?

ReDSeeDS był projektem realizowanym przez zespoły badawcze z wielu krajów Europy, takich jak Niemcy, Polska czy Wielka Brytania. Główną cechą narzędzia ReDSeeDS jest możliwość tworzenia oprogramowania w oparciu o specyfikację wymagań. Powoduje to zmianę proporcji udziału analityka i programisty w procesie wytwarzania oprogramowania na korzyść analityka. ReDSeeDS wykorzystuje wszystkim znaną notację przypadków użycia, scenariuszy i słownika pojęć, co ilustruje przykład na rysunku 1. Przy poprawnej ich definicji według określonego wzorca, możliwe staje się automatyczne wygenerowanie znacznej ilości kodu aplikacji. Skutkuje to ograniczeniem roli programisty tylko do odpowiedniego uzupełnienia wygenerowanego przez narzędzie ReDSeeDS kodu.

Wymagania wprowadzane do ReDSeeDSa formułowane są w bardzo intuicyjnym języku RSL (Requirements Specification Language) za pomocą wbudowanego w narzędzie edytora RSL. Jest to fundament działania tego narzędzia, a jego budowę w najprostszy sposób można zobrazować poniższą zależnością:

RSL = aktorzy + przypadki użycia + scenariusze + słownik pojęć

RSL jest pierwszym językiem łączącym ze sobą cechy słownika i modelowania pojęciowego, co znacznie ułatwia zrozumienie wymagań. Język ten umożliwia definiowanie relacji pomiędzy elementami dziedziny i zobrazowanie ich na diagramach, a także określa, w jaki sposób należy formować wyrażenia i pojęcia, aby potem możliwe było poprawne wygenerowanie architektury aplikacji. Pod tym względem język RSL czerpie wiele schematów działań z języka UML. W scenariuszach zalecane jest budowanie zdań jak najprostszych, zgodnych ze schematem Podmiot-Orzeczenie-Dopełnienie.



Rysunek 1: Typowy ekran narzędzia ReDSeeDS

Charakterystyczne dla aktualnej wersji narzędzia ReDSeeDS jest generowanie kodu w języku programowania Java w architekturze Model-View-Presenter. Na podstawie modelu wymagań automatycznie tworzona jest większość logiki aplikacji z wykorzystaniem frameworka aplikacji webowych Echo3, a także obiekty DAO (Data Access Objects) oraz DTO (Data Transfer Objects).

1.2. Baza danych H2 i framework Echo 3

Baza danych H2 jest bazą danych typu open source (licencja Mozilla Public License). H2 operuje w głównej mierze na zapytaniach napisanych w języku SQL, ale jeśli zaistnieje taka potrzeba, możliwe jest również użycie standardu ODBC PostgreSQL. Baza danych H2 ma możliwość osadzenia w aplikacji Java lub uruchomienia w trybie klient-serwer. Dużą jej zaletą jest również dostępność opcji skonfigurowania H2 jako bazy danych w pamięci, przez to żadne dane nie będą utrwalone na dysku. Istotnym aspektem jest również fakt spełnienia przez H2 wymogów bezpieczeństwa, ponieważ zawiera wbudowaną barierę ochronną, która uniemożliwia wstrzyknięcie niepożądanego kodu z zewnątrz w przetwarzane zapytanie. Przy generowaniu architektury aplikacji w narzędziu ReDSeeDS, to baza H2 jest wykorzystywana do generowania warstwy bazy danych.

Inną technologią domyślnie wybraną przy tworzeniu projektu w narzędziu ReDSeeDS jest framework aplikacji webowych Echo3. Podobnie jak baza danych H2 jest on również dystrybuowany na licencji typu open source. Z punktu widzenia programisty działa podobnie do innych narzędzi interfejsu użytkownika jak np. Swing lub Eclipse SWT. Oferuje znacznie prostsze API do tworzenia niestandardowych komponentów AJAX, które będą używać własnego kodu HTML i JavaScript.

Framework Echo3 daje możliwość tworzenia aplikacji, które mogą być wykonywane zarówno po stronie klienta, jak i po stronie serwera. Jeśli programista chce, aby aplikacja wykonywana była po stronie klienta, musi napisać ją w języku JavaScript. Po wykonaniu aplikacji w przeglądarce klienta, w celu ponownej komunikacji z serwerem, oprócz protokołu HTTP zostanie użyty inny język np. JSON lub XML. Wykonywanie aplikacji po stronie klienta może spowodować pewne opóźnienia czasowe w działaniu aplikacji. Dzieje się tak wówczas, gdy kod interfejsu użytkownika odwołuje się do zasobów, które znajdują się po stronie serwera np. do bazy danych.

Drugim możliwym podejściem stosowanym w technologii Echo3 jest stworzenie aplikacji działającej po stronie serwera. W tym wypadku, zachodzi konieczność napisania kodu aplikacji w języku Java, choć obecnie trwają prace nad udostępnieniem innych języków. Aplikacja po stronie serwera komunikuje się z tzw. cienkim klientem, dostarczanym przez framework. Zadaniem tego cienkiego klienta jest synchronizacja swojego stanu komponentu ze stanem komponentu po stronie serwera oraz wysyłanie wszelkich zdarzeń generowanych przez użytkownika z powrotem na serwer.

2. Praca w ReDSeeDS

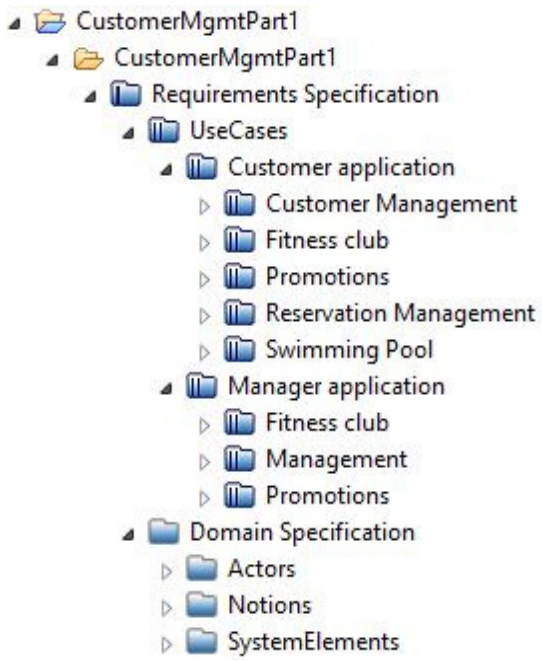
2.1. Tworzenie projektu i diagramów przypadków użycia

Pierwszą czynnością jaką należy wykonać jeszcze przed samym rozpoczęciem pracy w narzędziu ReDSeeDS jest uruchomienie serwera terminologii JWGNL, znajdującego się w odpowiednim folderze w zasobach ReDSeeDSa. Serwer ten musi być uruchomiony w tle przez cały czas pracy w narzędziu, a po jej zakończeniu powinien zostać zamknięty poprzez wpisanie w otwartym terminalu komendy „quit”.

Główny powód, dla którego konieczne jest działanie serwera JWGNL to zamiar implementacji dosyć istotnej funkcjonalności w przeszłości. Tą funkcjonalnością była możliwość ponownego wykorzystania istniejącego projektu do implementacji innego, podobnego oprogramowania. Serwer JWGNL przechowuje opisy znaczeń wielu pojęć słownikowych wykorzystywanych w projekcie. Znaczenia tych pojęć mogą się różnić w zależności od projektu i powiązania są przechowywane w serwerze terminologii.

Po uruchomieniu serwera terminologii JWGNL, należy uruchomić narzędzie ReDSeeDS. Następnie, w celu stworzenia nowego projektu użytkownik powinien wybrać opcję **File → New Project**, a w kolejnym kroku wybrać opcję **New Software Case Project** i nadać projektowi nazwę. Po tej czynności niezbędne jest utworzenie nowego pakietu wymagań poprzez wybranie opcji **Requirement Specification → New → Requirements Package**.

Możliwe jest utworzenie wielu pakietów wymagań, w zależności od potrzeby. W omawianym projekcie specyfikacja wymagań została podzielona logicznie na podpakiety: najpierw ze względu na rodzaj użytkownika (klient lub manager) a następnie każdy z nich na odpowiednie moduły funkcjonalne (klub fitness, pływalnia, promocje, zarządzanie rezerwacjami itp.). Taki podział wprowadza do projektu ład i porządek, co znacznie ułatwia późniejszą pracę nad projektem. Efekt tego działania jest widoczny na rysunku 2.



Rysunek 2: Podział projektu na pakiety

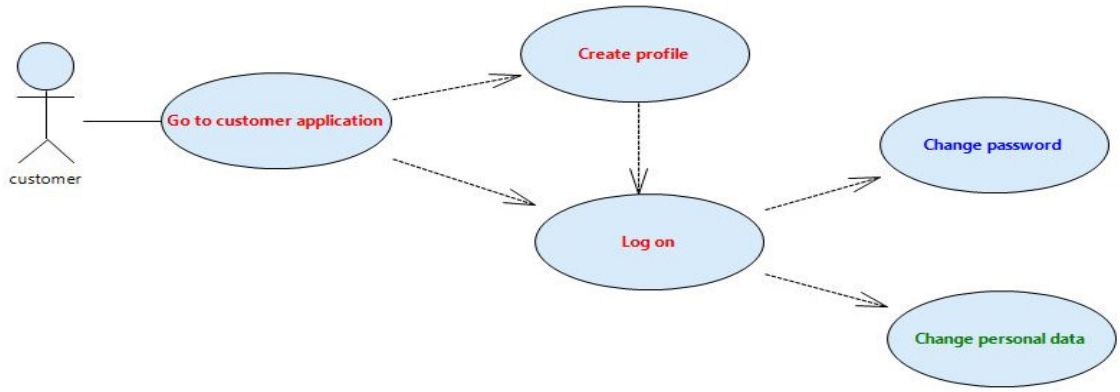
Po stworzeniu podpakietów przychodzi pora na dodawanie do nich przypadków użycia. Aby dodać taki element, należy najpierw stworzyć diagram przypadków użycia używając opcji **New → Use Cases Diagram**. Dzięki temu otworzy się nowa strona, na której będzie możliwe układanie przypadków użycia oraz aktorów. Aby dodać przypadek użycia wystarczy przeciągnąć element **Use Case** lub **Actor** z paska bocznego **Palette** lub użyć menu kontekstowego **Add → Use Case** lub **Add → Actor**. W przypadku dodania nowego aktora na diagramie, automatycznie jest on również dodawany w folderze **Domain Specification → Actors**.

Po umieszczeniu na diagramie wszystkich wybranych aktorów i przypadków użycia można połączyć relacją aktorów z przypadkami użycia poprzez przeciągnięcie linii z jednego elementu do drugiego. Należy zauważyć, że powyższe działanie nie znajduje zastosowania, jeżeli użytkownik chce połączyć ze sobą relacjami przypadki użycia. Aby zobaczyć na diagramie takie połączenie np. pomiędzy dwoma przypadkami użycia, należy w scenariuszu jednego z przypadków użycia dodać zdanie typu Invoke/Insert (więcej o pisaniu scenariuszy w dalszej części rozdziału). Następnie z listy rozwijanej użytkownik powinien wybrać, do którego elementu docelowego odnosi się ta relacja. Po tym wyborze i zapisaniu zmian, na diagramie pojawi się linia przerywana łącząca wybrane przypadki użycia. Istotny jest fakt, że w języku RSL nie wyróżnia się relacji Extend oraz Include, spotykanych w języku UML.

