Spis treści

1. Wstęp

1.1. Uzasadnienie i wybór tematu

Wzrost znaczenia aplikacji internetowych jako narzędzi pracy dla różnorodnych firm czy też przedsiębiorstw wciąż rośnie. Firmy produkcyjne, dystrybucyjne oraz sklepy korzystają z tych rozwiązań z uwagi na szybkość wymiany informacji oraz jej globalny zasięg. Ponadto brak konieczności instalacji dodatkowych bibliotek również przemawia za wyborem takiej, a nie innej formy wspomagania działalności. Również korzyści finansowe wynikające z uproszczonego modelu dystrybucji oprogramowania, oszczędność czasu i wzrost efektywności stanowią o sile aplikacji internetowych.

Obecnie jednym z najpopularniejszych zastosowań języka Java jest tworzenie komercyjnych aplikacji internetowych. Nie wystarczające są już rozwiązania znane z początków istnienia tego języka, z uwagi na wciąż rosnące potrzeby i wymagania klientów. Rezultatem tego były narodziny nowej gałęzi programów, a dokładniej ich zbioru - szkieletów aplikacji programistycznych, których głównym celem jest odciążenie programisty oraz wzrost jakości dostarczanych rozwiązań. Popyt generuje podaż - to zdanie jest wciąż bardzo aktualne ponieważ potrzeba lepszej jakości rozwiązań była motorem napędowym ich tworzenia i obecnie na rynku istnieje ponad 50 framework'ów, z publicznie dostępnym kodem źródłowym oraz dobrej jakości dokumentacją techniczną, napisanych w języku Java i przeznaczonych dla aplikacji biznesowych.

Obecnie na rynku istnieje co najmniej kilkanaście rozwiązań dedykowanych w mniejszym lub większym stopniu dla warsztatów samochodowych. Wspólnym mianownikiem dla wyżej wymienionych aplikacji jest zestaw funkcji przez nich realizowanych. Planowanie pracy warsztatu: data, godzina, czas realizacji, przypisany mechanik czy też lista czynności do wykonania przy pojeździe klienta, historia zleceń, kartoteka klientów, samochodów to jedynie niewielka część z nich. Niemniej wszystkie znalezione aplikacje łączy również brak otwartego kodu źródłowego oraz brak w pełni funkcjonalnych wersji dostępnych bez opłat. Część z nich to także programy działające na komputerze użytkownika, a nie w przeglądarce internetowej.

Czy potrzeba wypisać programy które znalazłem ?

Wybrany temat obrazuje użycie jednego ze szkieletów aplikacji dla przygotowania kompleksowego rozwiązania wspierającego warsztat samochodowy, które rozwiązywałoby te problemy. Program, będący częścią praktyczną pracy dyplomowej, będzie aplikacją dostępną przez przeglądarkę internetową, bez konieczności instalacji dodatkowych bibliotek lub aplikacji, z otwartymi źródłami oraz dostępną bez opłat.

1.2. Cel i zakres pracy

Głównym celem pracy jest poznanie wybranego szkieletu aplikacji internetowych, zrozumienie zasad pracy z nim w celu przygotowania aplikacji internetowej wspierającej misję przedsiębiorstwa - warsztatu samochodowego.

Pod kątem funkcjonalności, szczególna uwaga zostanie poświęcona zagadnieniom związanym z tworzeniem efektywnego modelu danych oraz bezpieczeństwu wrażliwych informacji takich jak dane osobowe, numery rejestracyjne oraz numery VIN pojazdów, utrzymanie i dostęp do kartoteki pojazdów, mechaników oraz klientów.

Drugim celem pracy jest dostarczyć wiedzy o projektowaniu oraz implementacji aplikacji biznesowych w języku Java przy wykorzystaniu Spring'a. Z tego powodu zagadnieniom takim jak: minimalizacja ilości kodu realizującego warstwę danych oraz ich walidacja, skupienie się na właściwej funkcjonalności aplikacji, przygotowanie generycznych modułów upraszczających dalszy rozwój aplikacji, jej utrzymanie oraz elastyczny model konfiguracji poświęcona zostanie szczególna uwaga. Ponadto przygotowany program ma służyć jednocześnie stworzeniu kilku potencjalnych modułów, które można by potem wykorzystać w innych projektach do podobnych celów.

2. Wymogi funkcjonalne aplikacji

2.1. Głowna funkcjonalność biznesowa

Analiza istniejących rozwiązań pozwoliła ustalić, że minimalna funkcjonalność programu realizujące wsparcie dla przedsiębiorstwa prowadzącego warsztat samochodowy nie powinna się ograniczać jedynie do wspomagania procesów: wymiany płynów eksploatacyjnych czy też napraw. Prowadzenia ewidencji klientów, pojazdów oraz ich wzajemnych powiązań przekłada się na efektywność pracy, dając jednocześnie dostęp do danych historycznych. Jest to szczególnie użyteczne w przypadku okresowych wymian oleju silnikowego czy też płynu chłodniczego, kiedy można odwołać się do danych z poprzedniej wizyty danego klienta i uzupełnić ewentualne braki magazynowe. Funkcjonalność programu dla warsztatów samochodowych to między innymi wsparcie w następujących obszarów:

1. **klienci**:
   a) ewidencja, możliwość rejestracji nowych klientów,
   b) powiązania z samochodem
2. **samochody**:
   a) historia własności,
   b) historia wizyt
3. **zlecenia serwisowe**:
   a) wprowadzenia nowych zleceń,
   b) edytowanie zleceń które nie zostały jeszcze zamknięte (nie minął termin ich wykonania)
   c) tworzenia listy zadań dla każdego zlecenia,
   d) usuwania zleceń,
   e) powiązania zlecenie - samochód - klient
4. **powiadomienia** - wewnętrzny system wiadomości:
   a) powiadomienia o nowym zleceniu,
   b) powiadomienia o zbliżającym się zleceniu,
   c) powiadomienia o zmianie stanu zlecenia

2.2. Architektura aplikacji

Aplikacja została zrealizowana w architekturze trójwarstwowej według modelu MVC[1], poprzez wzgląd na rozmiar aplikacji oraz mnogość funkcji implementowanych przez program. Niejednokrotnie, z uwagi na szczegółowość modelu danych, wygodniejsze okazuje

[1] Model-View-Controller

się przetwarzanie danych na poziomie serwera, co dodatkowo przemawia za wybraną architekturą. Dzięki warstwie kontrolerów można uzyskać wydajny sposób na przekazywanie informacji na linii klient-serwer, wykonanie operacji zgodnych z jej treścią i odesłanie wyniku. Skomplikowana logika biznesowa, w której na poszczególne wydarzenia należy reagować inaczej, zależnie od parametrów, jest powodem dla którego praca z danymi na poziomie klienta aplikacji mogłaby się okazać, z uwagi na konieczność pobrania dodatkowych danych z serwera, utrudniona bądź niemożliwa. Złożone przypadki można wydajniej i dokładniej obsłużyć mając dostęp do pełnego modelu danych oraz interfejsów operacji bazodanowych. Niemniej logika biznesowa nie jest jedynym powodem dla którego wybrana została architektura MVC. Kolejną przyczyną był poziom skomplikowania modelu danych, odnoszący się do ilości atrybutów oraz zależności między poszczególnymi obiektami. Powiązania, a także możliwe akcje wynikające z paradygmatu **CRUD**, wymusiły konieczność zastosowania efektywnego rozwiązania, dzięki któremu można by reagować na wspomniane akcje.

2.3. Minimalizacja własnego kodu

W momencie projektowanie nowej aplikacji bardzo często początki pracy to tworzenie kodu potrzebnego do wykonywania powtarzalnych operacji, takich jak zapis i odczyt z bazy danych, reagowanie na zdarzenia interfejsu użytkownika czy też walidacja danych wejściowych. Nie jest to problematyczne dla małych projektów. Niestety, wraz ze wzrostem ilości linii kodu, złożoności klas czy też skomplikowania logiki biznesowej, okazuje się, że początkowo proste moduły rozrastają się w niekontrolowany sposób. Odwrotnie proporcjonalnie zmniejsza się możliwość na utrzymanie i rozbudowę kodu aplikacji. Dodatkowe problemy pojawią się wraz z wprowadzeniem do programu cache'owania, konwertowania danych między niekompatybilnymi typami, transakcji operacji bazodanowych. W tym momencie wzrost ogólnej złożoności kodu, nie będącego częścią właściwej funkcjonalności, jest jedynie jednym z problemów. Rosnąca zależność klas miedzy sobą, głębokości drzewa dziedziczenia dla klas oraz konieczność objęcia wspomnianymi elementami kolejnych obszarów programu przekłada się na zmniejszenie jakości kodu. Z tego powodu głównym założeniem części praktycznej jest wykorzystanie możliwie jak najwięcej funkcji oferowanych przez wybrane biblioteki do realizacji powtarzalnych operacji:

- generowanie zapytań SQL,
- operacje zapisu i odczytu z/do bazy danych,
- obsługa transakcji bazodanowych,
- walidacja danych pod kątem logiki biznesowej,
- konwertowanie danych miedzy niekompatybilnymi typami,
- efektywne reagowanie na zdarzanie interfejsu użytkownika,
- minimalizacja ilości kodu linii JSP, skupienie się na właściwych stronach dostarczających interfejs,
- zmniejszenie kohezji między klasami,
- zmniejszenie głębokości drzew dziedziczenia

Podniesienie jakości kodu oraz możliwości jego późniejszego utrzymania oraz rozszerzenia o nowe funkcje stanowi o sile każdej aplikacji. Z tego powodu był to główny wyznacznik przy projektowaniu i implementacji części praktycznej pracy dyplomowej.

2.3.1. Generyczne moduły

Generyczne moduły są odpowiedzią na podniesienie optymalności oraz możliwości późniejszego rozszerzenia aplikacji. Koszt zaprojektowania niezależnych elementów realizujących pewne zadania jest akceptowalny z uwagi na fakt, że teoretyczny moduł przeznaczony, w którym dane wejściowe to obiekty modelu danych, można następnie użyć dla nowych obiektów tego typu bez konieczności modyfikacji lub nieznacznej zmiany istniejącej funkcjonalności.

2.4. Przewodniki - kreatory nowych obiektów

Biorąc pod uwagę funkcjonalność w zakresie realizacji celów biznesowych jednym z nadrzędnych celów jest dostarczenie możliwości tworzenia nowych obiektów modelu danych. Z uwagi na stopień skomplikowania modelu, jego wzajemnych relacji, pojedyncze formularze to zły wybór. Podzielenie całego procesu na sekwencję kroków, gdzie w kolejnych etapach wprowadzamy logicznie ze sobą powiązane informacje odpowiadające pewnym cechą danego obiektu, daje możliwość, w kolejnych krokach przewodnika, uwzględniania wcześniej wpisanych treści. Ponadto przetwarzanie danych odbywa się na serwerze. Dzięki czemu istnieje dostęp do istniejących mechanizmów walidacji oraz innych danych na podstawie których możliwa jest wstępna weryfikacja obiektu na poziomie powiązań z innymi obiektami. Tym samym możliwość wprowadzenia niepoprawnych informacji, których ewentualne sprawdzenie pod kątem długości łańcuchów znakowych mogłoby nie przechwycić, zostaje zmniejszona.

2.5. Terminarz - organizacja wizyt

Z uwagi na specyfikę aplikacji, jako narzędzia służącego przede wszystkim organizacji pracy warsztatu, terminarz jest funkcją której nie można pominąć. Intuicyjne używanie kalendarza jako terminarza, gdzie zapisywane są poszczególne spotkania, wizyty czy też zadania do wykonania, jest niezwykle przydatne w tego typu aplikacjach. Terminarz przygotowany w części praktycznej posiadać będzie 3 tryby widoku: **dzienny**, **miesięczny**, **tygodniowy**. Z uwagi na częstotliwość wizyt, ilość mechaników, a co ważniejsze ilość klientów, tym samym ich samochodów, podane 3 tryby pracy z terminarzem są koniecznością. Z poziomu kalendarza będzie można tworzyć nowe wizyty, zmieniać ich godziny lub je edytować.

3. Zaplecze technologiczne - wykorzystane biblioteki

3.1. Wybór narzędzi do pracy

Złożoność aplikacji, opisana w rozdziale 2, wymagała narzędzi potrzebnych do sprostania poszczególnym wymogom funkcjonalnym. Z prostej przyczyny opcja wielu zewnętrznych bibliotek została odrzucona: zbyt wielka ich ilość z czasem mogłaby stać się problemem, z uwagi na ewentualne konflikty między nimi lub wzrost liczby zależności przechodnich[1]. Spośród wielu dostępnych szkieletów aplikacji zdecydowano się, w pierwszej kolejności, na wybór tych, które wspierają **Dependency Injection** oraz **Inversion Of Control**. Obie techniki pozwalają znacząco podnieść jakość kodu, zminimalizować stopień kohezji oraz zależności między klasami. Jest to możliwe, ponieważ odpowiedzialność za sterowanie przepływem programu zostaje przeniesiona na kod użytej biblioteki, co odciąża programistę od konieczności kontrolowania tych aspektów. Po analizie takich rozwiązań jak **JBoss Seam Framework**, **Google Guice**, **PicoContainer**, **Spring Framework** okazuje się, że wszystkie z nich wspierają wymienione techniki. Niemniej stopień w jakim można by użyć jednego z nich do kompleksowego wsparcia aplikacji praktycznej nie jest już jednakowy. Zaczynając od **PicoContainer** oferującego wsparcia jedynie dla **Dependency Injection**, a kończąc na **Spring Framework** oraz **JBoss Seam Framework** posiadających najbogatszy wachlarz możliwości, najlepszym wyborem okazuje się być **Spring**. Z uwagi na popularność w środowisku programistów stanowi on doskonały kompromis między tym, co oferuje, a tym czego wymaga. Pozwalając na przygotowania aplikacji bez żadnego wyraźnego nacisku na technikę czy też inne technologie, jest wysoce konfigurowalnym fundamentem dla dowolnego programu.