W przykładowym projekcie przyjęto iteracyjny sposób wytwarzania oprogramowania, w związku z czym stworzone przypadki użycia zostały podzielone na 3 iteracje od najbardziej istotnych funkcjonalności do najmniej istotnych. Umożliwiło to łatwiejsze zapanowanie nad funkcjonalnościami i stanem prac. Na diagramach przypadków użycia iteracje te oznaczono następującymi kolorami:

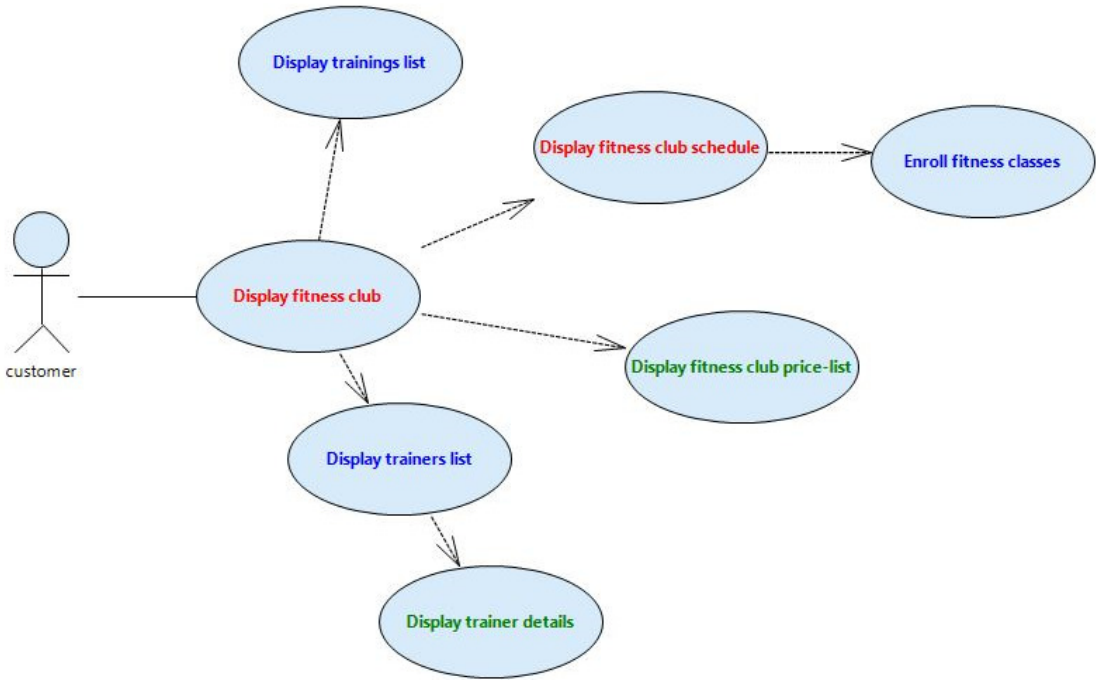
- 1 iteracja
- 2 iteracja
- 3 iteracja

Na rysunkach 3 – 7 widoczne są pakiety dla klienta. Pakiet „Zarządzanie kontem klienta” jest przestawiony na rysunku 3. Umożliwia on zarządzanie swoim kontem pod takimi aspektami jak stworzenie nowego konta w systemie, logowanie się, a także zmiana hasła oraz podanych wcześniej danych osobowych.



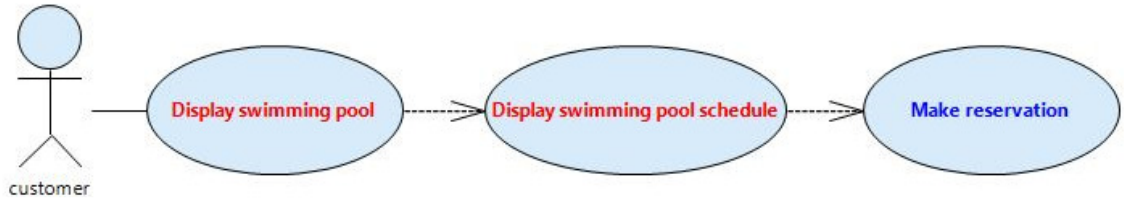
Rysunek 3: Zarządzanie kontem klienta

Na Rysunku 4 widoczny jest najbardziej rozbudowany pakiet przypadków użycia klienta, czyli „Fitness club”. Za jego pomocą klient ma możliwość przeglądania takich elementów, jak lista treningów, lista trenerów (wraz z ich szczegółowym opisem), cennik oraz grafik zajęć. Po wybraniu odpowiednich parametrów, klient może również zapisać się na konkretne zajęcia.



Rysunek 4: Klub fitness

Dodatkowo, klient ma do dyspozycji przypadki użycia w pakietach „Pływalnia”, „Promocje” i „Zarządzanie rezerwacjami”, co pokazują rysunki 5-7.

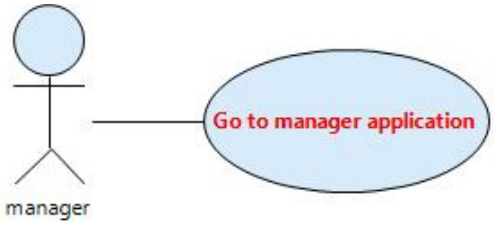


Rysunek 5: Pływalnia

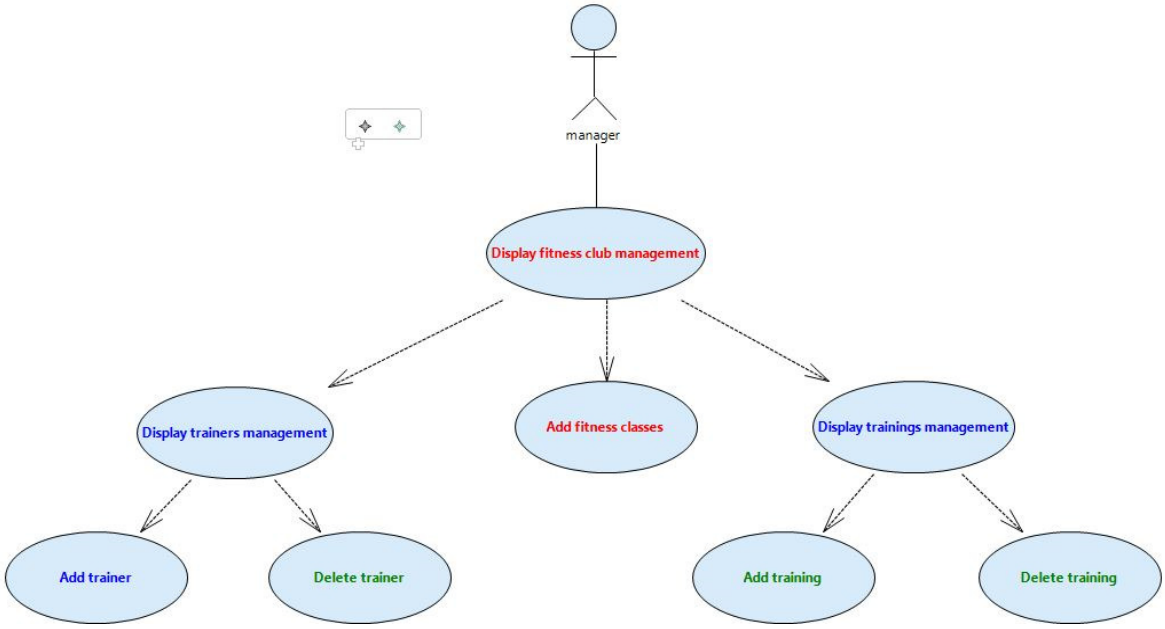
Rysunek 6: Promocje

Rysunek 7: Zarządzanie rezerwacjami

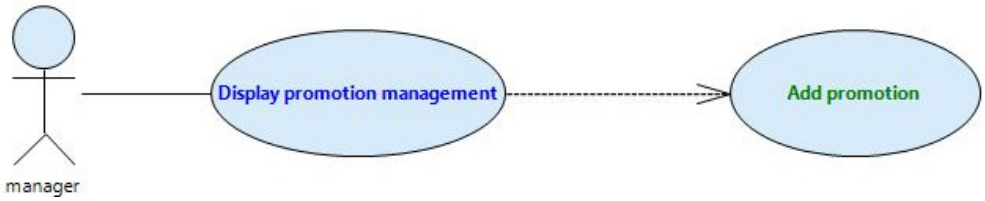
Na Rysunkach 8– 10 widoczne są pakiety dotyczące managera. Funkcjonalność managera w systemie ogranicza się do najbardziej istotnych aspektów, bez których nie byłoby możliwe prawidłowe działanie modułu klienta. Najbardziej istotnym i rozbudowanym modułem z punktu widzenia managera jest „Zarządzanie fitness clubem" (rysunek 9).



Rysunek 8: Panel managera



Rysunek 9: Zarządzanie klubem fitness



Rysunek 10: Zarządzanie promocjami

2.2. Gramatyka i terminologia

Po zdefiniowaniu przypadków użycia i zależności pomiędzy nimi, kolejnym krokiem jest napisanie scenariuszy. W tym celu po otworzeniu szczegółów danego przypadku użycia należy przejść do zakładki scenariusza (o tej samej nazwie co przypadek użycia). Użytkownik musi zainicjować pisanie scenariusza wciskając przycisk **Create SVO Sentence**. Następnie można rozpocząć wprowadzanie kolejnych zdań w scenariuszu. Powyżej pierwszego zdania znajduje się sekcja **Precondition**, w której użytkownik może wprowadzić warunki początkowe, jakie muszą być spełnione, aby dany przypadek użycia się rozpoczął.

Pisanie scenariuszy powinno odbywać się zgodnie z odpowiednią konwencją. Zdania powinny mieć budowę Podmiot – Orzeczenie – Dopełnienie, czyli np. „*System* **zapisuje** dane książki”. Możliwe jest również dodawanie dopełnienia dalszego, ale taka praktyka nie jest zalecana. Generalną zasadą jest budowanie zdań możliwie jak najkrótszych i najprostszych. Pozwoli to uniknąć generowania niepotrzebnych elementów w kodzie aplikacji. Zdecydowanie odradza się używania w zdaniach scenariusza polskich znaków, które wprawdzie są poprawnie obsługiwane przez narzędzie ReDSeeDS, ale w późniejszym etapie znaki te są przenoszone na kod. Może to uniemożliwić jego skompilowanie.

Użytkownik ma możliwość utworzenia scenariusza alternatywnego. Aby to zrobić, należy w trakcie pisania scenariusza głównego wybrać opcję **Fork scenario**. Efektem tego jest pojawienie się sekcji, w której można wpisać warunek, jaki musi zostać spełniony,

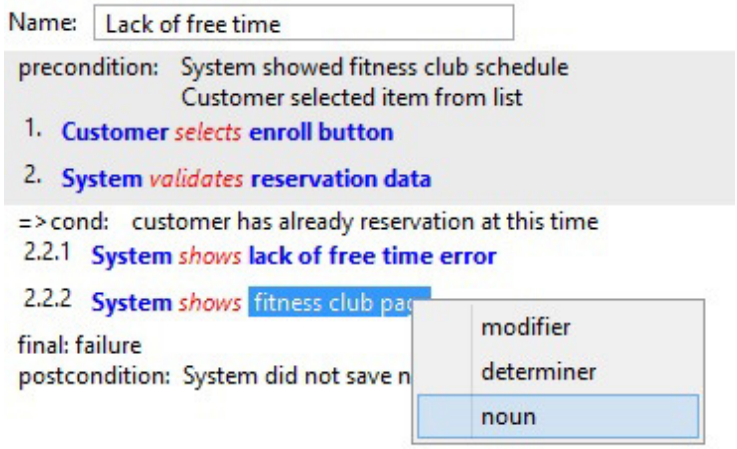
aby scenariusz główny był kontynuowany. Jednocześnie zostanie utworzona nowa karta w szczegółach danego przypadku użycia, zawierająca nowy scenariusz alternatywny.

Warto zauważyć, że jeśli użytkownik wybierze opcję **Fork scenario** w trakcie pisania scenariusza głównego (np. po 4 zdaniach), w scenariuszu alternatywnym będą widoczne wszystkie zdania, znajdujące się w scenariuszu głównym przed wywołaniem opcji **Fork scenario**. Pod nimi ponownie pojawia się sekcja, w której użytkownik wprowadza warunek jaku musi zostać spełniony, aby rozpoczęła się realizacja scenariusza alternatywnego. Zdania znajdujące się pod tym warunkiem, mają inną numerację, która odnosi się do poprzedzającego zdania (np. 4.1.1, 4.1.2 itd.).

Po zakończeniu pisania scenariusza alternatywnego, użytkownik ma możliwość powrotu do scenariusza głównego wybierając opcję **Create rejoin sentence**. Po jej wywołaniu użytkownik musi wybrać, do którego scenariusza i zdania chce powrócić. Jeśli w przypadku użycia występuje więcej niż 1 scenariusz alternatywny, można wybierać dowolny scenariusz, niekoniecznie musi być to scenariusz główny.