3.2. Znaczenie szkieletów aplikacji

Internetowe szkielety aplikacji nie są same w sobie bibliotekami programistycznymi. Stanowią one raczej ich zbiór, jak również zestaw narzędzi, mających na celu ułatwienie programiście implementacji własnego rozwiązania. Bardzo często są one również praktyczną implementacją standardów (tak jak **Seam Framework**) i tak zwanych **best practices**[2]. Jest to szczególnie użyteczne ponieważ nierzadko zdarza się, że programista popełnia błąd

[1] Zależność przechodnia - zależność jednej biblioteki od innej, a w efekcie zależność aplikacji od ostatniej z wykorzystanych w łańcuchu bibliotek.

[2] Zalecane i pożądane sposoby realizacji często spotykanych problemów

na pewnym etapie projektowanie lub implementacji określonego modułu, którego późniejsze konsekwencje wymagają stworzenia niepotrzebnego i nadmiarowego kodu, czego dałoby by się uniknąć, gdyby podążano już wyznaczonymi ścieżkami. Prawdziwe w takim wypadku staje się również zdanie, że jeden błąd generuje kolejne, a te mogą być zalążkiem następnych.

Powodem istnienia szkieletów aplikacji jest więc zapobieganie takim sytuacjom, poprzez proponowanie już gotowych modułów, które są przetestowane i ciągle modyfikowane przez doświadczone osoby, celem dostarczenia jeszcze lepszych rozwiązań[3].

Oprogramowanie zorientowane obiektowo jest doskonałym zobrazowaniem koncepcji wykorzystania szkieletu jako fundamentu do budowy własnego rozwiązania. Na najniższym poziomie szczegółowości każdy program czy też moduł większej części, jest zbiorem klas posiadającym jasno określony zbiór ról - obowiązków, a których obiekty współpracują ze sobą, celem dostarczenia gotowego wyniku lub jego części. Wspólnie te obiekty reprezentują pewną koncepcję, dla realizacji której zostały utworzone.

W kontekście szkieletu aplikacji internetowych można więc wyróżnić klasy przeznaczone do kooperacji z bazą danych, odpowiedzialnych za walidację informacji czy też pomocnych w momencie renderowania widoku. Warto nadmienić, że te zasady są równie ważne dla małych systemów, jak i dla dużych. Niemniej, w pierwszym przypadku, gdzie poziom skomplikowania jest niski, nie ma potrzeby definiowania wielu poziomów abstrakcji ułatwiających określone czynności, jak na przykład wcześniej wymienione walidacje danych. Niestety z czasem, początkowo prosty system, staje się coraz bardziej skomplikowany i bardzo często programista nie jest już wtedy w stanie zapanować na chaosem oraz dostarczyć zunifikowanego sposobu rozwiązywania powtarzalnych czynności. Z tego powodu dobry framework charakteryzuje się jasno, ale nie sztywno zdefiniowanymi granicami między poszczególnymi zbiorami funkcjonalnymi. Wprowadzone poziomy abstrakcji, często więcej niż jeden dla pojedynczego celu, jak na przykład sposób interakcji systemu i jego klientów, są wynikiem wieloletnich zmian, podczas których zidentyfikowano wiele wspólnych problemów i dla których znaleziono rozwiązanie w postaci ram projektowych czy też **best pratices**, będących ostatecznie właściwą esencją znaczenia szkieletu aplikacji[7].

Dobrymi przykładami tutaj będą z pewnością warstwy abstrakcji dla obsługi operacji bazodanowych. Zawierają one konkretne implementacje, posiadające funkcjonalność odpowiedzialną za wykonanie tych operacji na praktycznie elementarnym poziomie, zostawiając właściwą warstwę logiki w tworzonej aplikacji, odciążają one programistę od przysłowiowego wynajdowania koła od nowa. Praktyczną realizacją tej koncepcji jest na przykład **Spring Data** pozwalające na napisanie kodu, którego głównymi zaletami będzie odseparowanie logiki biznesowej od wybranej bazy danych oraz wyraźny podział na klasy odpowiedzialne za operacje **CRUD** na danych, jak i te wykonujące operacje biznesowe. Inne przykłady to między innymi **EJB**, czy też moduł innego szkieletu programistycznego **GWT** wykonującego identyczne zadanie. Warto nadmienić, że również warstwy odpowiedzialne za tworzenie i zarządzanie widokiem (warstwa prezentacji) oraz takie, których nadrzędnym celem jest pośredniczenie między widokiem a danymi, są potencjalnymi kandydatami do wyodrębnienia pewnego zbioru funkcjonalności, jako części składowych gotowego szkieletu aplikacji.

3.2.1. Funkcjonalność szkieletów aplikacji

Do najczęściej implementowanych funkcjonalności szkieletów aplikacji internetowych należą:

- internacjonalizacja oraz lokalizacja tworzonych stron w dowolnym języku oraz wsparcie dla efektywnego przełączania między nimi,
- wsparcie dla technologii widoków innych niż strony **JSP** lub takich, które je wykorzystują, ale definiują odmienny sposób korzystania z nich,
- integracja z językiem szablonów innym niż **JSTL**,
- walidacja danych,
- mapowanie żądań HTTP do tzw. **kontrolerów**[3],
- wsparcie dla popularnych języków transportu danych, jak JSON czy też jego odpowiednik XML.

3.2.2. Problemy szkieletów aplikacji

Mimo że szkielety aplikacji znacząco podnoszą jakość kodu oraz obniżają późniejsze koszty jej utrzymania, nie są doskonałym narzędziem. Większość niedociągnięć, które można zaobserwować związane są z:

- *złożonością klas* - obiekty klas zaimplementowane w szkielecie aplikacji współpracują ze sobą, wielokrotnie w więcej niż jednym kontekście. Definiowanie funkcjonalności danej klasy poprzez użycie pojedynczej klasy abstrakcyjnej lub interfejsu jest rozwiązaniem zbyt sztywnym, ponieważ często większa część zdefiniowanych metod nie będzie wykorzystywana w innym miejscu,
- *skupieniem się na szczególe* - w momencie projektowania klas, tj. kreowania późniejszego celu istnienia ich obiektów, zdarza się, że gubi się obraz całości zbytnio skupiając się na poszczególnych przypadkach,
- *złożoność współpracy* - mechanizmy współpracy obiektów odpowiadających, przykładowo za komunikację klient-serwer, mogą stać się zbyt skomplikowane,
- *trudnością użycia* - brak drobiazgowej dokumentacji może skutkować użyciem szkieletu w sposób niezamierzony przez jego twórców, co może skutkować implementowaniem tzw. **work arounds**[4] lub błędami funkcjonalnymi[7].

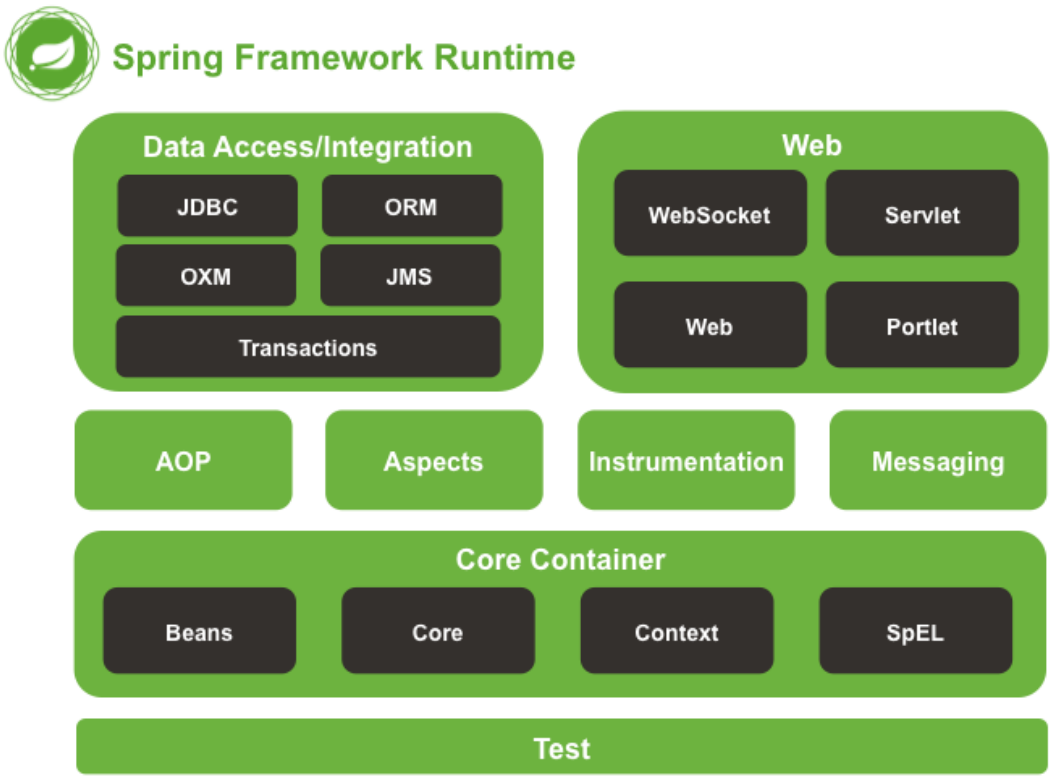
3.3. Spring Framework

Spring jest szkieletem tworzenia aplikacji w języku Java dla platformy Java (Standard Edition oraz Enterprise Edition) opisywanym jako *lekki szkielet aplikacji*. Lekkość szablonu odnosi się tutaj nie do rozmiarów całości, ale do filozofii, jaka przyświecała i cały czas przyświeca **Spring'owi**. Nie wymusza konkretnego stylu programowania czy też używania konkretnych zewnętrznych bibliotek, jednocześnie dając możliwość praktycznie

[3] Kontrolery należy rozumieć jako tzw. **POJO** które same w sobie są zwykłymi klasami, ale w kontekście framework'a nabierają konkretnego znaczenia jako wykonawcy pewnej logiki właściwej dla danej aplikacji

[4] Technika programistyczna, której celem jest naprawa jakiegoś błędu bądź uzyskanie zamierzonego celu podczas, gdy używany framework lub biblioteka nie pozwalają na dotarcie do niego.

dowolnej integracji. Dobrym przykładem jest mnogość opcji, które można wybrać w przypadku pisania warstwy widoku aplikacji internetowej. Spring oferuje wsparcie czystego JSP (ze wsparciem tagów JSTL) jednocześnie dając możliwość użycia takich bibliotek jak Velocity, FreeMarker, XSLT czy Apache Tiles. **Spring** składa się z ponad 20 samodzielnych



Rysunek 1: *Kontener Spring wraz z modułami*
źródło: [9]

modułów pogrupowanych zgodnie z obszarem:

- **Core Container** - fundament szkieletu na którym oparte są pozostałe modułu. Definiuje on funkcjonalność **Dependency Injection** oraz odwróconego sterowania[5], a także posiada definicję obiektów takich jak **Bean**, **Context**. Ostatecznie zawiera w sobie klasy niezbędne do ładowania zasobów, lokalizacji[6] oraz **Expression Language** - manipulowania obiektami poprzez wyrażania zapisane czystym tekstem, które są później tłumaczone na odpowiednie wywołania programowe,
- **Data Access/Integration** - określa sposób dostępu dla takich źródeł danych jak bazy danych, pliki XML, czy też zdalne źródła danych dostępne przez protokół JMS. Najważniejszą zaletą jest maksymalne wykorzystanie spójnych interfejsów do dostępu do danych i ukrycie ich źródła.

[5] IoC - Inversion Of Control
[6] Lokalizacja - internacjonalizacja aplikacja, wsparcie dla więcej niż jednego języka

- **Web** - jego zadaniem jest umieszczenie aplikacji działającej w kontenerze Spring w kontekście kontenera serwletów[7].
- **AOP** - dostarcza sposoby oraz środki do programowania aspektowego[8].

Modularna budowa jest praktyczną realizacją koncepcji odseparowania obszarów funkcjonalnych. Dzięki temu podejściu **Spring Framework** może być użyty w dowolnej konfiguracji, korzystając jedynie z wstrzykiwania zależności, odwróconego sterowania lub kolejnych modułów odpowiedzialnych za architekturę MVC dla aplikacji trójwarstwowych, dostępu do danych, a także przesyłania danych protokołem JMS.