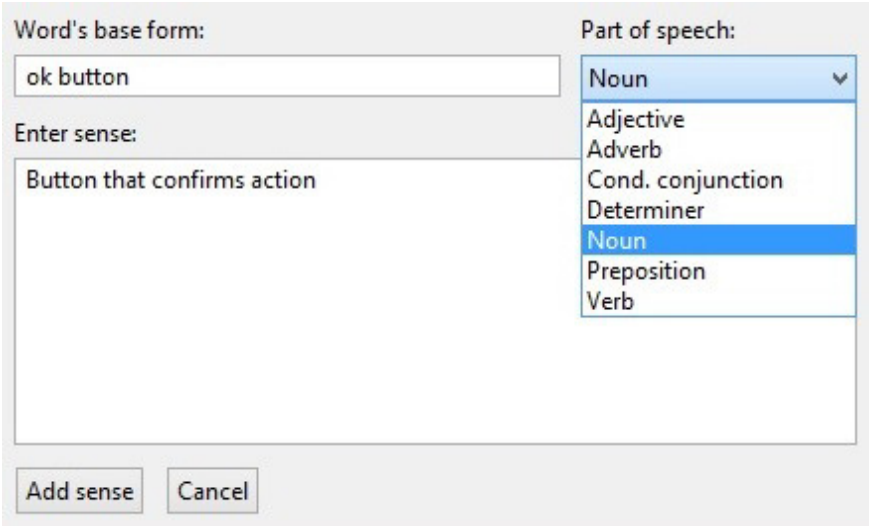
Po zakończeniu pisania każdego ze scenariuszy konieczne jest oznaczenie statusu, z jakim zakończył się dany scenariusz. Użytkownik ma do wyboru 2 opcje: **Create final/success sentence** (w przypadku pozytywnego zakończenia) oraz **Create final/failure sentence** (w przypadku negatywnego zakończenia np. wystąpienia błędu). Po wybraniu jednego z nich pod zdaniami pojawia się status **success** lub **failure** oraz sekcja **Postcondition**, w której użytkownik wprowadza warunek, jaki musi być spełniony, aby dany scenariusz się zakończył. Samo wprowadzanie warunku początkowego i końcowego nie jest konieczne do wygenerowania kodu aplikacji, ale warto to robić z 2 powodów. Pierwszym z nich jest fakt, że wprowadza to przejrzystość w specyfikacji wymagań i pomaga w zrozumieniu działania systemu. Drugi powód to poprawienie przejrzystości generowanego kodu, gdyż wprowadzone warunki są potem przeniesione do kodu jako komentarze przy pętlach warunkowych.

Gdy użytkownik napisze już wszystkie scenariusze dla danego przypadku użycia, kolejnym krokiem jest oznaczenie elementów tych zdań. Rekomenduje się przyjęcie właśnie takiej kolejności działań czyli najpierw napisanie scenariuszy, potem oznaczenie elementów zdań. Pomaga to w wychwyceniu błędów w scenariuszach i nie wytrąca ze skupienia nad poprawnością merytoryczną scenariusza. Aby oznaczyć elementy zdań w scenariuszach, należy zaznaczyć dane słowo lub wyrażenie oraz wybrać z menu kontekstowego odpowiednią część zdania (np. rzeczownik, czasownik czy dopełnienie). Po dokonaniu wyboru każdy element jest oznaczony kolorem odpowiadającym części zdania czyli tj. rzeczowniki na niebiesko, czasowniki na czerwono itp.. Przykład oznaczania elementów zdań został przedstawiony na rysunku 11.



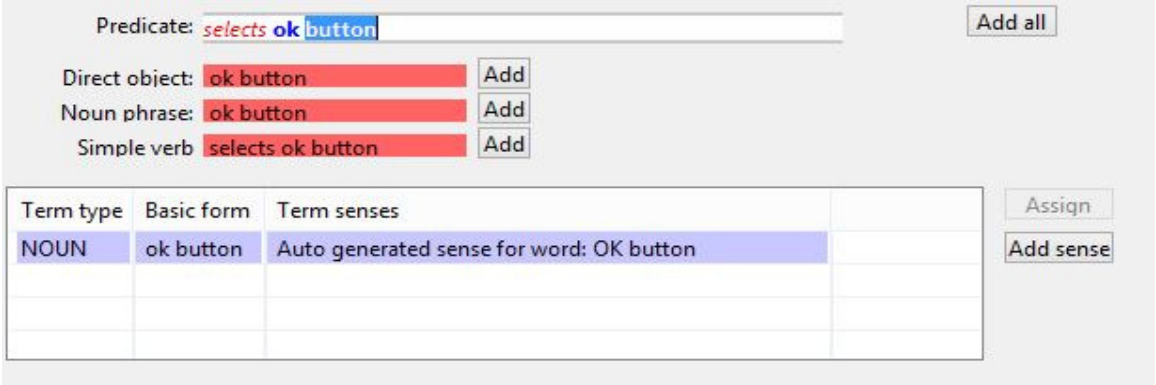
Rysunek 11: Oznaczanie elementów zdania

Po oznaczeniu elementów wszystkich zdań w danym scenariuszu, użytkownik może przystąpić do nadawania pojęciom znaczeń. Jest to czynność przydatna głównie w przypadku ponownego użycia danej specyfikacji. W pozostałych przypadkach nie ma to wielkiego znaczenia, ale trzeba ten proces wykonać by móc wygenerować kod aplikacji. Nadawanie pojęciom sensu odbywa się poprzez dwukrotne kliknięcie na dane pojęcie. Następuje wtedy otwarcie zakładki **DomainStatementEditor**, w której użytkownik wybiera z listy, które z dostępnych znaczeń danego pojęcia chce wybrać. Jeśli żadne z dostępnych znaczeń mu nie odpowiada, możliwe jest dodanie własnego znaczenia wybierając opcję **Add sense**. Pojawi się wówczas okno, w którym należy wybrać jaką częścią mowy jest dany element oraz wpisać jego sens. Przykład takiej operacji przedstawiono na Rysunku 17.



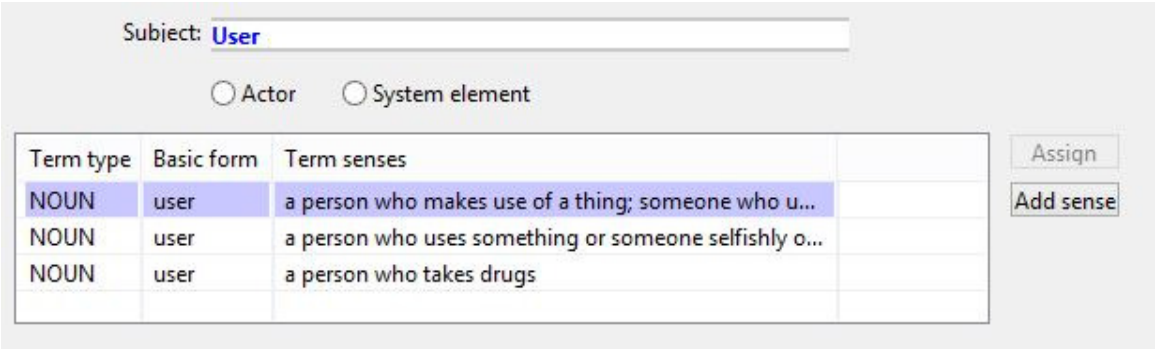
Rysunek 17: Dodawanie nowego znaczenia

Jeśli frazy są podświetlone na kolor czerwony oznacza to, że nie zostały wygenerowane i dodane do słownika. W wykorzystanej wersji transformaty dodawanie znaczeń do pojęć odbywa się automatycznie. Po przypisaniu znaczeń wszystkim danym należy wcisnąć przycisk **Add all**. Wtedy zaznaczone pojęcie zostanie dodanie do dziedziny aplikacji czyli do folderu **Domain Specification → Notions**. Proces nadawania znaczenia do pojęcia przedstawiono na rysunku 12.



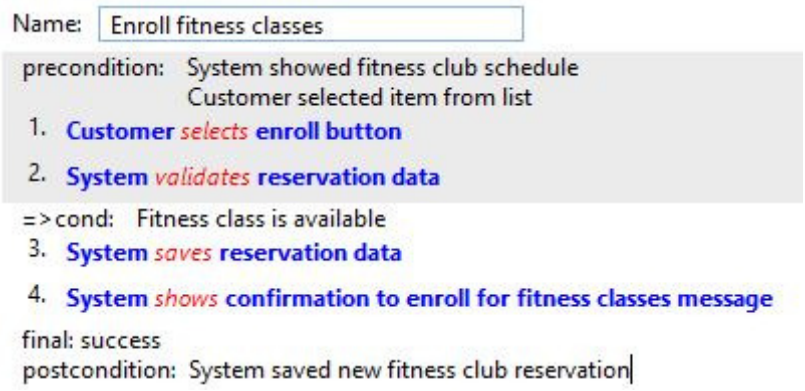
Rysunek 12: Przypisywanie znaczenia do pojęcia

Z powodu przyjętej konwencji pisania zdań w scenariuszach czyli Podmiot – Orzeczenie – Dopełnienie, ReDSeeDS automatycznie rozpoznaje, że pierwsze słowo w każdym zdaniu oznacza albo aktora albo system. W związku z tym, przy przypisywaniu znaczenia pierwszemu słowu, użytkownik nie ma wyświetlonych danych pojęcia, lecz ma do wyboru 2 opcje: **Actor** lub **System element**. W najnowszej wersji narzędzia ReDSeeDS nie jest już konieczne przypisywanie znaczenia aktora lub elementu systemu. Zostało to zilustrowane na rysunku 13.

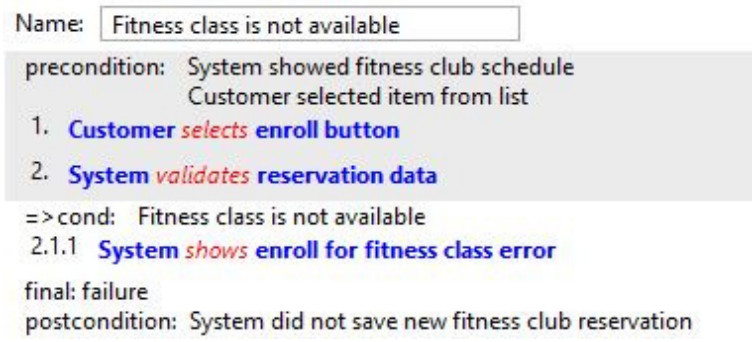


Rysunek 13: Oznaczanie aktora lub elementu systemu

Przykład kilku poprawnie napisanych scenariuszy przypadków użycia przedstawiono na rysunkach 14 oraz 15.



Rysunek 14: Przykładowy scenariusz główny



Rysunek 15: Przykładowy scenariusz alternatywny

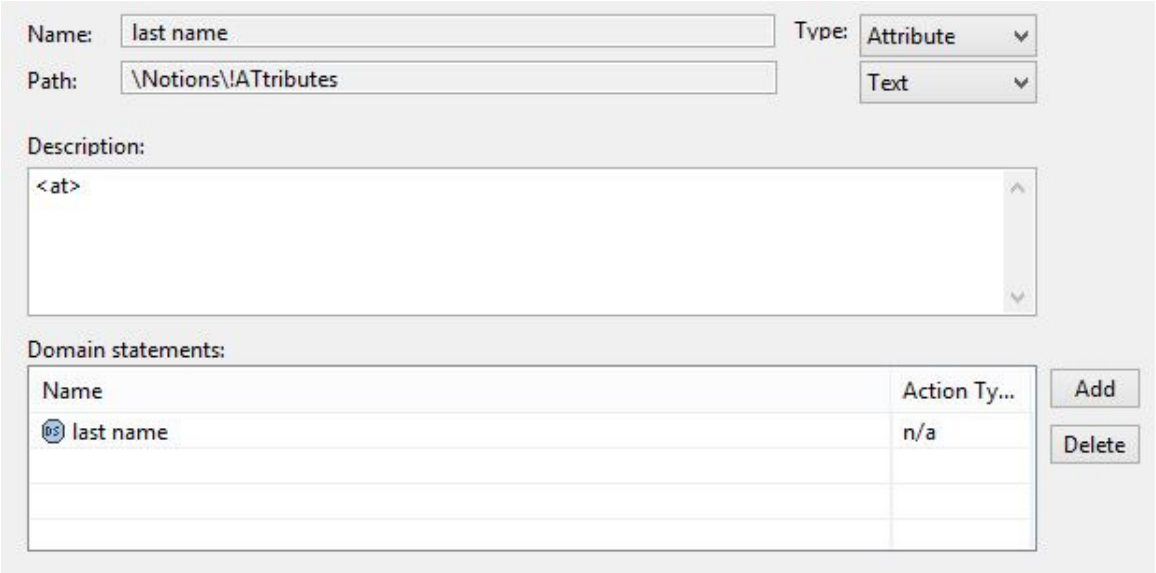
2.3. Zarządzanie pojęciami

Gdy użytkownik dodał już nowe pojęcia występujące w scenariuszach przypadków użycia, przychodzi pora na odpowiednie ich uporządkowanie. Zanim wygenerowany zostanie kod aplikacji, należy oddzielić pojęcia będące elementami interfejsu użytkownika (np. okna, komunikaty, formularze) od atrybutów i pojęć biznesowych. W tym celu użytkownik powinien utworzyć nowy pakiet pojęć poprzez wybór opcji **New → Notions Package**. Nowoutworzony pakiet z elementami interfejsu użytkownika powinien zaczynać się od nazwy **!UI** czyli np. **!UIElements**. Dla przeniesionych do tego pakietu pojęć wygenerowane zostaną charakterystyczne klasy wraz z importami niezbędnych bibliotek i specyficznymi konstrukcjami elementów interfejsu użytkownika.

Podobnie sytuacja wygląda dla pakietu zawierającego atrybuty innych pojęć. Konieczne jest utworzenie pakietu, którego nazwa zaczyna się od **!AT** czyli np. **!ATtributes**. Różni się on wcześniej utworzonego pakietu **!UIElements** tym, że dla pojęć zawartych w pakiecie **!ATtributes** nie zostaną wygenerowane żadne osobne klasy.

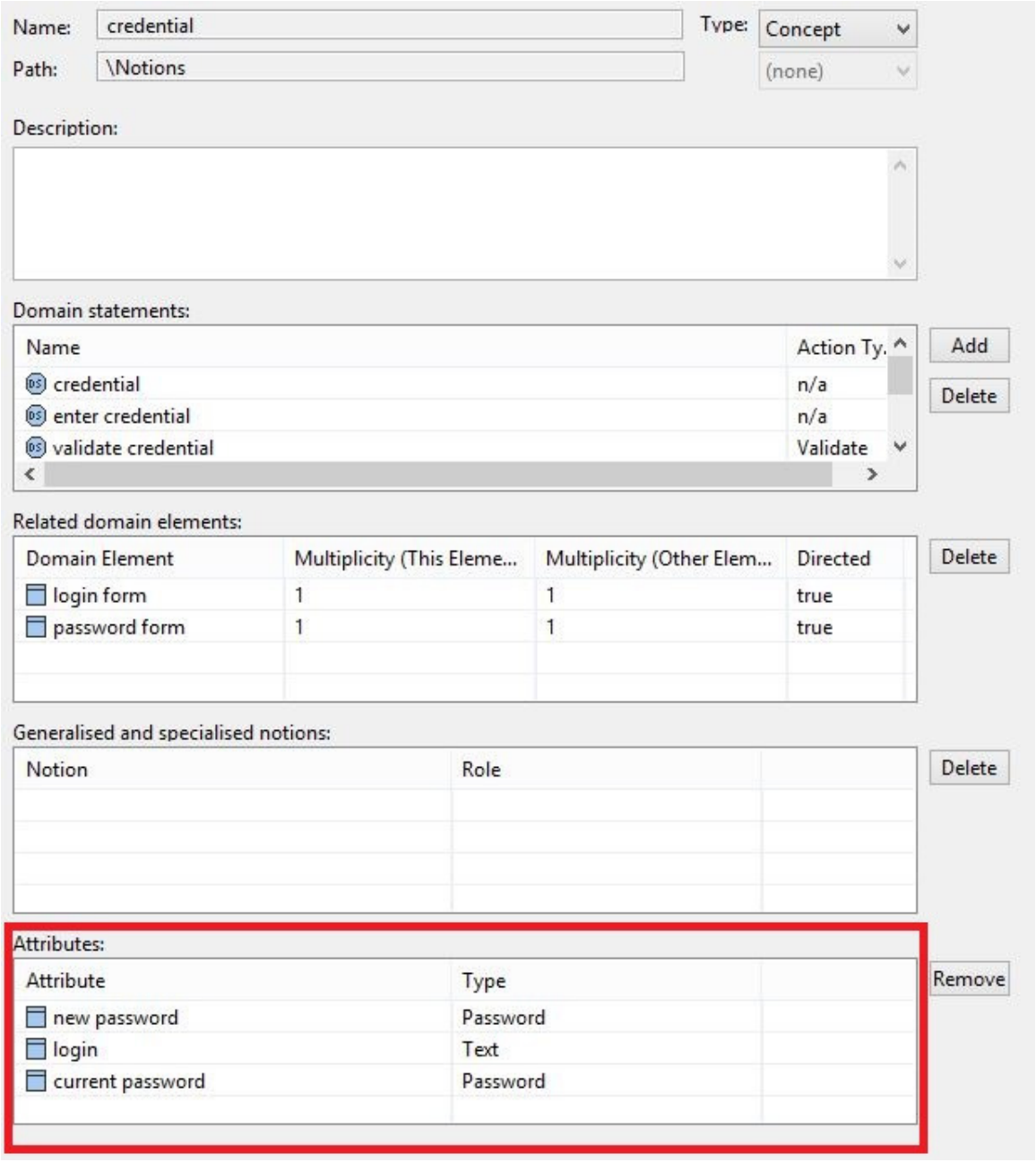
Warto wspomnieć, że użytkownik tworząc specyfikację wymagań powinien wiedzieć, które z pojęć powinny zostać wygenerowane jako osobne klasy, a które nie. Pojęcia, które nie powinny stać się w kodzie osobnymi klasami należy umieszczać w pakietach, których nazwy rozpoczynają się od znaku „**!**”, z wyjątkiem członów !UI oraz !AT. Przykładem takiej sytuacji są pojęcia odnoszące się do przycisków, które nie są osobnymi klasami lecz elementami okien. W związku z tym zaleca się aby użytkownik utworzył w pakiecie !UI podpakiet **!Buttons**, do którego przeniesie wszystkie pojęcia będące przyciskami.

Aby pojęcia będące atrybutami spełniały swoje zadanie, należy wykonać 2 operacje. Po pierwsze atrybuty powinny być odpowiednio oznaczone. W tym celu po przeniesieniu ich o pakietu z członem !AT, użytkownik powinien wejść w edytor danego atrybutu, w sekcji **Type** wybrać opcję **Attribute**, poniżej wybrać typ atrybutu, a w sekcji **Description** wprowadzić znacznik **<at>**. Efekt takiego działania został przedstawiony na rysunku 16.



Rysunek 16: Oznaczanie atrybutu

Drugą wymaganą operacją jest dodanie jednego pojęcia do drugiego jako atrybut. Aby to zrobić, należy w drzewie projektu przeciągnąć jedno pojęcie (to, które docelowo ma stać się atrybutem) na drugie, a w oknie wyboru, które się wtedy pojawi wybrać opcję **Add as attribute**. Wtedy w szczegółach drugiego pojęcia w sekcji **Attributes** pojawi się lista przypisanych do niego atrybutów. Przykład takiej listy przedstawiono na rysunku 17.

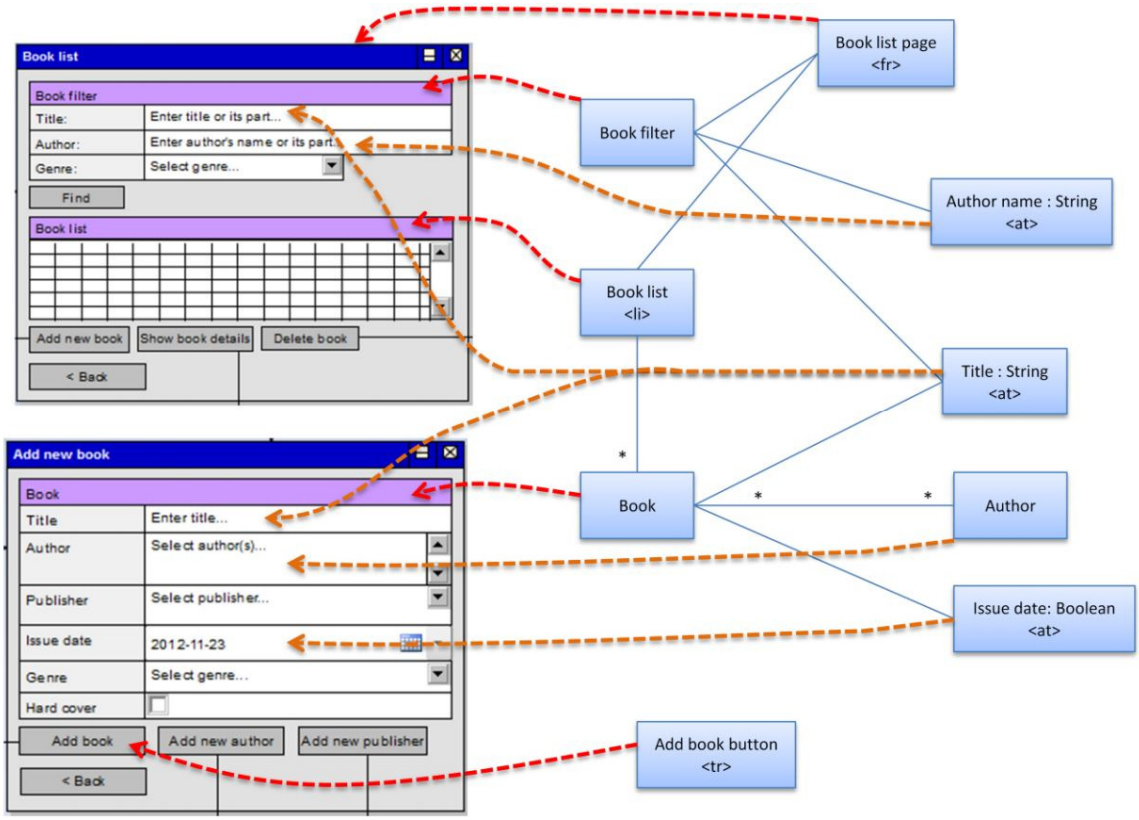


Rysunek 17: Lista przypisanych atrybutów do pojęcia „Credential”

Aby zdefiniowane pojęcia spełniały swoją rolę, podobnie jak w przypadku atrybutów, muszą mieć przypisany odpowiedni znacznik. Lista wymaganych znaczników i ich przeznaczenie została przedstawiona w poniższej tabeli.

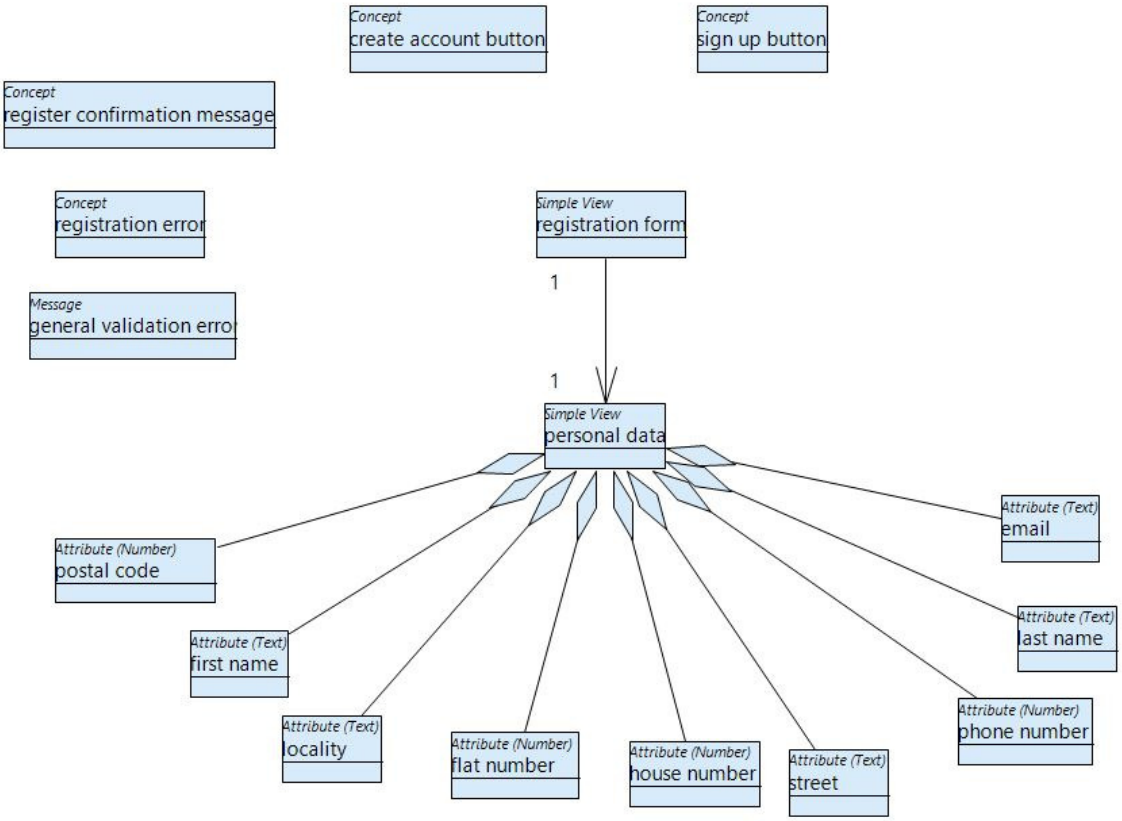
Rodzaj pojęcia	Oznaczenie	Uwagi
Pojęcie biznesowe	brak	Pojęcia biznesowe mogą być powiązane z innymi pojęciami biznesowymi, atrybutami, listami oraz oknami.
Lista (List)		Lista musi być powiązana tylko z jednym pojęciem biznesowym – elementem listy. Krotność po stronie elementu listy musi być ustawiona na „*''. Lista może być powiązana z pojęciem typu okno.
Atrybut (Attribute)	<at>	Atrybut musi mieć zdefiniowany typ prosty (String, Integer, Boolean, Date, itp.). Atrybut może być powiązany z dowolną liczbą pojęć biznesowych.
Okno/Strona (Frame)	<fr>	Pojęcie typu okno może być powiązane z dowolną ilością pojęć biznesowych lub list.
Przycisk (Trigger)	<tr>	Przyciski mogą, ale nie muszą być powiązane z oknami.
Komunikat (Message)	<ms> Treść kom. </ms>	

Na rysunku 18 przedstawiono przykład mapowań wymienionych w tabeli rodzajów pojęć na elementy interfejsu użytkownika.