3.3.1. Użyte moduły Spring'a

Spring MVC

Spring MVC jest zorganizowany wokół centralnego servletu DispatcherServlet oraz klas opatrzonych adnotacjami: @Controller lub @RestController - **kontrolerów**. Kontrolery są punktem łączącym warstwę widoku oraz logiki biznesowej. Konfigurowane za pomocą adnotacji są alternatywą dla standardowych serwletów. Ich główną zaletą jest możliwość ich wykorzystania w wielu przypadkach użycia, jako obiektów wywołujących operacje warstwy logiki biznesowej lub wspierających przetwarzanie danych z formularzy. Korzystając z takiego podejścia, programista nie jest zmuszony do definiowania kilku, bądź kilkunastu oddzielnych serwletów, z których każdy odpowiadałby innemu przypadkowi użycia[9] lub pisania własnego silnika, który pozwalałby na generyczne i automatyczne wywołania konkretnych metod w zależności od adresu odpowiadającemu danemu kontrolerowi.

W warstwie kontrolerów wykorzystywany jest wysoce elastyczny mechanizm odpowiedzialny za konwertowanie danych między niekompatybilnymi typami. Dzięki niemu praktycznie nie istnieje konieczność tworzenia własnych mechanizmów przeznaczonych do tego celu. Ponadto wszelkie błędy związane z tym procesem nie są traktowane jako błędy systemu, ale jako błędy konwersji. Odciąża to programistę od implementowania kolejnych klas, których pola odpowiadają skomplikowanym cechom właściwego modelu danych, a które, z uwagi na brak możliwości przedstawienia ich w pierwotnej formie, musiałyby być przedstawione jako typy znakowe.

Ostatecznie **Spring MVC** oferuje wsparcie dla operacji, której celem jest zrenderowanie pewnego widoku. Kontroler jest najczęściej odpowiedzialny za przygotowanie danych, które zostaną umieszczone w odpowiedzi wysłanej do klienta, oraz wybranie widoku poprzez unikatową nazwę. Dalszy proces zależy od wybranej technologii, użytej dla implementacji warstwy widoku. Istnieje możliwość wykorzystania zarówno plików JSP, jak i bibliotek gdzie końcowy widok jest złożeniem kilku innych. Z drugiej strony, kontroler nie jest ograniczony jedynie do wybrania widoków. Programista ma możliwość zwracania kompletnych kolekcji obiektów, które później zostaną wysłane do klienta w formacie JSON

[7] Servlet Container - komponent serwera aplikacji internetowej zarządzający między innymi cyklem życia serwletów, mapowaniem adresów URL

[8] Aspect Oriented Programming - sposób na zmniejszenie wzajemnej zależności klas oraz objęciem pewną funkcjonalnością pewnych obszarów aplikacji bez jawnego wykorzystania w nich konkretnych klas. Logika zdefiniowana w pojedynczym miejscu jest wykorzystywana w wielu miejscach

[9] **Przypadek użycia** (ang. usecase) jest sposobem opisu wymagań aplikacji na poziomie interakcji między użytkownikiem końcowym (aktorem), a systemem

lub XML. Wybrany format może zostać łatwo zmieniony poprzez odpowiednią konfigurację projektu.

Głównymi zaletami **SpringMVC** są:

- wyraźny podział obowiązków pomiędzy poszczególnymi artefaktami (kontrolery, walidatory, formularze),
- uproszczona oraz wysoce elastyczna konfiguracja:
  — adres pod którym kontroler jest dostępny,
  — typ żądania: **GET**, **POST**, **DELETE**, **PUT**,
  — typ zwracanych danych: **nazwa widoku**, **JSON**, **XML**,
- brak konieczności duplikowania modelu danych,
- wsparcia dla różnorodnych technologi widoku: **JSP**, **Velocity**, **Apache Tiles** lub **JSF**,

Spring Data

Jest to praktyczne rozwiązanie problemu związanego z implementacją warstwy dostępu do danych. Eliminuje konieczność implementacji szablonowego i powtarzalnego kodu, którego głównym zadaniem jest wykonanie operacji na bazie danych określanych skrótem **CRUD**[10]. Warto w tym miejscu zwrócić uwagę na generyczne API, które przekłada się na wysoki poziom abstrakcji, dzięki któremu możliwe jest korzystanie z praktycznie dowolnego źródła danych poprzez jednolity interfejs. Nie ważne staje się, czy dane przechowywane są w bazie danych **MySQL** lub **Oracle**, czy też w bazach nierelacyjnych, jak na przykład **MongoDB**.

Spring Data JPA

Spring Data JPA jest częścią **Spring Data**, zawierającym artefakty szczególnie użytecznie dla relacyjnych baz danych, jak na przykład MySQL. Jednym z tych elementów są repozytoria. Repozytorium jest niczym innym jak obiektem w naszej aplikacji dzięki któremu uzyskujemy faktyczny dostęp do danych i możemy nimi zarządzać. Co ważniejsze pojęcie to jest znacznie szersze niż mogłoby się wydawać, zwłaszcza w kontekście operacji wyszukiwania. Poniższy przykład kodu (1) pokazuje klasę **JpaRepository**. Istniejące tam deklaracje metod są jedynie rozszerzeniem tych zdefiniowanych w kolejnych interfejsach: **PagingAndSortingRepository** oraz **CrudRepository**. Niemniej widać, że nawet na wyższym poziomie abstrakcji programiści **Spring Data** zadbali o bardzo wiele możliwych przypadków użycia, co przekłada się na końcową produktywność programisty.

[10] CRUD - *Create-Read-Update-Delete* - zbiór czterech podstawowych funkcji w aplikacji korzystających z pamięci trwałej jako nośnika przechowywania danych, które umożliwiają zarządzanie nimi.

```java
@NoRepositoryBean
public interface JpaRepository<T, ID extends Serializable>
        extends PagingAndSortingRepository<T, ID> {

    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAll(Iterable<ID> ids);

    <S extends T> List<S> save(Iterable<S> entities);

    void flush();

    T saveAndFlush(T entity);

    void deleteInBatch(Iterable<T> entities);

    void deleteAllInBatch();

    T getOne(ID id);
}
```

Listing 1: **JpaRepository** interfejs dla operacji bazodanowych na relacyjnej bazie danych w **Spring Data**

Ponadto nie ma konieczności implementacji takiego interfejsu. Aby utworzyć nowe repozytorium dla konkretnego obiektu domenowego należy utworzyć nowy interfejs. Zostanie on zaimplementowany podczas działania programu poprzez proxy. W tym miejscu proxy jest pośrednikiem, gdzie odbywa się proces tłumaczenia wywołań metod repozytorium na kwerendy SQL. Repozytoria posiadają także inną, bardzo interesującą cechę - automatyczne mapowanie metod na kwerendy. Jest to alternatywa dla nazywanych kwerend znanych ze standardu **JPA**. Zapytanie SQL pobierane jest z nazwy metody, co oczywiście wymusza pewną konwencję nazewnictwa. Niemniej jest to koncepcja ciekawa i idealnie nadaje się do tworzenia zapytań odnoszących się do 1 lub 2 atrybutów danego obiektu, których użycie jest równoznaczne z wykorzystaniem operatora **where** języka SQL. Wywołanie jest silnie typizowane, dlatego programista ma pewność, że obiekt będzie tego typu, który go interesuje. Dla bardziej skomplikowanych kwerend istnieje możliwość zadeklarowanie metody i oznaczenia jej adnotacją *Query* z kodem JPQL [4] [6].

Głównymi zaletami **Spring Data JPA** są:

- silne typizowanie danych,
- automatyczne tłumaczenie nazw metod na kwerendy SQL,
- szeroka gama operacji wyszukiwania,
- gotowa implementacja operacji **CRUD**,
- uproszczona konfiguracja,
- jednolite interfejsy dostępu do danych, niezależne od źródła danych,
- minimalna ilość kodu niezbędna do utworzenia repozytoriów dla obiektów modelu danych

Spring Security

W momencie pisania aplikacji w technologii **Java EE**[11] nie można zapomnieć o problemie nieautoryzowanego dostępu do strony lub do niektórych jej części. Sposób uzyskania takiej funkcjonalności jest zależny od kontenera w którym działamy. Inaczej to zagadnienie rozwiązywane jest w przypadku **Apache Tomcat**, a inaczej w przypadku **JBoss**. Oba z nich są serwerami aplikacji Javy, niemniej tak samo jak identyczny jest cel ich istnienia, tak samo różna jest implementacja kwestii autoryzacji. Dzięki **Spring Security** programista może korzystać z niezależnego od kontenera, wysoce konfigurowalnego mechanizmu kontroli dostępu do zasobów. W tym miejscu warto nadmienić, że moduł można dostosować do weryfikacji użytkowników zarówno z wykorzystaniem bazy danych, jak i stałej listy zawierającej nazwy użytkowników posiadających dostęp do aplikacji. Ponadto niewielkim nakładem pracy można dodać mechanizm kontroli, znany pod nazwą **Access Control List**. Jest to koncepcja, gdzie prawa dostępu (zarówno zapisu, odczytu czy tez modyfikacji) związane są z konkretnym typem obiektu. Wszystkie wyżej wymienione cechy czynią **Spring Security** doskonałym wyborem do ochrony wrażliwych elementów aplikacji internetowej.

Spring HATEOAS

[11] Jest to praktyczna implementacja paradygmatu znanego jako **HATEOAS - Hypermedia as the Engine of Application State**. W przypadku **HATEOAS** duży procent funkcjonalności aplikacji jest oferowany w postaci hiperłączy powiązanych z obiektami. To, co wyróżnia to podejście to fakt, że klienci nie muszą znać funkcjonalności do momentu, kiedy jest ona im prezentowana na stronie internetowej. Przykładowa odpowiedź w rozumieniu tego paradygmatu mogłaby wyglądać następująco:

```
{
    "name"          : "John",
    "lastname"   : "Doe",
    "links"         : [
        {
            "rel" : "self",
            "href": "http://localhost/customer/1"
        },
        {
            "rel" : "parent",
            "href": "http://localhost/customer/1/parent"
        }
    ]
}
```

Listing 2: Przykładowa odpowiedź serwera na akcję w rozumieniu paradygmatu HATEOAS

Spring Web Flow

"Przemieszczenia się kogoś, czegoś, przekazywanie, obieg czegoś."[8]

[11] Java Enterprise Edition

SWF[12] jest szczególnie użyteczne gdy aplikacja wymaga powtarzalności tych samych kroków w więcej niż jednym kontekście. Czasami taka sekwencja operacji jest częścią większego komponentu, co desygnuje je do wyodrębnienia ich jako samodzielnego modułu. Najlepszym przykładem użycia są w tym wypadku różnego rodzaju formularze służące do rejestracji użytkowników czy też kreatory nowych obiektów, gdzie umieszczenie wszystkich wymaganych pól na jednej stronie mogłoby zaciemnić obraz i uniemożliwić użytkownikowi zrozumienie działania. Ponieważ **SWF** jest modułem Spring, jest on w pełni zintegrowany z platformą **Spring MVC** 3.3.1 oraz silnikiem walidacji i konwersji typów.

Flow [10] - jest centralnym obiektem modułu, w którym definiowane są kolejne kroki przepływu. Dzięki deklaratywnemu językowi XML definicje są czytelne, a możliwości których dostarcza **SWF** pozwala na kreowanie sekwencji w dowolny sposób, łączenie kroków z modelem danych, korzystania z podstawowych jak i zaawansowanych mechanizmów implementacji akcji. Akcje zawierają właściwą logikę biznesową dla konkretnej fazy przepływu, ale także pozwalają na pobieranie danych wejściowych oraz zwracanie wyników na ich podstawie. Są także zalecanym sposobem obsługi błędów związanych z logiką biznesową.

```
<view-state id="entity" view="ui.wizard.NewReportWizard.PickEntity" popup="true">
    <on-render>
        <evaluate expression="pickEntityFormAction.setupForm"/>
    </on-render>
    <transition on="next" bind="true" validate="true" to="pickColumns">
        <evaluate expression="pickEntityFormAction.bindAndValidate"/>
    </transition>
    <transition on="cancel" to="cancel" history="invalidate"/>
</view-state>
```

Przykład 3 pokazuje kod XML, który definiuje jeden z kroków - stanów. Powyższy przykład korzysta z klasy org.springframework.webflow.action.FormAction. Metoda setupForm może służyć między innymi do wprowadzenia danych wejściowych do kontekstu przepływu.

```
@Override
public Event setupForm(final RequestContext context) throws Exception {
    final MutableAttributeMap<Object> scope = this.getFormObjectScope().getScope(context);
    final Set<RBuilderEntity> entities = this.getReportableEntities(context);
    scope.put(ENTITIES, entities);
    scope.put(ASSOCIATION_INFORMATION, this.getReportableAssociations(entities));
    return super.setupForm(context);
}
```

Listing 4: *PickEntityFormAction#setupForm* - metoda *setupForm* wykorzystywana w definicji kroku 3 do umieszczenia danych w kontekście przepływu

3.4. JPA - Java Persistance API

JPA - Java Persistance API jest standardem określającym reguły opisu mapowania obiektowo-relacyjnego dla języka **Java**. W aplikacji demonstracyjnej **JPA** została zastosowana do opisania wszystkich obiektów należących do modelu danych na poziomie nazw tabel, nazw kolumn oraz wzajemnych powiązań między różnymi obiektami, czyli innymi

[12] Skrót od Spring Web Flow

słowy relacji klucz główny - klucz obcy na poziomie tabel w bazie danych. Użycie standardu jako sposobu specyfikacji modelu danych, zostało podyktowane wykorzystaniem tego samego standardu przez inne biblioteki, a w konsekwencji moduły aplikacji.