Rysunek 18: Przykłady mapowań pojęć na elementy UI

Po uporządkowaniu pojęć w pakiecie Notions, użytkownik może stworzyć diagram pojęć, na którym w sposób przejrzysty przedstawi hierarchie i zależności panujące w słowniku pojęć. W tym celu powinien z menu kontekstowego pakietu Notions wybrać opcję **New → Notions Diagram**. Następnie użytkownik powinien przeciągnąć z drzewa projektu na diagram pojęcia, które chce na nim przedstawić. Należy pamiętać, aby nie umieszczać na jednym diagramie zbyt dużej liczby pojęć, gdyż zmniejsza to czytelność diagramu. Optymalnym rozwiązaniem jest relacja 1 przypadek użycia – 1 diagram pojęć. Przykładowy diagram pojęć dla przypadku użycia „Create profile" został przedstawiony na rysunku 19.

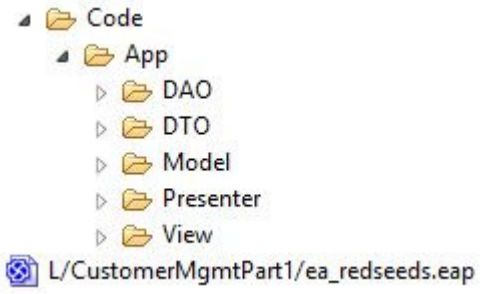


Rysunek 19: Diagram pojęć dla przypadku użycia „Create profile”

2.4. Walidacja i generowanie kodu

Zanim użytkownik przystąpi do procesu generowania kodu w języku Java, powinien najpierw zweryfikować, czy stworzony przez niego model specyfikacji wymagań nie zawiera żadnych błędów. W tym celu powinien otworzyć menu kontekstowe pakietu **Requirements Specification** i wybrać opcję **Validate**. Spowoduje to sprawdzenie przez narzędzie ReDSeeDS, czy stworzony model jest poprawny. Jeśli w raporcie końcowym znajdują się błędy, nie będzie możliwa generacja kodu, dopóki nie zostaną one naprawione.

W przypadku gdy nie stwierdzono żadnych błędów, ale pojawiły się tylko jakieś ostrzeżenia, użytkownik może wygenerować kod aplikacji. Aby tego dokonać musi ponownie otworzyć menu kontekstowe pakietu **Requirements Specification** i wybrać opcję **Generate Implementation**. Następuje wówczas proces tłumaczenia zależności zachodzących w modelu na kod. W trakcie transformacji, wykorzystywany jest program Enterprise Architect, więc użytkownik musi pamiętać o wcześniejszym zainstalowaniu pełnej wersji tego narzędzia. Po pozytywnym zakończeniu procesu transformacji, który trwa kilka minut, ReDSeeDS wyświetla komunikat o zakończeniu generowania implementacji. W drzewie projektu pojawi się plik projektu programu Enterprise Architect oraz folder **Code**, w którym znajduje się wygenerowany kod z podziałem na warstwy zgodne z modelem MVP oraz obiekty DAO i DTO. Zostało to zilustrowane na rysunku 20.



Rysunek 20: Efekty generowania implementacji

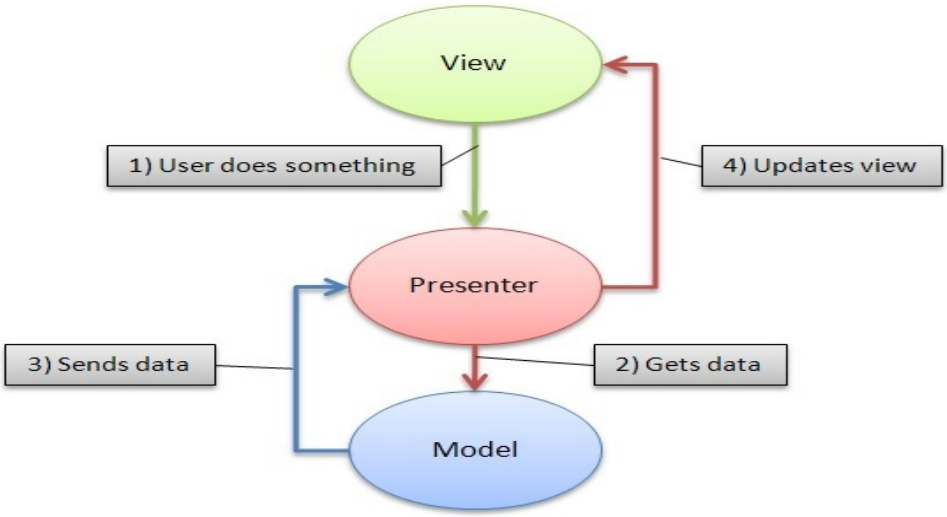
3. Praca nad wygenerowanym kodem

Niniejszy rozdział zawiera szczegółowy opis implementacji aplikacji klienta i managera MOSiR z wykorzystaniem kodu wygenerowanego przez narzędzie ReDSeeDS. Zostaną tu zaprezentowane sposoby generowania przez narzędzie ReDSeeDS istotnych fragmentów kodu. Opisane zostaną także szczegółowo zmiany jakie zostały wprowadzone w każdym z pakietów.

3.1. Przegląd wygenerowanego kodu aplikacji

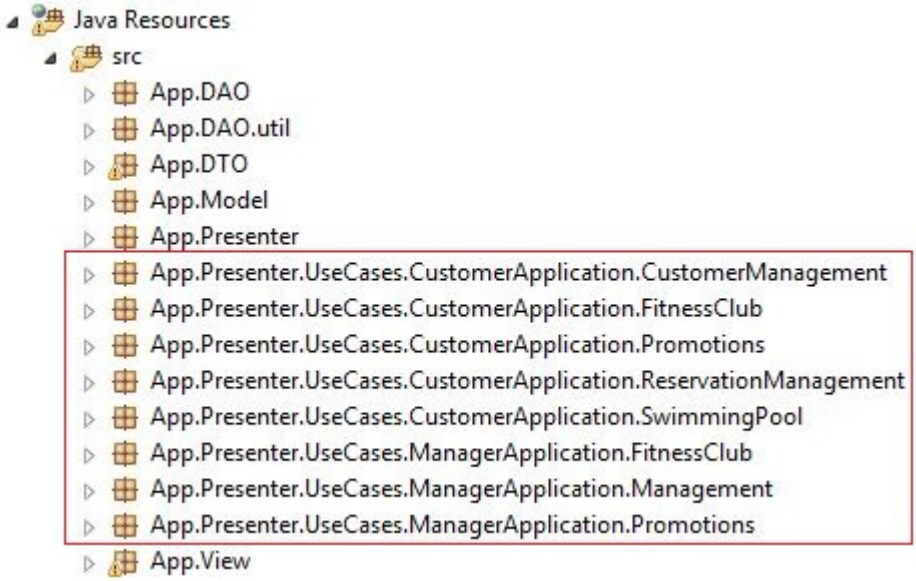
Po wygenerowaniu kodu przez narzędzie ReDSeeDS w obszarze roboczym powstał folder o nazwie „App”, w którym znalazły się interfejsy oraz klasy pogrupowane w odpowiednie pakiety. Transformacja wykorzystywana przez narzędzie ReDSeeDS generuje implementację w języku programowania Java w oparciu o architekturę Model – Widok - Prezenter. MVP jest popularnym wzorcem projektowym służącym do oddzielenia warstwy logiki aplikacji od warstwy prezentacji. Dzięki jego wykorzystaniu odseparowana jest od siebie prezentacja i przetwarzanie danych. Przekłada się to na większą przejrzystość kodu a także łatwiejsze jego utrzymanie. Aplikacje w MVP są również bardziej podatne na zmiany technologii.

Każda z trzech warstw wzorca MVP pełni z góry określone zadanie, co jest zobrazowane na rysunku 21. Warstwa widoku umożliwia wyświetlanie w interfejsie użytkownika danych pobranych przez model, co dzieje się za pośrednictwem warstwy prezenter. Widok obsługuje także przekazywanie danych wprowadzonych przez użytkownika do prezentera, co odbywa się za pomocą zdarzeń. Głównym zadaniem prezentera jest aktualizacja modelu i widoku. Prezenter może pobierać dane z modelu i przekazywać je do widoku, ale działa także w drugą stronę, czyli pobiera dane z widoku i przekazuje je do modelu. Można więc powiedzieć, że jest pośrednikiem między warstwą widoku i modelu, które nie komunikują się ze sobą i co więcej nie wiedzą o swoim istnieniu. Ostatnia warstwa, czyli model, odpowiedzialna jest za logikę aplikacji. Realizuje ona zarządzanie danymi, na których operuje aplikacja.



Rysunek 21: Graficzna prezentacja wzorca MVP

W celu dalszego zarządzania wygenerowanym kodem w środowisku Eclipse został utworzony pusty projekt aplikacji webowej. Następnie folder „App", z całą swoją zawartością, został skopiowany do folderu „src" znajdującego się w obszarze roboczym środowiska Eclipse. Na rysunku 22 zaprezentowany został widok projektu po imporcie kodu.

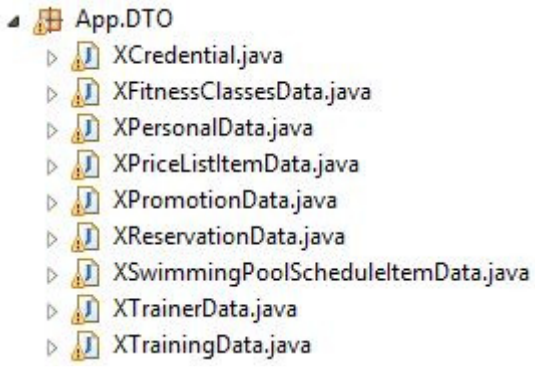


Rysunek 22: Struktura pakietów po wygenerowaniu kodu

Poszczególne pakiety odpowiadają omówionym wcześniej warstwom wzorca MVP. Warto zwrócić uwagę, że wersja transformacji, która została wykorzystana do generowania kodu niepotrzebnie duplikuje ten sam kod dla warstwy prezenter. Cała niezbędna implementacja dla tej warstwy znajduje się w pakiecie „App.Presenter". Warto tutaj zaznaczyć, że właśnie ta lokalizacja jest domyślnie wykorzystywana przez warstwę widoku. Pozostałe pakiety, oznaczone na rysunku czerwoną ramką, zawierają ten sam kod w rozbiciu na poszczególne moduły aplikacji. Pakiety te nie zostały wykorzystane w dalszej implementacji. Oprócz pakietów odpowiadających warstwom wzorca MVP powstały także pakiety „App.DTO", „App.DAO" oraz „App.DAO.util", które zostaną omówione w kolejnym podrozdziale.

3.2. Obiekty DTO i DAO

Zanim zostaną szczegółowo omówione kluczowe pakiety wzorca MVP warto zapoznać się z pakietami dodatkowymi. Są nimi pakiety „App.DTO", „App.DAO" oraz „App.DAO.util". Jak już wspomniano wcześniej, pomiędzy poszczególnymi warstwami przepływają różne dane. Aby łatwiej było tymi danymi operować, grupuje się je w specjalne klasy zwane Data Transfer Object (DTO). Narzędzie ReDSeeDS generuje obiekty DTO z pojęć, które nie są elementami ekranowymi, przyciskami ani atrybutami. Wszystkie klasy reprezentujące DTO posiadają w nazwie przedrostek „X". Są one umieszczone w oddzielnym pakiecie „App.DTO". Na rysunku 23 zaprezentowana została zawartość tego pakietu.