3.5. Hibernate - Object Relational Mapping

Wykorzystanie **Hibernate** w aplikacji jest w głównej mierze transparentne. Dzięki **Spring Data (3.3.1)** rola silnika ORM zostaje zmniejszona do wykonawcy mapowania obiektowo relacyjnego. W swojej najnowszej wersji **Hibernate**, pomijając część specyficznych dla siebie sytuacji, opiera się o standard **JPA** 3.4. Takie podejście pozwala ponownie na zaprojektowanie modelu danych, którego przenośność jest wysoka, a migracja do innego szkieletu aplikacji ORM ogranicza się do wybrania takiego, który obsługiwałby standard.

3.6. c3p0

c3p0 jest łatwą do użycia biblioteką zaprojektowaną dla **Java**, której głównym zadaniem jest realizacja postulatów zdefiniowanych przez specyfikacje **jdbc3**. Dzięki powyższej bibliotece można w łatwy sposób zdefiniować n - połączeń z bazą danych, gdzie kolejne z nich będą wykorzystywane jeśli kolejka żądań do innych będzie już pełna. Także elementy takie jak zarządzanie zajętymi zasobami, ich zwalnianie są obsługiwane przez **c3p0**. Dzięki wsparciu dla szkieletu aplikacji **Spring** konfiguracja okazuje się trywialna i polega na zadeklarowaniu odpowiedniego obiektu w pliku konfiguracyjnym XML lub adnotacji na poziomie języka Java [2].

3.7. QueryDSL

QueryDSL jest projektem wspierającym **Spring Data JPA 3.3.1**. Z tego powodu, podobnie jak **Spring Data JPA**, **QueryDSL** zostało zaprojektowane aby dostarczyć wysoko poziomowego interfejsu budowania zapytań do bazy danych. Ponadto, w porównaniu z API Hibernate, **QueryDSL** pozwala na konstruowanie silnie typizowanych zapytań. Dzięki temu programista nie jest zmuszony do rzutowania oraz wcześniejszego sprawdzania, czy obiekt, który chce uzyskać odpowiada jego wymaganiom. Podczas pracy z **QueryDSL** najistotniejszym elementem jest meta model, który generowany jest z modelu danych podczas kompilacji przez **ATP**. To, co świadczy najlepiej o **QueryDSL** to fakt, że wspomniany meta model to klasy języka Java. Wynikająca z tego korzyść polega na tym, że wraz z meta modelem, można tworzyć zapytania korzystając z uzupełnia składni, oferowanego przez praktycznie wszystkie środowiska programistyczne. Inną ważną zaletą są pola klas meta modelu. Odpowiadają one atrybutom biznesowego modelu danych, dzięki czemu nie ma konieczności pamiętania o ich wewnętrznych nazwach[13], jakby to miało miejsce podczas pisania identycznego zapytania opartego o łączenia łańcuchów znakowych.

[13] Wewnętrzna nazwa - nazwa kolumn odpowiadającej danemu atrybutowi klasy modelu danych

3.8. Ehcache

Ehcache jest biblioteką dostarczającą funkcjonalność pamięci podręcznej dla aplikacji Java oraz Java Enterprise. Główną zaletą posiadania takiego rozwiązania jest odciążanie bazy danych, ponieważ część zapytań oraz ich wyników zapisana jest w pamięci lub w systemie plików. Użyteczność tej biblioteki potwierdza zasada znana, jako **zasada Pareto**, czyli stosunku 80:20. Jeśli weźmiemy pod uwagę 20% obiektów (np. rekordów z bazy danych), które używane są przez 80% czasu działania aplikacji to używając pamięci podręcznych możemy poprawić wydajność aplikacji o koszt uzyskania 20% obiektów.

W ogólnym zarysie idea działania pamięci podręcznej opiera się na tablicy asocjacyjnej, gdzie każdemu z unikatowych kluczy odpowiada pewna wartość. Podczas umieszczania obiektu do pamięci obliczana jest unikatowa wartość klucza dla tego obiektu. Samą pamięć można opisać jako miejsce, w którym czasowo przechowuje się obiekty pochodzące z bazy danych lub wyniki długotrwałych obliczeń. Podczas próby pobrania elementu z cache można mówić o pojęciu **hit** - element dla danego klucza zostaje znaleziony oraz o pojęciu **miss**, kiedy element o danym kluczu nie istnieje w pamięci podręcznej [12].

3.9. Apache Tiles

Apache Tiles to biblioteka umożliwiająca dekompozycję widoku aplikacji na wiele niezwiązanych ze sobą bezpośrednio elementów - płytek [14]. Płytki można dowolnie łączyć w konkretne widoki, definiując je na poziomie plików XML. Główną zaletą korzystanie ze wzorca kompozycji jest wyeliminowanie powielania się elementów stron i zastąpienie ich szablonami gotowymi do użycia w dowolnym miejscu. Narzut obliczeniowy potrzebny na połączenie kilkunastu płytek w gotowy szablon jest akceptowalny z uwagi na obniżenie złożoności pojedynczych plików JSP oraz wyeliminowanie problemu duplikowanego kodu JSP. Dodatkową zaletą użycia tej biblioteki było gotowe wsparcie dla modułu Spring'a – Spring Webflow 3.3.1, gdzie jedną z preferowanych technologii widoku jest właśnie Apache Tiles, a także możliwość prostszego wsparcia dla **partial rendering**[15] stron, gdzie podczas przechodzenia do innego adresu w rzeczywistości zamiast ładować całość strony wraz ze wszystkimi plikami *CSS* oraz *JavaScript*, ładuje się jedynie treść danej strony.

3.10. Jasper Reports/Dynamic Jasper

Jasper Reports to kompleksowe rozwiązanie dla języka **Java** wspierające tworzenie oraz generowanie raportów biznesowych dla różnorodnych formatów wyjściowych: **PDF**, **XLS**, **CSV**. Jego główną zaletą jest pojedynczy format przechowywania raportu oraz mnogość formatów reprezentacji, a także ogromna ilość narzędzi oraz bibliotek wspierających tworzenie i modyfikacje raportów. Z drugiej strony wiele tych narzędzi jest aplikacjami uruchamianymi na komputerach użytkowników, a nie zaprojektowanych do wykorzystania

[14] Z angielskiego *tiles* może oznaczać płytkę, w kontekście technologii **Apache Tiles** należy rozumieć to wyrażenie, jako element, który można wykorzystać w dowolnym szablonie

[15] Partial rendering należy rozumieć, jako usunięcie konieczności przeładowania całej strony WEB, a jedynie jej konkretnej części.

w środowisku aplikacji WEB. Z tego powodu właściwą biblioteką, która została użyta celem utworzenia raportu jest **Dynamic Jasper**. Działając po stronie serwera oraz bazując na danych wejściowych uzyskanych od użytkownika pozwala na kompilację do pliku **.jasper*. Możliwe jest ustalenie takich właściwości jak nagłówki, styl, ilość kolumn oraz typ danych w nich przechowywanych.

3.11. Dandelion Datatables

Dandelion-Datatables jest biblioteką zaprojektowaną dla języka **Java**, której zadaniem jest wsparcie dla tworzenia tabel korzystając z użyciem tagów JSP. Instrukcje dostarczane tą drogą uruchamiają proces generowania kodu JavaScript dla wtyczki **jQuery - DataTables**. Nie ma konieczności bezpośredniego pisania kodu JS co pozwala na budowania responsywnych tabel bez znajomości języka JavaScript. **Dandelion Datables** pozwala na bezproblemowe sortowanie, filtrowanie oraz eksportowanie danych.

3.12. Słownik pojęć

Tabela 1: Adnotacje Spring opisujące poziom abstrakcji cache

Pojęcie	Znaczenie
Obiekt domenowy	Klasy takich obiektu zdefiniowane są w modelu danych. Innymi słowy dostarczają one definicji informacji opisujących, w sposób abstrakcyjny, rzeczywiste obiekty wykorzystywane w aplikacji.
Repozytorium	Abstrakcyjne pojęcie odnoszące się do klasy obiektów - interfejsów leżących na linii baza danych - aplikacja, pośredniczących w operacjach zapisu/odczytu. Implementują ideę stojącą za pojęciem CRUD.
CRUD	Anglojęzyczny skrót wykorzystywany w programowaniu opisujący 4 podstawowe operacje wykonywany na pewnym źródle danych. • create - utworzyć, • read - odczytać, • update - uaktualnić, zmodyfikować, • delete - usunąć
Cache	Cache jest specjalnym obiektem przeznaczonym do czasowego przechowywania danych w pamięci dla zapewnienia szybszego dostępu niż w przypadku odwoływania się do bazy danych, plików lub metod wykonujących kosztowne obliczeniowo operacje.
ATP	Pojęcie odnosi się do przetwarzania adnotacji języka Java i wykonywania pewnych operacji i/lub generowania wyniku.
Adnotacje	Ideą adnotacji jest dodawanie do kodu źródłowego aplikacji metadanych.
	Następna strona...

Tabela 1 – kontynuacja...

Pojęcie	Znaczenie
AOP	**Aspect Oriented Programming** jest paradygmatem programistycznym zwiększającym skalowalność, zmniejszającym kohezję klas i tym samym eliminującym tak zwana *ścisłe zależności*. Programowanie z jego wykorzystaniem odwołuje się do organizacji kodu w tak zwane *aspekty* realizujące pewną logikę. Aspekt swoim zasięgiem jest w stanie objęć więcej niż jeden poziom abstrakcji zdefiniowanego z użyciem technik programowania obiektowego jak na przykład wzorce strategii czy też fabryk.
Kohezja	Kohezja, w odniesieniu do programowania, oznacza stopień w jakim dwie klasy są zależne od siebie.
Getter/Setter	Zwyczajowe pojęcia opisujące metody dostępowe klasy służące do pobierania lub ustawienia wartości pól danej obiektu tej klasy
API	Ściśle określony zbiór reguł i metod, dzięki którym program może się komunikować ze sobą lub z innym programem.

4. Aplikacja demonstracyjna

4.1. Opis aplikacji wspomagającej warsztat samochodowy

Nadrzędnym celem aplikacji jest wsparcie dla misji przedsiębiorstwa[1] prowadzącego warsztat samochody, zajmujący się serwisowaniem samochodów, prowadzącym naprawy, przeglądy i dokonującym okresowych czynności eksploatacyjnych jak na przykład wymiana oleju czy filtrów. Z tego powodu w kolejnych modułach aplikacji została zrealizowana część zarówno serwerowa jak i kliencka dostarczająca funkcjonalności pozwalających na tworzenie, edycję oraz usuwanie obiektów biznesowych, a także przeglądanie informacji o nich. Duży nacisk został położony na zrealizowanie warstwy serwerowej z uwagi na jej krytyczne znaczenie. Jest ona odpowiedzialna za realizację postulatów logiki biznesowej, zarządzanie prawami dostępu, walidację i konwersję danych. Z uwagi na wymienione punkty wiele elementów zostało dopracowanych, a poszczególne autorskie implementacje pewnych problemów zastąpione lepiej przetestowanymi i wydajniejszymi bibliotekami.

4.1.1. Funkcjonalność aplikacji

W części praktycznej zrealizowane została następująca funkcjonalność:

1. weryfikacja dostępu do konkretnych stron:
   - aplikacja rozpoznaje czy użytkownik jest zalogowany, dostosowując ilość dostępnych funkcji, w zależności od grupy (grup), w których użytkownik się znajduje,
   - weryfikacja jest jedno etapowa
2. tworzenie nowych spotkań,
3. przeglądanie terminarza spotkań,
4. tworzenie nowych raportów, zapisywanie ich oraz późniejsze generowanie dla nich wyników,
5. tworzenie nowych użytkowników:
   - nazwa użytkownika oraz hasło,
   - zestaw ról definiujących poziom uprawnień,
   - dane kontaktowe
6. przeglądanie obiektów domenowych jako pojedynczych stron internetowych.

4.1.2. Generyczny moduły

Praca nad niewidoczną dla użytkownika końcowego częścią generycznych modułów zaowocowała opracowaniem 3 niezależnych modułów. Działając po stronie serwera uprasz-

[1] Misja przedsiębiorstwa - zestaw wartości opisujących rolę danego przedsiębiorstwa w jego otoczeniu

czają pracę z takimi elementami, jak strony wyświetlające informacje o obiektach pochodzących z modelu danych, definiowanie nowych tabel i zbioru danych oraz ostatecznie tworzenie, zapisywanie i generowanie raportów biznesowych.

Strony obiektów domenowych oraz tabele

Komponenty istnieją w dwóch różnych wariantach. Wariant 1 służy do generowania stron dla obiektów biznesowych, zwanych dalej **info page**, natomiast zadaniem wariantu drugiego jest generowanie zarówno konfiguracji, jak i danych dla tabel. Oba rodzaje zostały zaprojektowane, aby zminimalizować ilość potrzebnych do przesłania danych. Powodem istnienia dwóch różnych klas jest różnica w konstrukcji strony HTML a tabeli. Dodatkowo w przypadku tabeli pewne charakterystyczne elementy, takie jak zmienne opisujące zachowanie kolumn czy też całej tabeli zostały wymuszone przez zastosowanie biblioteki **Dandelion Datatables** 3.11.

```
public interface ComponentBuilder<COMP extends Serializable> {

    String getId();

    Class<?> getBuilds();

    ComponentBuilds.Produces getProduces();

    ComponentDataResponse<?> getData() throws ComponentException;

    COMP getDefinition() throws ComponentException;

    void init(ComponentDataRequest componentDataRequest);
}
```

Listing 5: **ComponentBuilder** - korzeń hierarchii modułu komponentów, który wyznacz rolę tego rodzaju obiektów w systemie.

Szczególnie ważne są tutaj następujące metody:

- **getDefinition()** - wywołanie następuje w momencie kiedy użytkownik otwiera stronę, w strukturze DOM, gdzie zapisana jest informacja o obiekcie biznesowym, pobrana z aktualnego adresu URL.
- **getData()** - jest to metoda zaprojektowana do wygenerowania asynchronicznej odpowiedzi **Ajax** na żądania pobrania gotowego widoku renderowanego komponentu.

Z uwagi na dwa oddzielne odnogi **ComponentBuilder**'ów w strukturze dziedziczenie oraz różnych wymogów, co do konstrukcji końcowego widoku prezentowanego użytkownikowi, istnieją także dwa kontrolery. Jeden z nich został zaprojektowany specjalnie dla **info page**. Rolą drugiego jest umożliwić generowanie tabel.

Poniższe listingi pokazują implementacje metod w warstwie kontrolerów odpowiednio wywołujących funkcje **getData()** oraz **getDefinition()**.

```
@RequestMapping(value = "/{path}/{id}", method = RequestMethod.GET)
public ModelAndView getInfoPageView(@PathVariable("path") final String path,
                                    @PathVariable("id") final Long id) throws InfoPageNotFoundException {
    LOGGER.debug(String.format("/getInfoPageView/path=%s/id=%s", path, id));
    final SInfoPage page = this.getInfoPageForPath(path);
    if (page != null) {
        LOGGER.trace(String.format("Resolved infoPage => %s for path => %s", page, path));
        final ModelMap modelMap = new ModelMap();

        modelMap.put(InfoPageConstants.INFOPAGE_PAGE, page);
        modelMap.put(InfoPageConstants.INFOPAGE_VIEW_DATA_TEMPLATE_LINK, this.getViewDataTemplateLink());

        return new ModelAndView(
                this.getViewForPage(page),
                modelMap
        );
    }
    throw new InfoPageNotFoundException(path);
}
```

Listing 6: Obsługa żądania w kontrolerze, które wywołuje metodą **getDefinition()** dla strony domenowej

```
@ResponseBody
@RequestMapping(value = "/data/{id}")
public DatatablesResponse<?> getBuilderData(
        @PathVariable("id") final String builderId,
        final ComponentTableRequest tableRequest,
        final WebRequest request) throws ControllerTierException {
    LOGGER.info(String.format("/getBuilderData -> builder=%s, tableRequest=%s", builderId, tableRequest));

    final ComponentBuilder<?> builder = this.builders.getBuilder(builderId, new ModelMap(ComponentConstants.REQUEST_BEAN, tab
    try {
        if (builder != null && builder instanceof TableComponentBuilder) {
            LOGGER.trace(String.format("Found builder %s:%s:%s", builderId, builder.getId(), builder.getBuilds()));
            return DatatablesResponse.build((DataSet<?>) builder.getData().getValue(), tableRequest.getCriterias());
        }
    } catch (Exception e) {
        LOGGER.error("/getBuilderData threw exception", e);
        throw new ControllerTierException(e);
    }
    return null;
}
```

Listing 7: Obsługa żądania w kontrolerze, które wywołuje metodą **getData()** dla tabeli

Dla tabel istnieje możliwość zdefiniowania atrybutów do zaprezentowania użytkownikowi, które bezpośrednio nie istnieją w klasie odpowiadającej danemu obiektowi. Należy wtedy przeciążyć metodę *handleDynamicColumn*, której zadaniem jest zwrócenie wartości dla danego, **dynamicznego** atrybutu. Poniższy kod pokazuje przykładową implementację:

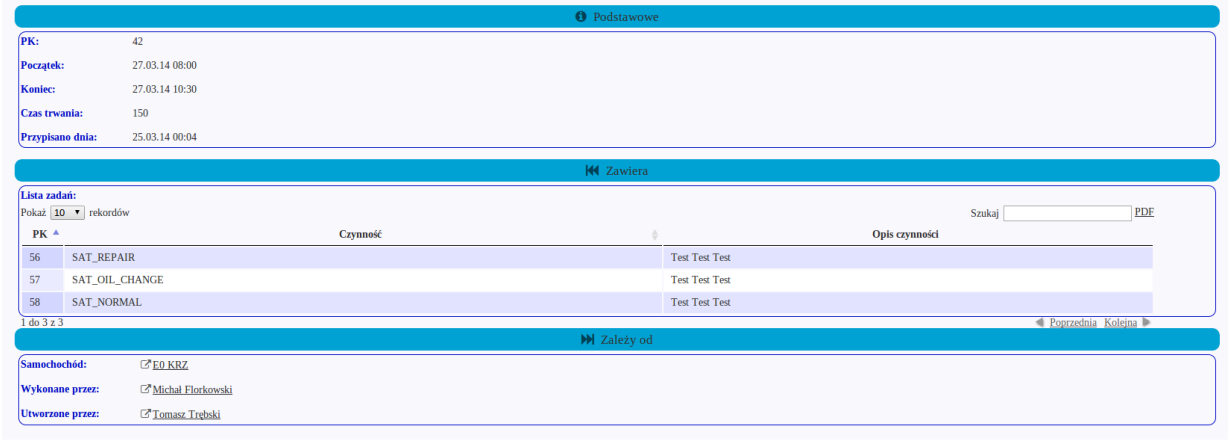
```
@Override
protected Object handleDynamicColumn(final SReport object, final String path) throws DynamicColumnResolutionException {
	Object retValue = super.handleDynamicColumn(object, path);
	if (retValue != null) {
		return retValue;
	}
	switch (path) {
		case "generate-action": {
			try {
				final Link link = linkTo(ReportBuilderController.class).slash(object.getId()).withSelfRel();
				retValue = new PopupAction().setType(RequestMethod.POST)
						.setUrl(link.getHref());
			} catch (Exception exception) {
				final String message = String.format("Error in creating link over path=%s", path);
				this.logger.error(message, exception);
				throw new DynamicColumnResolutionException(message, exception);
			}
		}
		break;
		case "delete-action": {
			try {
				final Link link = linkTo(ReportBuilderController.class).slash(object.getId()).withSelfRel();
				retValue = new AjaxAction().setType(RequestMethod.DELETE)
						.setUrl(link.getHref());
			} catch (Exception exception) {
				final String message = String.format("Error in creating link over path=%s", path);
				this.logger.error(message, exception);
				throw new DynamicColumnResolutionException(message, exception);
			}
		}
	}
	return retValue;
}
```

Listing 8: Uzyskania wartości dynamicznego atrybutu w momencie tworzenia odpowiedzi dla tabeli

Zadaniem kodu na listingu 8 jest zwrócenie akcji dla poszczególnych wierszy, które użytkownik będzie mógł wykonać na obiekcie odpowiadającym danemu rekordowi w tabeli.

InfoPage - strony obiektów modelu danych

InfoPage jest komponentem prezentującym atrybuty danego obiektu domenowego w postaci strony internetowej. Dzięki wykorzystaniu informacji pobranych z meta modelu obiektów domenowych, utworzenie nowej strony sprowadza się do zaimplementowania rozszerzenia specjalnej klasy *EntityInfoPageComponentBuilder*. Uzyskanie dostępu do danych i wkomponowanie ich do struktury drzewa nie wymaga w tym miejscu dalszej implementacji kodu o ile dany atrybut istnieje w klasie **Java**. W innym wypadku jest on atrybutem dynamicznym i należy zaimplementować kod, który zwróci jego wartość. Moduł jest wysoce generyczny i zautomatyzowany. Utworzenie strony dla nowego typu w modelu danych, które pojawiły by się w systemie, sprowadza się do utworzenia klasy, gdzie zdefiniowania zostanie struktura strony - panele oraz ich atrybuty. Warto w tym miejscu zaznaczyć, że tak zwany **panel systemowy** dodawany jest do każdej strony, jeśli powiązany z nią obiekt jest wersjonowany, tj. posiada historię zmian.



Rysunek 2: *Strona domenowa dla spotkania*

Na rysunku 2 pokazana została strona domenowa obiektu opisującego pojedyncza wizytę w warsztacie samochodowym. Zdefiniowane zostały 3 panele z atrybutami. Panel podstawowy oraz dwa opisujące relacje zachodzące między spotkaniem a innymi obiektami. Spotkanie posiada więc listę zadań oraz zostało przypisane do pewnego samochodu, zgłoszone i wykonane przez konkretnych mechaników. Listing kodu 9 przedstawia kod Java odpowiedzialny za zbudowania struktury strony.

```
@EntityBased(entity = SAppointment.class)
@ComponentBuilds(id = "appointmentInfopage", builds = AppointmentInfoPage.class)
public class AppointmentInfoPageComponentBuilder
        extends EntityInfoPageComponentBuilder<SAppointment> {

    private static final long serialVersionUID = 7743548253782408262L;

    @Override
    protected Logger getLogger() {
        return Logger.getLogger(AppointmentInfoPageComponentBuilder.class);
    }

    @Override
    protected InfoPageComponent buildDefinition() {
        final InfoPageComponent cmp = new InfoPageComponent();
        this.populateBasicPanel(helper.newBasicPanel(cmp, LayoutType.VERTICAL));
        this.populateTablePanel(helper.newOneToManyPanel(cmp, LayoutType.VERTICAL));
        this.populateInfoPagePanel(helper.newManyToOnePanel(cmp, LayoutType.VERTICAL));
        return cmp;
    }

    private void populateInfoPagePanel(final InfoPagePanelComponent panel) {
        this.helper.newLinkAttribute(panel, "car", this.getEntityName());
        this.helper.newLinkAttribute(panel, "assignee", this.getEntityName());
        this.helper.newLinkAttribute(panel, "reporter", this.getEntityName());
    }

    private void populateBasicPanel(final InfoPagePanelComponent panel) {
        this.helper.newAttribute(panel, "id", "persistentobject.id", AttributeDisplayAs.VALUE);
        this.helper.newValueAttribute(panel, "begin", this.getEntityName());
        this.helper.newValueAttribute(panel, "end", this.getEntityName());
        this.helper.newValueAttribute(panel, "interval", this.getEntityName());
        this.helper.newValueAttribute(panel, "assigned", this.getEntityName());
    }

    private void populateTablePanel(final InfoPagePanelComponent panel) {
        this.helper.newTableAttribute(panel, "tasks", this.getEntityName());
    }
}
```

Listing 9: Strona domenowa dla spotkania - kod źródłowy

TableBuilder - definiowanie oraz dane dla tabel

Rzadko zdarza się żeby aplikacja nie wymagała korzystania z tabel do prezentowania danych. Są one szczególnie użyteczne zwłaszcza w momencie, kiedy ilość możliwych obiektów do jednorazowego wyświetlenia sięga co najmniej kilkudziesięciu elementów. W części praktycznej za obsługę tabel odpowiada biblioteka **Dandelion Datatables**. Renderowanie tabel oraz wsparcie dla funkcji takich jak sortowanie, filtrowanie. Niemniej ta biblioteka, podobnie jak wiele innych, nie posiada bezpośredniego wsparcia dla generowania struktur tabel oraz danych po stronie serwera, ze szczególnym naciskiem na definicję tabeli. Moduł **TableBuilder** rozwiązuje problemy takie jak:

- dynamiczna struktura tabel - to jakie kolumny będą widocznie zależne jest od dowolnych czynników,
- wsparcie dla sortowania po stronie serwera,
- wsparcie dla filtrowania po stronie serwera,
- zwracania danych tekstowych dla kolumn odpowiadających danym nie atomowym[2],
- wsparcie dla dynamicznych kolumn, które nie odpowiadają żadnej z cech obiektów wyświetlanych w danej tabeli

[2] Obiekt atomowy - pojedynczy, jeden. Obiekt nie atomowym to obiekt zwierający więcej niż jedną informację na raz

```
@EntityBased(entity = SCar.class)
@ComponentBuilds(
        id = CarsTableBuilder.BUILDER_ID,
        builds = SCar.class,
        produces = ComponentBuilds.Produces.TABLE_COMPONENT
)
public class CarsTableBuilder
        extends TableComponentBuilder<DandelionTableComponent, SCar> {

    protected static final String BUILDER_ID      = "carsTableBuilder";
    private static final   String TABLE_ID        = String.format("%s%s", "table", StringUtils.uncapitalize(SCar.ENTITY_NAME));
    private static final   Logger LOGGER          = Logger.getLogger(CarsTableBuilder.class);
    private static final   long   serialVersionUID = 3079491907844336996L;

    @Override
    protected Object handleColumnConversion(final SCar object, final Object value, final String path) {
        switch (path) {
            case "owner": {
                return object.getOwner().getPerson().getIdentity();
            }
        }
        return super.handleColumnConversion(object, value, path);
    }

    @Override
    protected Predicate getPredicate(final Long id, final Class<?> contextClass) {
        return QSCar.sCar.carMaster.id.eq(id);
    }

    @Override
    protected Logger getLogger() {
        return LOGGER;
    }

    @Override
    protected DandelionTableComponent buildDefinition() {
        final DandelionTableComponent component = this.helper.newDandelionTable(TABLE_ID, BUILDER_ID);
        this.helper.newTableColumn(component, "id", "persistentobject.id");
        this.helper.newTableColumn(component, "owner", "scar.owner");
        this.helper.newTableColumn(component, "licencePlate", "scar.licenceplate");
        this.helper.newTableColumn(component, "vinNumber", "scar.vinnumber");
        return component;
```

Listing 10: *CarsTableBuilder* - klasa definiująca strukturę tabeli wyświetlającej listę samochodów

RBuilder - raporty biznesowe

RBuilder jest praktyczną realizacją, będącej na wczesnym etapie rozwoju, koncepcji zaprojektowania generycznego silnika raportowania dla aplikacji internetowej. Głównymi założeniami tego komponentu są:

- możliwość generowania raportów przez użytkowników systemu,
- możliwość zapisywania raportów do bazy danych, celem późniejszego ich użycia,
- możliwość edycji istniejących raportów,
- możliwość usuwania istniejących raportów,
- wsparcie dla zabezpieczenie akcji możliwych do wykonania na raportach,
- eksport raportów do formatów:
  - PDF
  - XLS
  - HTML
  - CSV

Jest to obecnie najbardziej skomplikowany komponent aplikacji, którego złożoność jeszcze wzrośnie, z uwagi na założenie, że raporty mogą być definiowane przez użytkowników.