Rysunek 23: Zawartość pakietu „App.DTO"

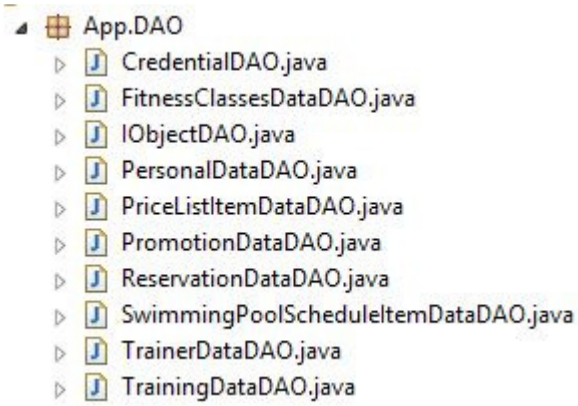
Jeżeli w fazie definiowania wymagań w narzędziu ReDSeeDS elementy biznesowe zostały uzupełnione o odpowiednie atrybuty, wówczas powstała klasa będzie zawierała te atrybuty, domyślny konstruktor a także standardowe gettery i settery dla każdego atrybutu. Następnie zaprezentowano sposób generowania kodu dla obiektów DTO.

```
public class XTrainerData {
	private String description;
	private String email;
	private String specialization;
	private String trainerFullName;
	private int xTrainerDataID;
	private int yearsOfExperience;
	public XTrainerData(){
	}
	...
	public String getDescription(){
		return description;
	}
	...
	public void setDescription(String description){
		this.description = description;
	}
}
```

Fragmenty kodu pochodzą z klasy **XTrainerData**. Pierwszy fragment prezentuje wszystkie atrybuty, jakie ta klasa zawiera oraz standardowy konstruktor. Kolejne fragmenty

prezentują sposób generowania przez ReDSeeDS getterów i seterów dla atrybutu „description” (metody *getDescription()* oraz *setDescription()*). Analogicznie metody wyglądają dla pozostałych atrybutów. Wcześniejsze zdefiniowanie atrybutów dla każdego pojęcia biznesowego na poziomie analizy wymagań w narzędziu ReDSeeDS znacznie skraca czas późniejszej pracy z kodem, ponieważ klasy DTO zostaną w całości uzupełnione niezbędnymi metodami i są od razu gotowe do użycia bez konieczności wprowadzania większych zmian.

Kolejnymi przydatnymi elementami, które generuje narzędzie ReDSeeDS, są obiekty DAO czyli Data Access Objects. Komponenty DAO zapewniają interfejs umożliwiający komunikację aplikacji z zewnętrznym źródłem danych jakim jest w tym przypadku baza danych H2. Służą one do odseparowania dostępu do danych od logiki biznesowej i warstwy prezentacji. Wszystkie klasy reprezentujące obiekty DAO zostały przez narzędzie ReDSeeDS wygenerowane z dokładnie tych samych pojęć, co obiekty DTO i w nazwie posiadają przyrostek „DAO” Zawartość pakietu „App.DAO” została zaprezentowana na rysunku 24.



Rysunek 24: Zawartość pakietu „App.DAO”

Wszystkie klasy widoczne na rysunku 24 implementują interfejs **IObjectDAO**. Interfejs ten zawiera deklarację kluczowych operacji zarządzających informacjami zgromadzonymi w bazie danych. Metody te umożliwiają zapis, odczyt, modyfikację oraz usuwanie elementów. Niestety próby wykorzystania wygenerowanych metod w praktyce kończyły się w większości niepowodzeniem. Do poprawnego ich działania konieczne było dokonanie wielu zmian, dlatego w rezultacie podjęto decyzję o tylko częściowym ich wykorzystaniu. Większość metod realizujących operacje na bazie danych została, dla oszczędzenia czasu, napisana samodzielnie. Następujący fragment kodu pochodzi z klasy **PromotionDataDAO** i przedstawia metodę zapisującą nową promocję w bazie.

```
public int savePromotion(XPromotionData p) {
              int res = 0;
              Statement stm = null;
              try {
                   stm = this.connection.createStatement();
                   int id = getRowCount("PROMOTION");

                   String sql = "INSERT INTO PUBLIC.PROMOTION "
                   +"(ID, NAME, DESCRIPTION, DISCOUNT, EXPIRATION_DATE, TYPE)"
                   +" VALUES ("+ id +", '"+ p.getPromotionName() +"','"
                   + p.getDescription() +"',"+ p.getDiscount() +" ,'"
                   + p.getExpirationDate() +"','"+ p.getType() +"');";

                   res = stm.executeUpdate(sql);
```

```
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                if (stm != null) {
                    try {
                        stm.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
            }
            return res;
        }
```

Ostatnim pakietem wartym omówienia, który generuje narzędzie ReDSeeDS jest pakiet „App.DAO.utils". Cała jego zawartość nie zostanie zaprezentowana. Istotną klasą wchodzącą w skład tego pakietu jest klasa **ConnectionFactory**, której fragmenty prezentuje następujący kod.

```
public class ConnectionFactory {

        private String dbDriver = "org.h2.Driver";
        private static ConnectionFactory factory = null;
        private String password = "";
        private Properties prop = new Properties();
        private   String   url   =   "jdbc:h2:file:C:\\Studia\\Sem   7\\Praca   dyplo-
mowa\\Baza\\mosir.db";
        private String user = "sa";
...
public ConnectionFactory(){
            try {
                Class.forName(dbDriver);
                prop.setProperty("user", user);
                prop.put("password", password);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
...
        public Connection getConnection(){
            Connection conn = null;
            try {
                conn = DriverManager.getConnection(url, prop);
            } catch (SQLException e) {
                e.printStackTrace();
            }
            return conn;
        }
```

Wśród atrybutów klasy istotny jest parametr „dbDriver", który wskazuje nazwę sterownika dla bazy danych H2. Parametr „url" wskazuje ścieżkę do katalogu, gdzie znajduje się plik z bazą danych. Domyślnie jest to adres widoczny na czyli katalog domowy użytkownika z nazwą rozpoczynającą się od „test". Metoda *getConnection()* odpowiada za nawiązanie połączenia ze wskazanym adresem URL.

3.3. Warstwa widoku

Pakietem odpowiadającym za interakcję użytkownika z aplikacją a także prezentację informacji jest pakiet „App.View”. Jak było już napisane wcześniej ReDSeeDS generuje klasy odpowiedzialne za interfejs użytkownika wykorzystując technologię Echo3. Wszystkie klasy warstwy widoku posiadają w nazwie przedrostek „V”. Powstały na podstawie pojęć z pakietów **!UIPages**, **!UIForms** oraz **!UIMessages**. W całej aplikacji istnieje wiele komponentów interfejsu użytkownika, takie jak strony, formularze oraz okna komunikatów. W związku z tym pakiet „App.View” jest bardzo obszerny. Zawiera on również interfejs **IView** oraz klasy **EchoApp** i **EchoServlet**. **IView** zawiera deklaracje operacji odpowiedzialnych za wyświetlanie poszczególnych komponentów. Klasa **EchoApp** implementuje interfejs **IView**. Implementacja została w całości wygenerowana przez narzędzie ReDSeeDS i nie wymaga większych zmian. Następujący fragment kodu prezentuje implementację metody wyświetlającej stronę główną.

```
public void showCustomerApplicationPage(){

              if (currentPage != null)
                  pageStack.push(currentPage);
              VCustomerApplicationPage page = new VCustomerApplicationPage();
              page.init(presenter);
              show(page);

       }
```

Analogicznie wyglądają metody wyświetlające pozostałe komponenty aplikacji MOSiR.

Strony odpowiadają przede wszystkim za prezentowanie użytkownikowi danych pobranych z modelu, a także umożliwiają nawigację za pomocą umieszczonych na nich przyciskach oraz zdefiniowanych zdarzeniach. Przykładowo klasa **VHomePage** reprezentuje stronę startową aplikacji. Z jej poziomu można zadecydować za pomocą przycisków, o przejściu do podstrony z aplikacją klienta bądź managera. Następujący fragment wygenerowanego kodu prezentuje sposób umieszczenia przycisków na stronie i zdefiniowanie dla nich zdarzeń.

```
public class VHomePage extends ContentPane implements ActionListener {

       private MutableStyle buttonStyle;
       private Column column;
       private IPresenter presenter;
       private Button customerApplicationButton;
       private Button managerApplicationButton;
...
      public void actionPerformed(ActionEvent e){

              if (e.getActionCommand().equals("customerApplicationButton")) {
                  presenter.SelectsCustomerApplicationButton();
              }

              if (e.getActionCommand().equals("managerApplicationButton")) {
                  presenter.SelectsManagerApplicationButton();
              }
       }
...
        private void addContent(){

              customerApplicationButton = new Button("Customer application");
              customerApplicationButton.setStyle(buttonStyle);
```

```
                customerApplicationButton.
                        setActionCommand("customerApplicationButton”);
                customerApplicationButton.addActionListener(this);
                column.add(customerApplicationButton);

                managerApplicationButton = new Button("Manager application");
                managerApplicationButton.setStyle(buttonStyle);
                managerApplicationButton.setActionCommand("managerApplicationButton");
                managerApplicationButton.addActionListener(this);
                column.add(managerApplicationButton);

        }
```

W metodzie *addContent()* na stronie umieszczane są dwa przyciski. Przycisk „customerApplicationButton” ma za zadanie przenieść użytkownika do strony klienta, natomiast przycisk „managerApplicationButton” – do strony managera. Metoda *actionPerformed(ActionEvent e)* definiuje zdarzenia jakie zajdą po kliknięciu na każdy z przycisków. W obu przypadkach po kliknięciu na przycisk wywołana zostanie odpowiednia metoda z warstwy prezenter, w tym przypadku wyświetlająca żądaną podstronę, czyli podstronę dla klienta bądź dla managera. Kod w ten sposób wygenerowany jest w pełni funkcjonalny i nie wymaga już żadnej ingerencji ze strony programisty.

Specjalnym rodzajem stron są formularze. Umożliwiają one użytkownikowi aplikacji wprowadzanie danych, które następnie są przekazywane do prezentera. Cała struktura formularza oraz jego pola powstają podczas generowania kodu. Programista nie musi się martwić o to w jaki sposób rozmieścić poszczególne etykiety i pola tekstowe, ponieważ ReDSeeDS zrobi to automatycznie. Sposób w jaki ReDSeeDS generuje formularze zostanie zaprezentowany na przykładzie formularza rejestracji.

```
public class VRegistrationForm extends ContentPane implements ActionListener {
        private Button _closeButton;
        private MutableStyle buttonStyle;
        private Column column;
        private Button createAccountButton;
        private Label emailLabel;
        private TextField emailTextField;
        private Label firstNameLabel;
        private TextField firstNameTextField;
        private Label flatNumberLabel;
        private TextField flatNumberTextField;
        private Grid grid;
        private GridLayoutData gridLayout;
        private Label houseNumberLabel;
        private TextField houseNumberTextField;
        private Label lastNameLabel;
        private TextField lastNameTextField;
        private Label localityLabel;
        private TextField localityTextField;
        private Label newPasswordLabel;
        private TextField newPasswordTextField;
        private Label personalDataHeadingLabel;
        private Label phoneNumberLabel;
        private TextField phoneNumberTextField;
        private Label postalCodeLabel;
        private TextField postalCodeTextField;
        private IPresenter presenter;
        private Label streetLabel;
        private TextField streetTextField;
        private Label confirmNewPasswordLabel;
        private TextField confirmNewPasswordTextField;
...
```

Powyższy kod przedstawia fragment klasy **VRegistrationForm**, zawierający listę atrybutów. Warto zwrócić uwagę, że dla każdego atrybutu zdefiniowanego w słowniku pojęć, ReDSeeDS generuje dwa elementy: etykietę (Label) oraz pole tekstowe (TextField) lub kalendarz (Calendar) w przypadku atrybutu o typie daty.

```
/* Add controls for personal data notion */
// Add notion heading spanning two columns
personalDataHeadingLabel = new Label("personal data");
gridLayout = new GridLayoutData();
gridLayout.setColumnSpan(2);
gridLayout.setBackground(new Color(105, 89, 205));
personalDataHeadingLabel.setLayoutData(gridLayout);
personalDataHeadingLabel.setFont(new Font(Font.HELVETICA, Font.ITALIC, new Ex-
tent(15)));
personalDataHeadingLabel.setForeground(Color.WHITE);
grid.add(personalDataHeadingLabel);

//Add control for notion attribute
firstNameLabel = new Label("first name:  ");
gridLayout = new GridLayoutData();
gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
firstNameLabel.setLayoutData(gridLayout);
grid.add(firstNameLabel);
firstNameTextField = new TextField();
firstNameTextField.setWidth(new Extent(75, Extent.PERCENT));
grid.add(firstNameTextField);
```

Fragment metody *addContent()* przedstawia sposób umieszczenia nagłówka formularza oraz etykiety i pola tekstowego na stronie. Jak widać wszystkie elementy są zwykłymi polami tekstowymi, jednak w przypadku atrybutu „password" pożądane jest zastępowanie wprowadzanych przez użytkownika znaków kropkami. W tym celu należy dokonać drobnej modyfikacji a dokładnie zmienić typ pola z TextField na PasswordField, co przedstawia następujący kod.

```
//Add control for notion attribute
newPasswordLabel = new Label("password:  ");
gridLayout = new GridLayoutData();
gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
newPasswordLabel.setLayoutData(gridLayout);
grid.add(newPasswordLabel);
newPasswordTextField = new PasswordField();
newPasswordTextField.setWidth(new Extent(75, Extent.PERCENT));
grid.add(newPasswordTextField);
```

Ostatnim etapem jest zebranie z formularza danych wprowadzonych przez użytkownika i przekazanie ich do prezentera. Dzieje się to po kliknięciu na przycisk „Create account". Standardowo obsługa zdarzenia kliknięcia na przycisk wygląda tak jak pokazuje następujący kod.

```
public void actionPerformed(ActionEvent e){
...

    if (e.getActionCommand().equals("createAccountButton")) {
                presenter.SelectsCreateAccountButton();
     }
}
```

Narzędzie ReDSeeDS nie generuje kodu odpowiedzialnego za pobieranie danych z formularza i przekazywanie ich do prezentera, dlatego programista musi zrobić to ręcznie.

Następnie przedstawiono fragment metody *actionPerformed()* uzupełnioną o niezbędny do tego celu kod.

```
public void actionPerformed(ActionEvent e){
...

    if (e.getActionCommand().equals("createAccountButton")) {
       boolean is_correct = true;
       is_correct = validateRegistrationData();
       if(is_correct){
       XPersonalData rd = new XPersonalData();
       rd.setFirstName(firstNameTextField.getText());
       rd.setLastName(lastNameTextField.getText());
       rd.setStreet(streetTextField.getText());
       rd.setHouseNumber(Integer.parseInt(houseNumberTextField.getText()));
       if (!flatNumberTextField.getText().equals("")){
              rd.setFlatNumber(Integer.parseInt(flatNumberTextField.getText()));
       }
       rd.setPostalCode(Integer.parseInt(postalCodeTextField.getText()));
       rd.setLocality(localityTextField.getText());
      rd.setPhoneNumber(Integer.parseInt(phoneNumberTextField.getText()));
       XCredential c = new XCredential();
       c.setLogin(emailTextField.getText());
       c.setNewPassword(newPasswordTextField.getText());
       c.setCurrentPassword(newPasswordTextField.getText());
       presenter.SelectsCreateAccountButton(rd, c);
       }else{
              presenter.wrongData();
       }
}
}
```

Dane użytkownika zaczytywane są do dwóch obiektów DTO: **XPersonalData** oraz **XCredential**. Warto tutaj wspomnieć, że narzędzie ReDSeeDS generuje wszystkie metody walidujące poprawność danych na poziomie modelu. Jednak z uwagi na to, że w wielu sytuacjach walidacja sprowadza się do sprawdzenia czy dane wprowadzone przez użytkownika są kompletne i w odpowiednim formacie, zdecydowano o przeniesieniu walidacji do warstwy widoku. W przykładzie zaprezentowanym powyżej za walidację odpowiada prywatna metoda *validateRegistrationData()*. W zależności od jej wyniku wywoływana jest odpowiednia metoda z prezentera.

Rysunek 25: Formularz rejestracji

Rysunek 25 pokazuje jak prezentuje się formularz po uzupełnieniu go przykładowymi danymi.

Formularze nie są wykorzystywane wyłącznie do wprowadzania danych przez użytkownika. Ich innym zastosowaniem jest prezentacja danych pobranych przez model. Za przykład może posłużyć tutaj klasa **VPromotionDetailsPage**, która służy do prezentacji szczegółowych danych promocji. Kod formularza, w którym zostaną umieszczone dane został w całości wygenerowany przez narzędzie ReDSeeDS w sposób analogiczny jak w przypadku klasy **VRegistrationForm**. Aby możliwe było zaprezentowanie w takim formularzu danych, konieczne były zmiany w kodzie. Pierwszym krokiem była zmiana domyślnie pustego konstruktora klasy **VPromotioDetailsPage** na konstruktor z parametrem, co pokazuje następujący fragment kodu.

```
public class VPromotionDetailsPage extends ContentPane implements ActionListener {

...
	private XPromotionData promotion;


	public VPromotionDetailsPage(XPromotionData p){
		this.promotion = p;
	}
}
```

Należy pamiętać, że zmiana taka pociąga za sobą także zmianę w interfejsie **IView** oraz w klasie **EchoApp**. Następnie konieczne było wprowadzenie danych z instancji obiektu **XPromotionData** do odpowiednich pól tekstowych w formularzu i zablokowanie edycji tych pól, co w przypadku atrybutu „promotion name" wygląda następująco (linie oznaczone na żółto).[8]

```
//Add control for notion attribute
promotionNameLabel = new Label("promotion name:  ");
gridLayout = new GridLayoutData();
gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
promotionNameLabel.setLayoutData(gridLayout);
grid.add(promotionNameLabel);
promotionNameTextField = new TextField();
promotionNameTextField.setText(promotion.getPromotionName());
promotionNameTextField.setEnabled(false);
promotionNameTextField.setWidth(new Extent(75, Extent.PERCENT));
grid.add(promotionNameTextField);
```

Po uruchomieniu aplikacji, ekran z przykładowymi szczegółami promocji prezentuje się tak jak na rysunku 26.

Rysunek 26: Ekran szczegółów promocji

Warto także wspomnieć o stronach prezentujących dane w postaci list. W przypadku aplikacji MOSiR listy takie zostały wykorzystane do prezentacji grafiku pływalni i klubu fitness a także do promocji, treningów, trenerów i rezerwacji. Z otrzymanej dokumentacji narzędzia ReDSeeDS wynika, że posiada on mechanizmy generowania kodu dla list. Niestety, po licznych próbach nie udało się takiego kodu wygenerować, co świadczy najprawdopodobniej o posiadaniu wadliwej wersji transformaty (problem z generowaniem list został szerzej opisany w rozdziale 5). W związku z tym, zaszła konieczność dopisania potrzebnego kodu ręcznie. Pierwszym krokiem było utworzenie klasy rozszerzającej AbstractTableModel.[8] W przypadku listy rezerwacji powstała więc klasa **ReservationsListTableModel**, której fragment przedstawia następujący kod.

```
public class ReservationsListTableModel extends AbstractTableModel{
	private List<XReservationData> rows = new ArrayList<XReservationData>();

	public ReservationsListTableModel(List<XReservationData> list){
		this.rows = list;
	}
...
	public int getColumnCount(){
		return 6;

	}

	public String getColumnName(int column){
		switch (column) {
		case 0: return "ID";
		case 1: return "Day";
		case 2: return "Time";
		case 3: return "Type";
		case 4: return "Duration";
		case 5: return "Training name";
```

```
                default: return null;
                }
        }

        public Object getValueAt(int column, int row){
                XReservationData tableRow = rows.get(row);
                switch (column) {
                case 0: return tableRow.getID();
                case 1: return tableRow.getDay();
                case 2: return tableRow.getTimeFrom();
                case 3: return tableRow.getType();
                case 4: return tableRow.getDuration();
                case 5: return tableRow.getTrainingName();
                default: return null;
                }
        }
```

Klasa posiada konstruktor z parametrem **List<XReservationData>** oraz trzy kluczowe metody. W metodzie *getColumnCount()* definiuje się ile kolumn będzie zawierała tabela. Z kolei metoda *getColumnName()* ustawia nazwy poszczególnym kolumnom. Metoda *getValueAt()* wstawia w odpowiednie kolumny tabeli wartości pobrane z listy. Kolejnym krokiem przy tworzeniu tabeli jest modyfikacja strony na której tabela ma zostać wyświetlona. W tym przypadku zmian wymaga klasa **VReservationsPage**. Sam kod strony został przez narzędzie ReDSeeDS wygenerowany poprawnie, jednak brakuje tam kilku elementów, które umożliwią wyświetlenie tabeli, a więc muszą one zostać uzupełnione przez programistę.

```
public class VReservationsPage extends ContentPane implements ActionListener {

        private Button _closeButton;
        private MutableStyle buttonStyle;
        private Column column;
        private Grid grid;
        private GridLayoutData gridLayout;
        private IPresenter presenter;
        private Table reservationsTable;
        private Button cancelReservationButton;
        private ReservationsListTableModel reservationsTableModel;
        private DefaultListSelectionModel reservationsSelectionModel;
        private List<XReservationData> reservations;

        public VReservationsPage(List<XReservationData> reservations){
                this.reservations = reservations;
        }
```

Powyżej zaprezentowano fragment klasy **VReservationsPage**. Atrybuty oznaczone na żółto, zostały dodane w celu umożliwienia wyświetlenia listy. Dodatkowo zmianie uległ konstruktor klasy analogicznie jak to miało miejsce w przypadku strony ze szczegółami promocji. Wśród nowo dodanych atrybutów znalazł się element typu ReservationsListTableModel, który został utworzony wcześniej. Element reservationsSelectionModel będzie umożliwiał dokonanie wyboru konkretnej rezerwacji z listy, a następnie przeprowadzenia na niej innych działań.

```
private void addContent(){

...

        reservationsTableModel = new ReservationsListTableModel(reservations);
        reservationsSelectionModel = new DefaultListSelectionModel();
        reservationsSelectionModel.
                setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        reservationsSelectionModel.setSelectedIndex(0, true);
        reservationsTable = new Table(reservationsTableModel);
        reservationsTable.setSelectionModel(reservationsSelectionModel);
        reservationsTable.setSelectionEnabled(true);
        reservationsTable.setSelectionBackground(Color.LIGHTGRAY);
        reservationsTable.setBorder(new Border(1, Color.BLACK, Border.STYLE_DOTTED));
        reservationsTable.setBackground(Color.WHITE);
        reservationsTable.setInsets(new Insets(5, 2));
        reservationsTable.setRolloverEnabled(true);
        reservationsTable.setRolloverBackground(Color.LIGHTGRAY);
        gridLayout = new GridLayoutData();
        gridLayout.setColumnSpan(2);
        reservationsTable.setLayoutData(gridLayout);
        grid.add(reservationsTable);
...

}
```

Ostatnim krokiem przy tworzeniu listy było umiejscowienie odpowiednich elementów w treści strony, ich zainicjowanie oraz sformatowanie, co realizuje przedstawiony powyżej fragment metody *addContent()*. Jak widać z powyższego opisu cała procedura ręcznej implementacji tabeli nie jest zbyt skomplikowana, jednak może zająć znaczną ilość czasu gdy jest wykonywana po raz pierwszy. Jeżeli transformata RSL to C generowałaby listy poprawnie, z pewnością zaoszczędziło by to programiście dużo czasu, ponieważ nie było by potrzeby od podstaw tworzyć klasy **ReservationsListTableModel**. Uniknięto by także wielu modyfikacji w klasie **VReservationsListPage**.

Rysunek 27: Lista rezerwacji klienta

Ostatnim elementem jaki zostanie omówiony w ramach warstwy widoku są okna komunikatów. Służą one do informowania użytkownika o powodzeniu lub błędzie wykonania określonej czynności. Pojedyncze okno reprezentuje klasa rozszerzająca klasę **WindowPane**. Są to elementy, które w zasadzie nie wymagają żadnych ulepszeń ze strony programisty pod warunkiem, że w narzędziu ReDSeeDS zostanie poprawnie zdefiniowany komunikat dla takiego okna.

```
public class VCancelReservationConfirmationMessage extends WindowPane implements
ActionListener {


        private Button _okButton;
        private MutableStyle buttonStyle;
        private Column column;
        private Label messageText;
        private IPresenter presenter;


        public VCancelReservationConfirmationMessage(){


        }
...
public void init(IPresenter presenter){


...


        // Add message content
        setTitle("Message");
        setModal(true);
        messageText = new Label("Reservation has been cancelled successfully.");
        column.add(messageText);
...
}
```

Kod zaprezentowany na powyżej przedstawia fragment klasy **VCancelReservationConfirmationMessage**, która informuje użytkownika o pomyślnym anulowaniu rezerwacji. Budowa tej klasy jest niezwykle prosta. Posiada ona treść komunikatu „messageText" oraz przycisk potwierdzający „_okButton". Fragment metody *init()* odpowiada za umiejscowienie tytułu oraz komunikatu tekstowego w oknie.