Definiowanie raportu przebiega według następujących kroków:

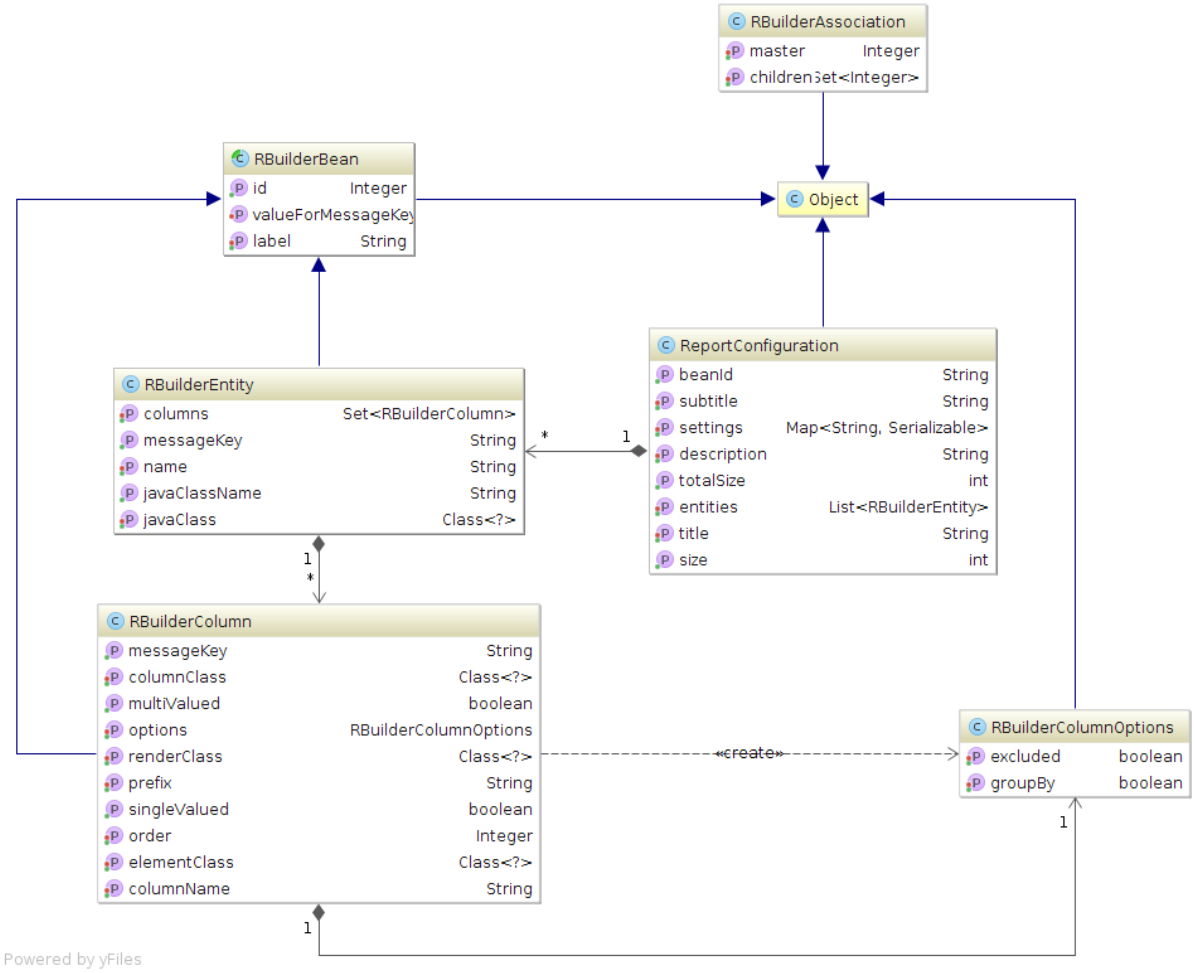
1. Użytkownik wybiera z listy tabele dla których chce wygenerować raport,
2. Dla zaznaczonych tabel użytkownik wybiera kolumny oraz format, w jakim zostaną wyświetlone,
3. Użytkownik podaje informacje takie jak:
   - nazwa
   - opis
4. Sterowanie jest przekazywane do serwisu odpowiedzialnego za:
   - zdecydowanie o rodzaju raportu: dla jednej tabeli, dla wielu tabel,
   - utworzenie obiektu domenowego raportu zawierającego informację takie jak tytuł,
   - utworzenie, kompilacja i zapisanie zserializowanego obiektu klasy **DynamicReport** do systemu plików,
   - zwrócenie sterowania do **Spring Web Flow**
5. Generowanie raportu jest dostępne z tabeli zawierającej listę wszystkich raportów

Report należy w tym miejscu rozumieć jako obiekt domenowy, który służy do późniejszego zapisanie wymaganych informacji do bazy danych oraz jako obiekt **Dynamic Jasper Report**. Celem dla którego tworzony jest ten obiekt leży w wykorzystanej bibliotece **DynamicJasper**. Aby raport mógł być zaprezentowany użytkowniki, musi on zostać w pierwszej kolejności skompilowany do pliku **.jasper*. Załadowanie, a dokładniej odczytanie tego obiektu z zserializowanego pliku, zapisanego w systemie plików, jest wymogiem koniecznym i dostatecznym, aby sterowanie procesem tworzenie jednej z reprezentacji przekazać do szkieletu aplikacji **Spring**. Niemniej nie jest to jedyny wymóg. W specjalnej mapie, pod jasno określonymi kluczami, musi znaleźć się źródło danych dla raportu spójne z jego budową. Powyższe powody generują wiele niedogodności takich jak konieczność przechowywania dodatkowych plików na serwerze oraz trudności w uogólnieniu niektórych aspektów projektowania nowego raportu.

Model danych przeznaczony dla **RBuilder** jest pewnym rozszerzeniem informacji o danych biznesowych (domain object). Zawarte w nim informacje pozwalają na stwierdzenie następujących właściwości obiektów domenowych:

- nazwa tabeli,
- lokalizowana nazwa tabeli,
- atrybuty,
- powiązania z innymi modelami.

Późniejsze instancje obiektów zaprezentowanych na diagramie UML są wykorzystywane do przechowywania informacji niezbędnych do prawidłowego utworzenia konfiguracji raportu **ReportConfiguration**.



Rysunek 3: *Diagram UML modelu danych* ***RBuilder***

Serwisy **RBuilder**'a dostarczają metod, dzięki którym możliwe jest przygotowanie gotowego raportu w wybranej przez użytkownika reprezentacji i wsparcia dla przewodnika tworzenia nowego raportu. Dostarczone usługi to:

- ustalenie możliwych formatów danego atrybuty, a tym samym kolumny w raporcie,
- tworzenie obiektu domenowego w zależności od konfiguracji raportu,
- generowanie raportu,
- zapis i odczyt informacji o raporcie z bazy danych oraz systemu plików.

Funkcjonalność tej grupy klas dla komponentu **RBuilder** można podzielić na następujące bloki:

Tabela 2: Bloki funkcjonalne modelu serwisów **RBuilder**

Grupa	Funkcjonalność
Operation Management	Grupa **Operation Management** odpowiedzialna jest za tworzenie obiektu **SReport** w zależności od ilości tabel wybranych dla konkretnego raportu. Lista klas: • RBuilderOperation • RBuilderCreateOperation • SingleEntityRBuilderCreateOperation • MultipleEntitiesRBuilderCreateOperation
Data Management	Klasy z grupy **Data Management** zostały zaprojektowane do pobierania danych takich jak: • informacje o typach obiektów domenowych, które można uwzględnić w raportach. Takie klasy adnotowane są przez *ReportableEntity*, a ich lista udostępniana jest poprzez interfejs *ReportableEntityResolver*, • listę kolumn wraz z ich cechami takimi jak nazwa, typ danych rzechowywanych w odpowiadającej jej polu w klasie, odpowiednie typu na które można rzutować dany typ. Informacje tego typu udostępniane są poprzez interfejs *ReportableBeanResolver*, • listę powiązań między modelami w uproszczonej formie na potrzeby wybierania tabel podczas projektowania raportu. Na obecną chwile możliwe jest utworzenie jedynie nieprzechodnich powiązań opisanych na bazowym poziomie przez relacje klucz główny - obcy. Dane tego typu udostępnione są przez interfejs *ReportableAssociationResolver*.
Dynamic Jasper Operation	*JasperBuilderService* jest jedyną klasą tej grupy, dostarczającą możliwości utworzenia skompilowanego raportu typu **JasperReport**, który potem jest serializowany do systemu plików. Niemniej jej głównym zadaniem jest wkomponowanie w obiekty wyżej wymienionego typu takich danych jak: • tytuł, • podtytuł, • opis, • język, • szerokość odpowiednich sekcji jak nagłówek, stopka itp., • lista kolumn, • lista kolumn według których dane mają być grupowane.
Następna strona...	

Tabela 2 – kontynuacja...

Grupa	Funkcjonalność
View helper	*ReportBuilderService* jest interfejsem serwisu, którego celem istnienia jest przekazanie sterowania do modułu **Operation Management**, celem utworzenia instancji obiektu domenowego **SReport** oraz wsparcie dla operacji renderowania raportu w konkretnej reprezentacji. Kiedy pierwsza z funkcji jest trywialna w kontekście złożoności, służąc jedynie separacji zadań i zmniejszeniu kohezji klas, druga z wymienionych metod jest dużo bardziej złożona. Jej celem jest pobranie danych wymaganych przez moduł **Spring**, używanych później do zrenderowania raportu w wybranej reprezentacji, na przykład PDF. Operacje przez nią wykonywane to: • pobranie obiektu domenowego z bazy danych dla danego numeru raportu, • deserializacja skompilowanego pliku **.jasper* z systemu plików, • utworzenie źródła danych na podstawie informacji takich jak lista kolumn, ich typ, wybranych typ reprezentacji danych w kolumnie

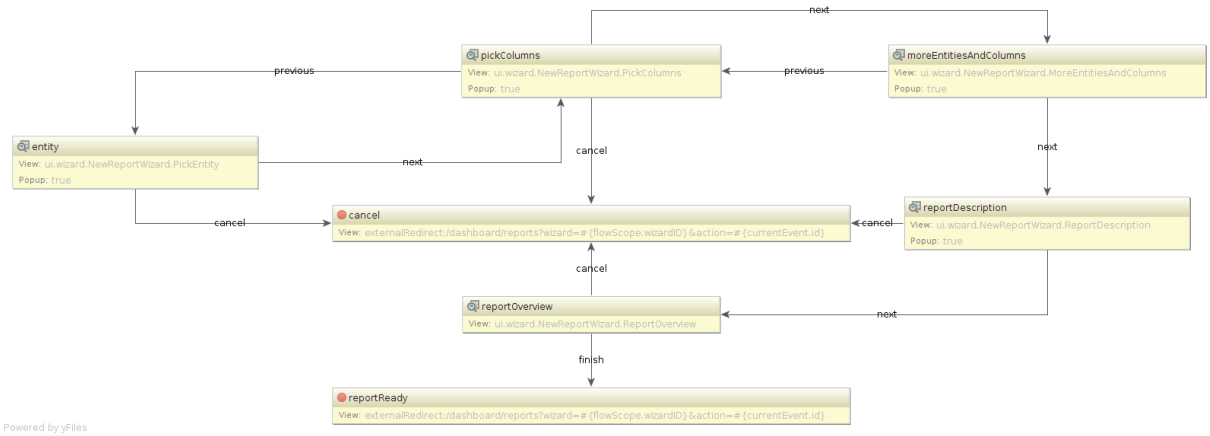
Na część obsługującą widok komponentu **RBuilder** składają się:

- tabela z istniejącymi raportami ,
- przewodnik tworzenia nowego raportu ,
- specjalnie skonfigurowany **ViewResolver** [3].