Rysunek 28: Okno komunikatu o pomyślnym anulowaniu rezerwacji

Ostatnim istotnym fragmentem tej klasy jest obsługa zdarzenia kliknięcia przycisku OK. Jak można się domyślić po jego kliknięciu zostanie zamknięte okno z komunikatem, jednak dodatkowo w większości przypadków pożądany jest powrót do strony poprzedniej. Aby to zrealizować uzupełniono obsługę zdarzenia o dodatkową linię kodu oznaczoną na żółto.

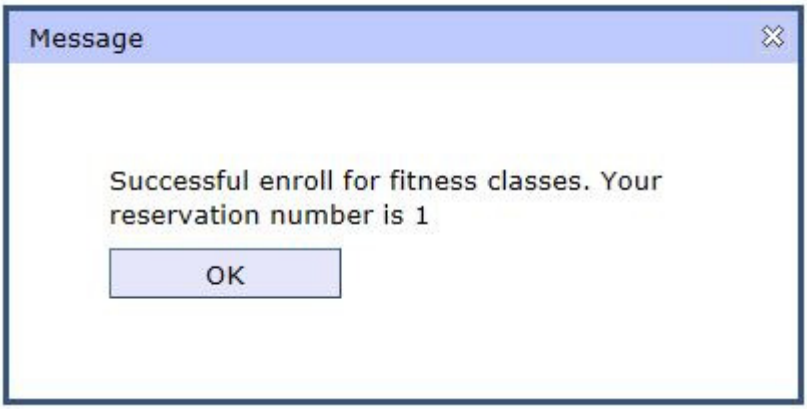
```
public void actionPerformed(ActionEvent e){


	if (e.getActionCommand().equals("_okButton")) {
		userClose();
		((EchoApp) ApplicationInstance.getActive()).closeCurrentPage();
	}
}
```

Domyślny konstruktor okna komunikatu jest konstruktorem bezparametrowym i w większości przypadków jest on wystarczający. Istnieją jednak sytuacje, gdy potrzebne jest pokazanie użytkownikowi czegoś więcej niż tylko informacji o sukcesie czy błędzie wykonanej operacji. Za przykład może tutaj posłużyć okno potwierdzenia rezerwacji, na którym oprócz samej informacji, że rezerwacja została utworzona, warto także umieścić numer rezerwacji.

```
public class VConfirmationToEnrollForFitnessClassesMessage extends WindowPane imple-
ments ActionListener {

	private Button _okButton;
	private MutableStyle buttonStyle;
	private Column column;
	private Label messageText;
	private IPresenter presenter;
	private int reservationID;

	public VConfirmationToEnrollForFitnessClassesMessage( int id){
		this.reservationID = id;
	}
```

Powyższy kod przedstawia fragment klasy **VConfirmationToEntollForFitnessClassesMessage**. Jak widać dodany został nowy parametr – „reservationID”, a także zmienio-

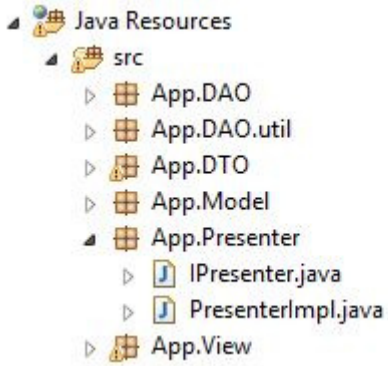
ny domyślny konstruktor, po to aby prezenter mógł przekazać do okna numer rezerwacji. Po zmianach i uruchomieniu aplikacji okno prezentuje się tak jak na Rysunku 36.



Rysunek 29: Okno komunikatu o pomyślnym zapisaniu się na zajęcia fitness

3.4. Prezenter

Warstwa prezentera jest zarządcą aplikacji. Pośredniczy ona w komunikacji pomiędzy warstwą modelu i widoku. Odpowiada za sterowanie zachowaniem systemu w ramach poszczególnych przypadków użycia. Zawartość pakietu „App.Presenter" przedstawia rysunek 30.



Rysunek 30: Zawartość pakietu 'App.Presenter'

Składa się on z dwóch elementów. **IPresenter** jest interfejsem zawierającym definicję wszystkich operacji warstwy prezenter. Klasa **PresenterImpl** jest klasą implementującą interfejs **IPresenter**.

```
public interface IPresenter {

	public void SelectsAddFitnessClassesButton();

	public void SelectsAddNewFitnessClassesButton();

	public void SelectsAddPromotionButton();

	public void SelectsAddTrainerButton();

	public void SelectsAddTrainingButton();

	public void SelectsCancelReservationButton();

	public void SelectsChangeButton();

	public void SelectsChangePasswordButton();

	public void SelectsChangePersonalDataButton();

	public void SelectsCreateAccountButton();
…
}
```

Powyższy kod przedstawia fragment interfejsu **IPresenter** bezpośrednio po wygenerowaniu przez narzędzie ReDSeeDS. Wszystkie metody reprezentują akcje, które mają miejsce po wyborze przez użytkownika systemu określonych przycisków.

```
public class PresenterImpl implements IPresenter {

	private IModel model;
	int res;
	private IView view;

	public PresenterImpl(){

	}

...

	public void SelectsAddFitnessClassesButton(){
		res = model.ValidatesFitnessClassesData();
		if ( res == 1 /*fitness classes data valid*/) {
		res = model.SavesFitnessClassesData();
		view.showNewFitnessClassesConfirmationMessage();
		} else if ( res == 2 /*day or time invalid*/) {
		view.showNewFitnessClassesError();
		}else if( res == 3 /*incorrect or missing data*/) {
		view.showGeneralValidationError();
		}
	}
...
}
```

Powyżej zaprezentowany został fragment klasy **PresenterImpl** bezpośrednio po wygenerowaniu kodu. Przedstawia on implementację metody *SelectAddFitnessClassesButton()*. Metoda ta jest wywoływana w trakcie przypadku użycia „Add fitness classes” (Dodanie

nowych zajęć fitness do grafiku), po wypełnieniu przez managera formularza i kliknięciu przycisku „Add fitness classes". Pierwszym krokiem jest wywołanie metody z modelu walidującej wprowadzone dane. Następnie na podstawie wyniku tej walidacji podejmowane są dalsze akcje. Jeżeli dane są poprawne – ponownie wywoływana jest metoda z modelu zapisująca dane oraz wyświetla się okno komunikatu o pomyślnym dodaniu zajęć fitness do grafiku. Jeżeli dane są niepoprawne – wyświetlane jest okno komunikatu o odpowiednim błędzie. Jak już zostało napisane, wszystkie proste walidacje danych zostały przeniesione na poziom widoku. Z kolei walidacje korzystające z zapytań do bazy danych przeprowadzane są na poziomie modelu, czyli dokładnie tak jak domyślnie generuje je ReDSeeDS. Warto zwrócić uwagę że wszystkie operacje z interfejsu **IPresenter** nie mają żadnych parametrów wejściowych. Uniemożliwia to przepływ danych z warstwy widoku do modelu, dlatego kluczową modyfikacją, jaka jest konieczna do dokonania przez programistę, jest zmiana deklaracji odpowiednich metod poprzez dodanie parametrów wejściowych.

Zagadnienie to zostanie omówione na przykładzie przypadku użycia „Create profile" (Utworzenie konta użytkownika). Po wprowadzeniu danych użytkownika do formularza dane te muszą zostać przekazane do prezentera a następnie do modelu, gdzie zostaną sprawdzone a następnie zapisane. Następnie przedstawiono metodę *SelectsCreateAccountButton()* uzupełnioną o niezbędny kod.

```
public void SelectsCreateAccountButton(XPersonalData rd, XCredential c){
	res = model.ValidatesRegistrationData(c);
	if ( res == 1 /*registration data is valid*/) {
		res = model.SavesPersonalData(rd, c);
		view.showRegisterConfirmationMessage();
	} else if ( res == 2 /*registration data is not valid*/) {
		view.showRegistrationError();
	}
}
```

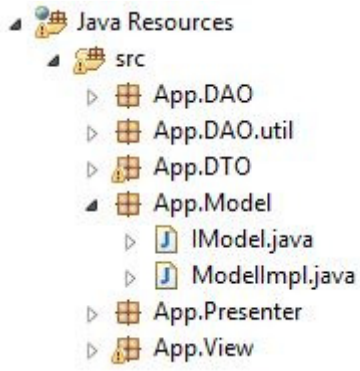
Na wejściu metody dodano dwa obiekty DTO: **XPersonalData** oraz **XCredential**, co umożliwi transport danych wprowadzonych przez użytkownika do warstwy modelu. Warto zaznaczyć, że metody warstwy prezentera, które przekazują dane do widoku nie wymagają w zasadzie żadnych zmian w parametrach wejściowych. Można to zaobserwować na przykładzie metody pobierającej i prezentującej użytkownikowi grafik pływalni.

```
public void SelectsSwimmingPoolScheduleButton(){

	List<XSwimmingPoolScheduleItemData> swimmingPoolSchedule = null;
	swimmingPoolSchedule = model.DownloadsSwimmingPoolSchedule();
	view.showSwimmingPoolSchedulePage(swimmingPoolSchedule);

}
```

3.5. Model

Warstwa modelu jest ostatnią warstwą jaka została do omówienia w ramach implementacji aplikacji MOSiR. Warstwa ta odzwierciedla wszystkie operacje jakie można wykonać

na obiektach przechowywanych w systemie. Podobnie jak w przypadku warstwy prezenter, struktura pakietu „App.Model" jest bardzo prosta co przedstawia Rysunek 38.



Rysunek 31: Zawartość pakietu „App.Model"

Interfejs **IModel** zawiera deklarację wszystkich operacji tej warstwy natomiast klasa **ModelImpl** implementuje interfejs **IModel**.

```
public interface IModel {

	public int ChangesCredentials();
...
	public int DeletesTrainerData();

	public int DeletesTrainingData();
...
	public int DownloadsPersonalData();

	public int DownloadsPromotionsList();
...

	public int SavesTrainingData();

	public int ValidatesCredentials();
...

}
```

Powyżej przedstawiono fragment interfejsu **IModel**. Metody tej warstwy sprowadzają się do walidacji, pobierania, modyfikacji, zapisu oraz usuwania obiektów a także list obiektów. Są także, podobnie jak w przypadku prezentera, bezparametrowe. Ważne jest, że ReDSeeDS nie generuje automatycznie żadnej implementacji dla metod warstwy modelu. Programista musi stworzyć implementację samodzielnie. Nie jest to zadanie skomplikowane, ponieważ polega głównie na wywołaniu odpowiednich metod z pakietu „App.DAO" oraz zwróceniu odpowiedniej wartości. W przypadku metod walidujących, modyfikujących, zapisujących bądź usuwających obiekty z bazy wartość zwracana jest typu „int" tak jak to wygenerował ReDSeeDS. Jeśli chodzi o pobieranie obiektów wartość zwracana została zmieniona na odpowiedni typ. Przykład uzupełnionej kodem metody z warstwy modelu pokazuje następujący fragment kodu.

```
public List<XTrainingData> DownloadsTrainingsList(){

	ConnectionFactory f = new ConnectionFactory();
	Connection connection = f.getConnection();
	TrainingDataDAO s_dao = new TrainingDataDAO(connection);
	List<XTrainingData> trainings = new ArrayList<XTrainingData>();
```

```
        trainings = s_dao.getTrainings();
        return trainings;
}
```

Metoda służy do pobrania listy dostępnych treningów na potrzeby ich prezentacji na liście. Na początku tworzony i inicjowany jest obiekt klasy **ConnectionFactory** oraz **Connection**. Następnie inicjowany jest odpowiedni obiekt **TrainingDataDAO**. Do pobrania listy treningów wykorzystana została zaimplementowana wcześniej metoda *getTrainings()*. Wartością zwracaną jest lista treningów. Inny przykład prezentuje metodę zapisującą dane nowej promocji.

```
public int SavesPromotionData(XPromotionData p){

        ConnectionFactory f = new ConnectionFactory();
        Connection connection = f.getConnection();
        PromotionDataDAO p_dao = new PromotionDataDAO(connection);
        int res = p_dao.savePromotion(p);
        return res;
}
```

W tym przypadku konieczne było dodanie parametru wejściowego jakim jest obiekt **XPromotionData**, aby umożliwić przekazanie danych wprowadzonych przez użytkownika za pomocą prezentera. Pozostałe metody z warstwy modelu zostały zaimplementowane na tej samej zasadzie i nie ma potrzeby ich szczegółowego omawiania.