Tabela (4.1.2) jest obiektem należącym do warstwy widoku, który został utworzony z użyciem komponentu tabeli. Zawiera ona dodatkowo akcje, umożliwiające operacje na raportach.

Przewodnik (4.1.2) został zaprojektowany na podstawie **Spring Web Flow**. Dzięki temu zbiór formularzy, na których definiuje się poszczególne elementy składowe gotowego obiektu **ReportConfiguration** (stanowiącego bazę do generowania wybranej reprezentacji 4.1.2 reportu), jest logicznie połączony. Dodatkową korzyścią jest uzyskanie bezproblemowego wsparcia dla przetwarzania danych wejściowych po stronie serwera bez konieczności pisania własnej logiki do zarządzania komunikacją Ajax oraz możliwość przekazywania danych między kolejnymi krokami.

[3] ViewResolver - interfejs, które implementują specjalne klasy zadaniem których jest ładowanie widoków poprzez odwołanie m.in. do plików JSP, logicznych nazw widoków itp.



Rysunek 4: *Logiczne połączenie kroków w przewodniku dla **RBuilder***

Każdy z kroków przewodnika wspiera odpowiednia klasa Java, będąca specjalizowanym rozszerzeniem FormAction, zarządzającą ustawieniem danych wejściowych dla formularza, walidacji i konwerterów. Ostatecznie jest to rzecz szczególnie ważna, ponieważ praktycznie wszystkie formularze zostały napisana aby wspierać wprowadzanie danych dla więcej niż jednego obiektu. Z uwagi na brak natywnego wsparcia ze strony szkieletu aplikacji **Spring** należało napisać odpowiednie metody, tworzące z prostych łańcuchów znakowych poprawne obiekty opisujące nowy raport.

```
private abstract class BaseConverter
        extends MatcherConverter {
    protected Set<RBuilderEntity> doConvert(final Set<String> list) {
        LOGGER.trace(String.format("converting with selected clazz=%s", list));
        Preconditions.checkNotNull(list);
        Preconditions.checkArgument(!list.isEmpty());
        final Set<RBuilderEntity> reportedEntities = Sets.newHashSet();
        for (final String beanIdentifierString : list) {
            final Integer identifier = Integer.valueOf(beanIdentifierString);
            final RBuilderBean bean = reportableBeanResolver.getReportableBean(identifier);
            if (ClassUtils.isAssignable(RBuilderEntity.class, bean.getClass())) {
                reportedEntities.add((RBuilderEntity) bean);
            }
        }
        return reportedEntities;
    }
}
```

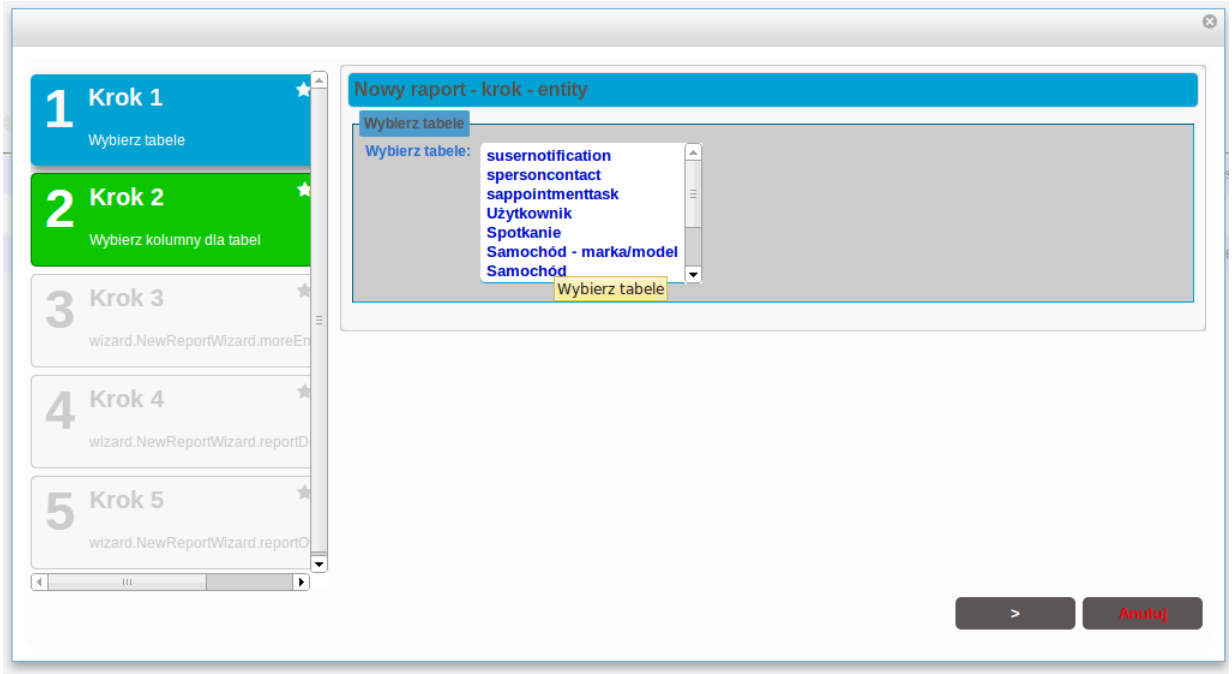
Listing 11: Bazowy konwerter dla operacji **PickEntityFormAction** który tworzy z łańcuchów znakowych obiekty **RBuilderEntity**

4.1.3. Przewodniki tworzenia nowych obiektów

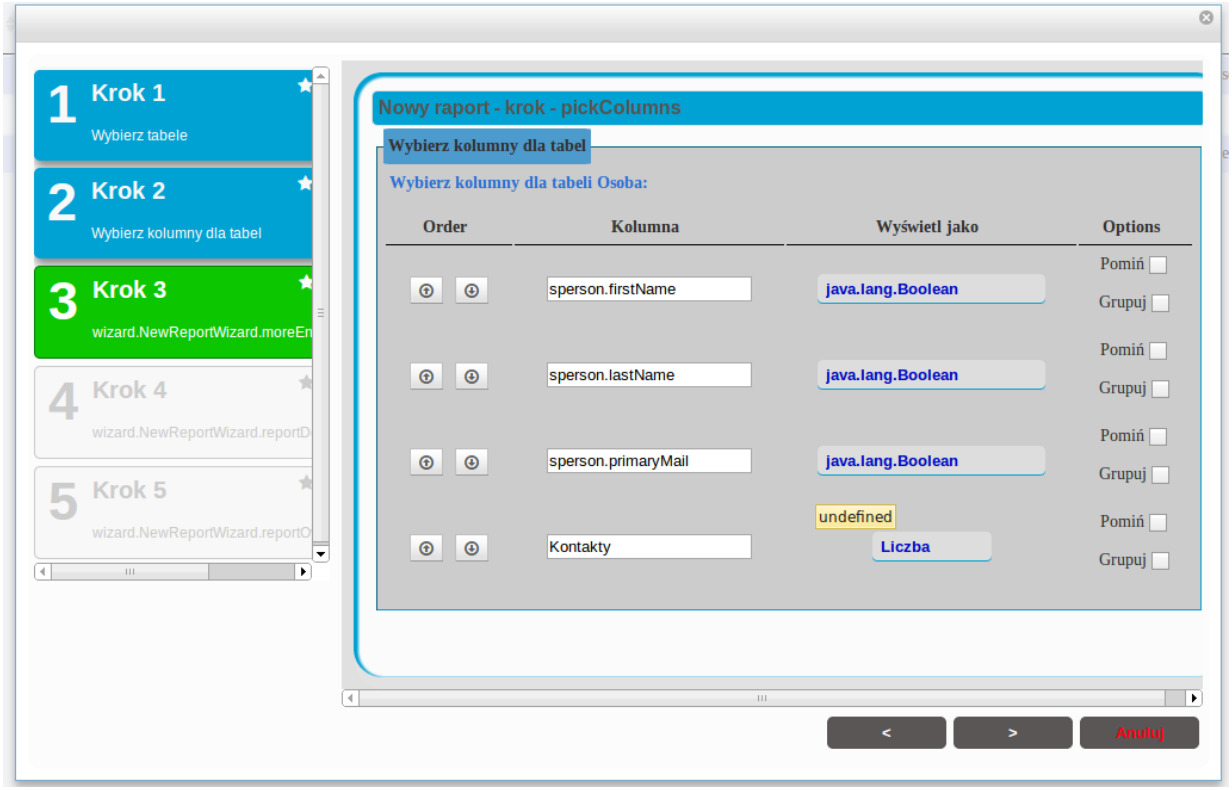
ReportBuilder - reporty biznesowe

ReportBuilder jest narzędziem będącym obecnie w fazie rozwoju, niemniej pozwalającym już teraz na tworzenie raportów biznesowych. Dedykowany dla użytkownika, daje mu możliwość wybrania zbioru interesujących go tabel, kolumn które zebrane razem stanowią logiczny zbiór używany w dalszej kolejności do konstrukcji zapytania do bazy danych i utworzenia gotowego raportu. Nie udało się znaleźć żadnego rozwiązania, które można by wykorzystać bezpośrednio w aplikacji WEB. Przewodnik składa się z 4 wymaganych kroków:

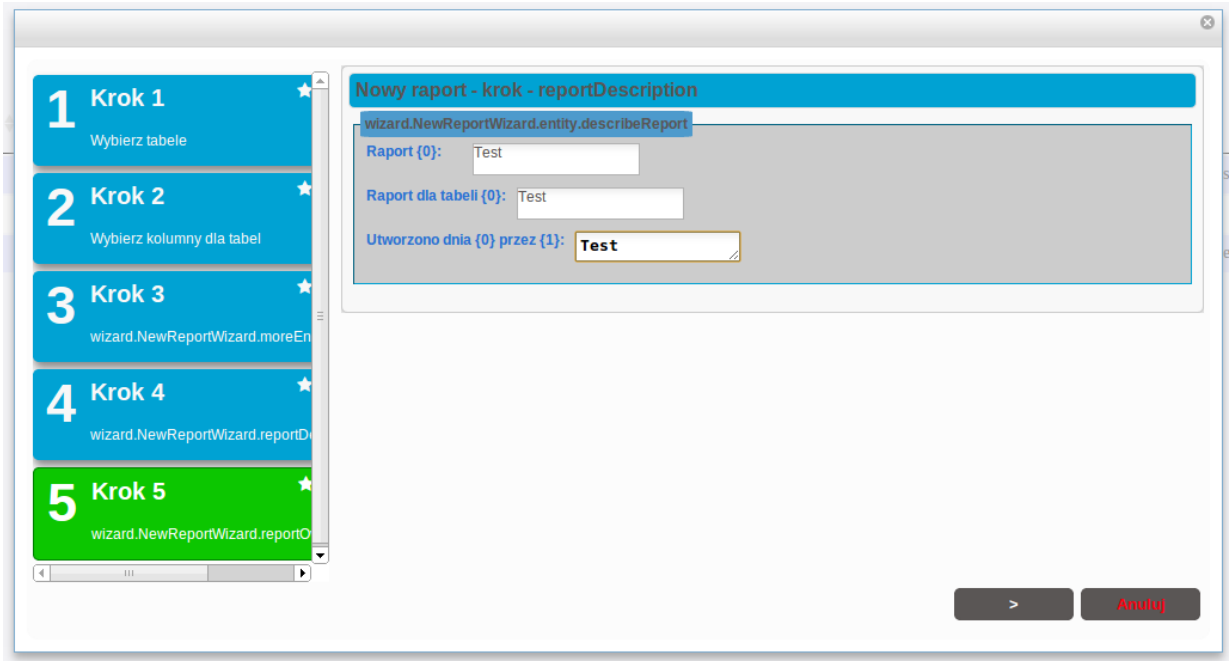
1. wybranie tabeli, dla której wygenerowany zostanie raport
2. dostosowanie formatowania kolumn,
3. podanie danych opisujących raport: tytuł, podtytuł, opis.



Rysunek 5: *Kreator nowego raportu - krok 1*

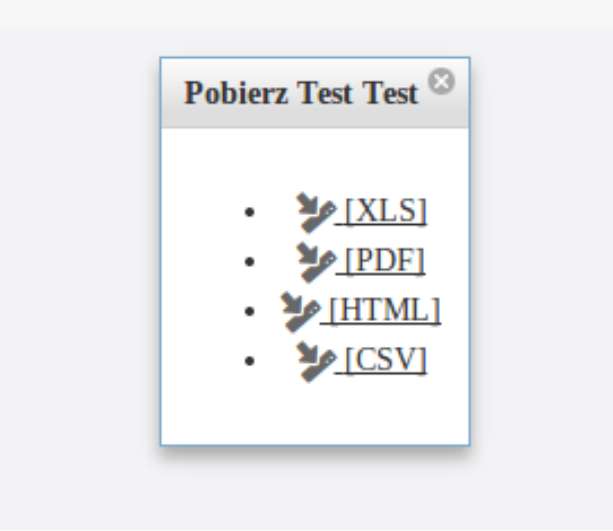


Rysunek 6: *Kreator nowego raportu - krok 2*



Rysunek 7: *Kreator nowego raportu - krok 3*

Dzięki możliwości wykonywania akcji na poszczególnych wierszach tabeli, udało się przypisać dla każdego raportu link uruchamiający jego generowania oraz usunięcie. Usunięcie jest operacją trywialną z punktu widzenia użytkownika, ponieważ nie widzi on niczego poza końcowym rezultatem. Z drugiej strony w momencie kliknięcia na przycisk **Generuj**, użytkownik ma możliwość wybrania końcowego formatu w jakim chciałby zobaczyć swoje dane. Dalsza część funkcjonalności, czyli faktycznego zrenderowania gotowych danych została zaimplementowana z użyciem biblioteki wspierającej **JasperReports**, pochodzącą ze szkieletu aplikacji **Spring**. Przykładowy raport wygląda w następujący sposób:



Rysunek 8: *Generowanie nowego raportu - wybór docelowego formatu:* ***PDF***, ***XLS***, ***CSV***, ***HTML***

SpringAtom

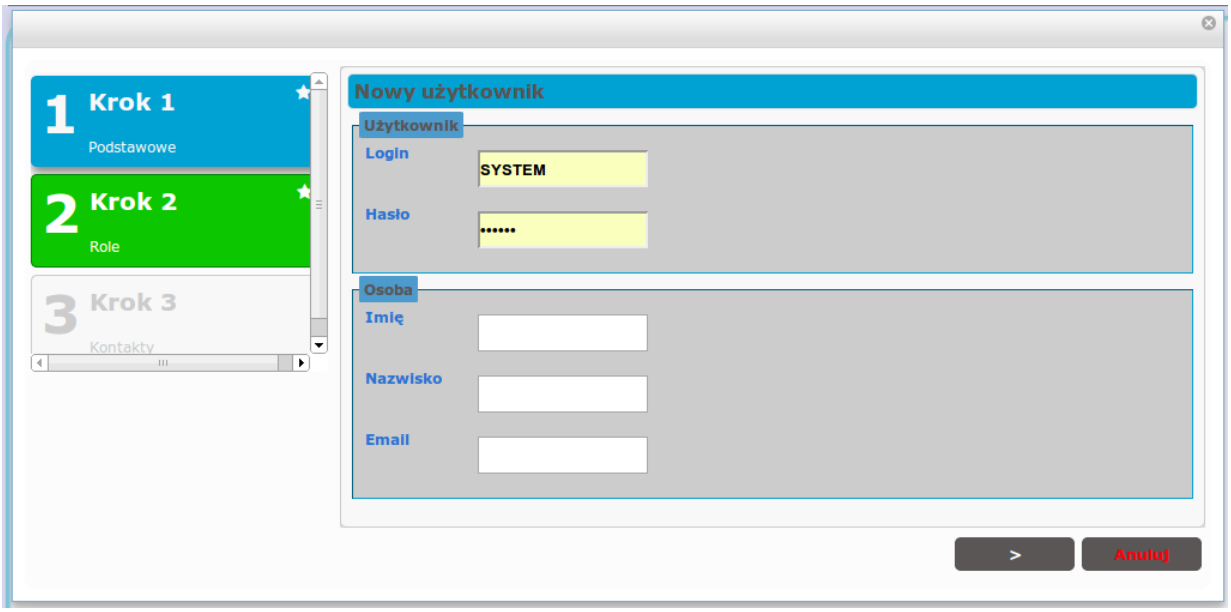
Test Test
Test Test

Kontakty	sperson.firstName	sperson.primaryMail	sperson.lastName
[]	Tomasz	kornicameister@gmail.com	Trbski
[maja.staszczyk@gmail.com, majkowy@super.pl, 781485465]	Maja	m2311007@gmail.com	Staszczyk
[]	Micha	kk@gmail.com	Florkowski
[]	Maja	aa@gmail.com	Stasczyk 2
[]	Micha	bb@gmail.com	Staszczyk

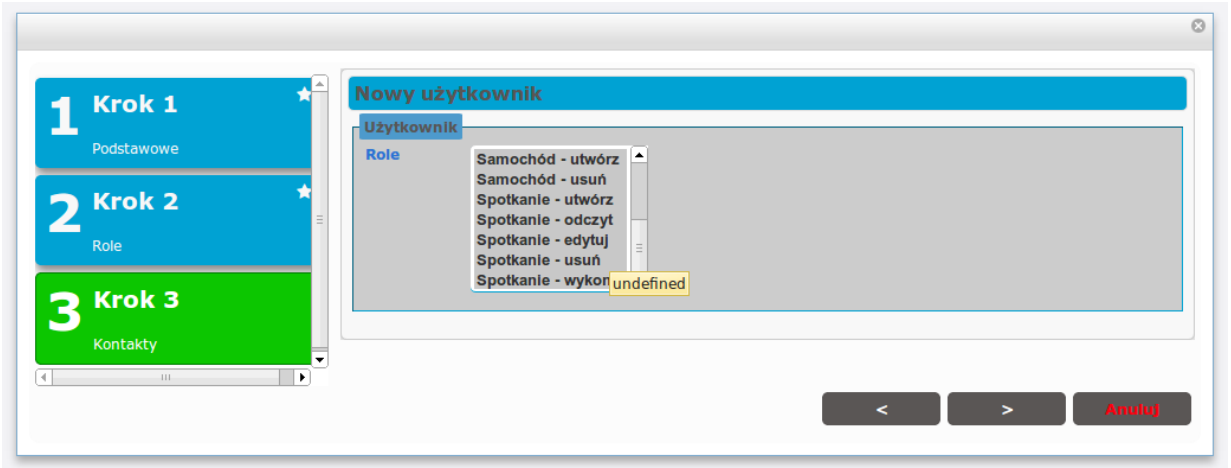
Rysunek 9: *Gotowy raport utworzony przez komponent* ***RBuilder***

Kreator nowego użytkownika

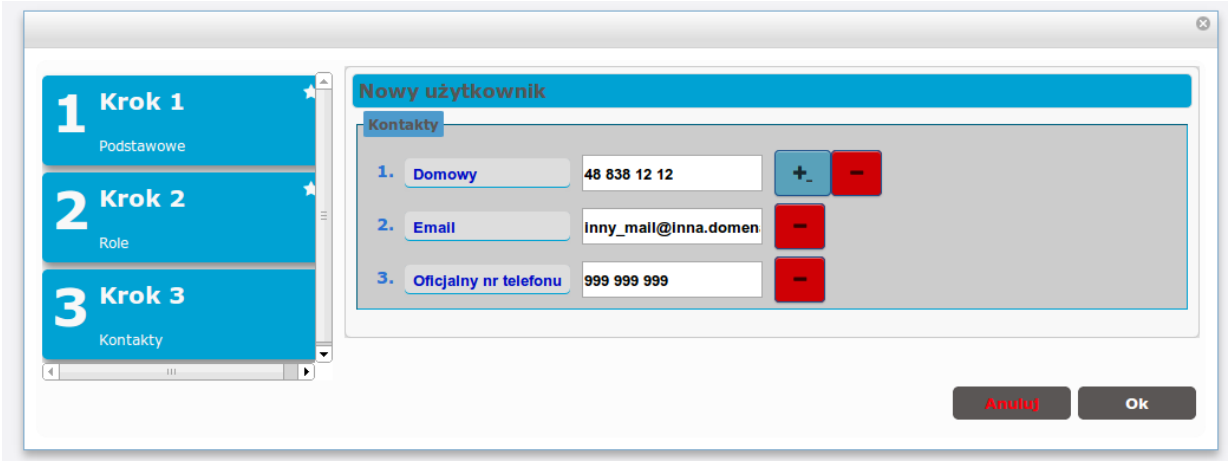
Kreator nowego użytkownika pozwala na tworzenie nowy obiektów klasy **SUser**. Kwestia uprawnień jest tutaj szczególnie ważna z uwagi na to, że w przewodniku wybierany jest zestaw ról. Role, do których przypisany jest użytkownik, stanowią późniejszą bazę do weryfikacji dostępności funkcji systemu dla poszczególnych użytkowników.



Rysunek 10: *Kreator nowego użytkownika - dane podstawowe*



Rysunek 11: *Kreator nowego użytkownika - uprawnienia użytkownika*



Rysunek 12: *Kreator nowego użytkownika - dane kontaktowe*

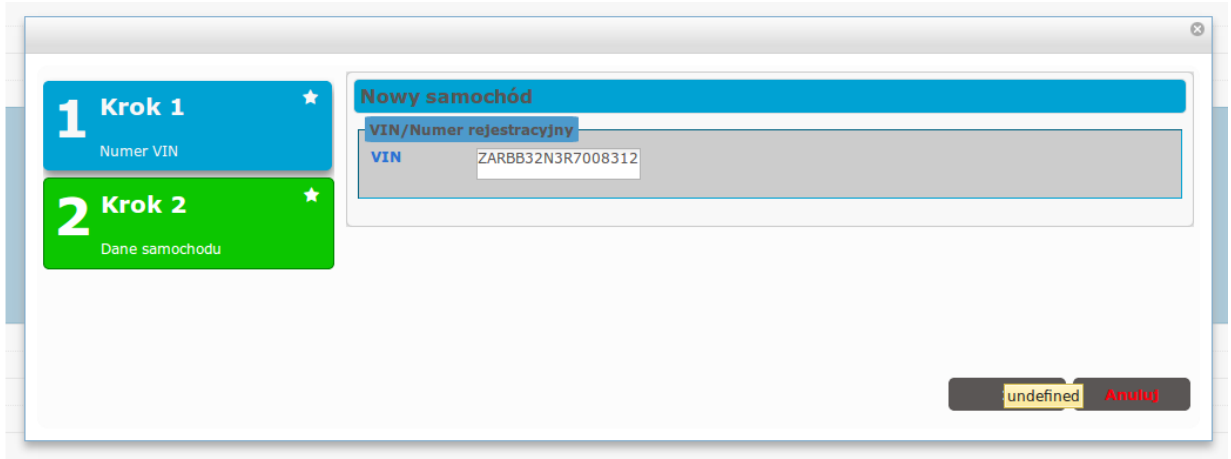
Kreator nowego samochodu

Kreator nowego samochodu został zaprojektowany aby tworzyć nowe obiekty klasy **SCar**. Pierwszym krokiem jest podanie **numeru VIN**, z którego system odczytuje wszelkie możliwe dane, które można odkodować korzystając z informacji dostępnych publicznie. Na obecną chwilę są to:

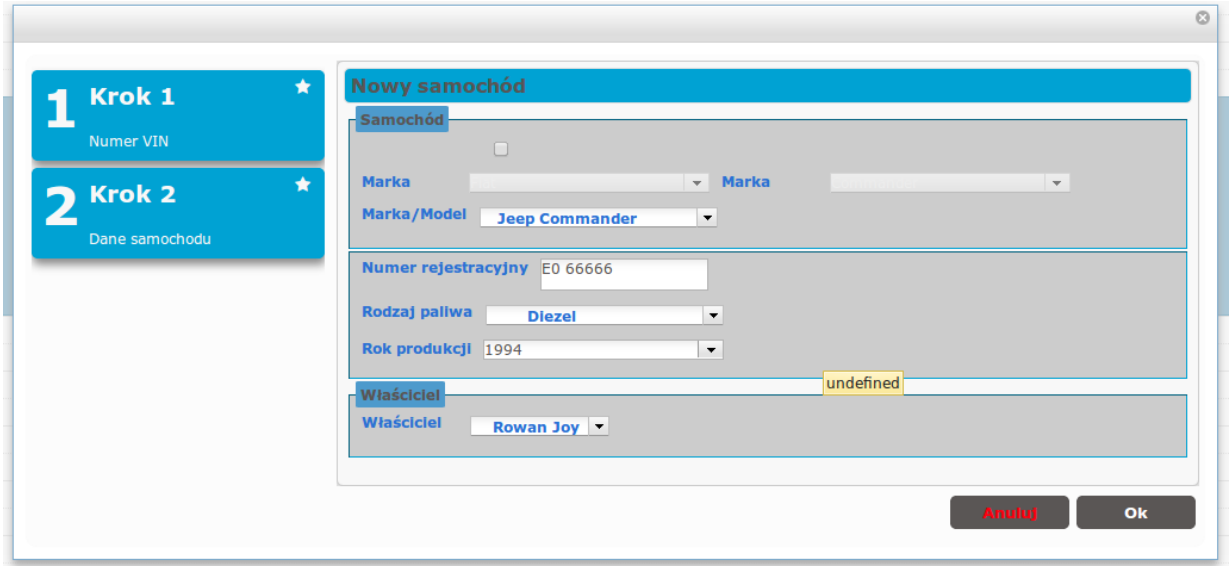
- rok produkcji, zwracany jako lista lat w których samochód mógł być wyprodukowany,
- kraj, w którym samochód został wyprodukowany.

W drugim kroku kreator nowego samochodu podaje takie informacje jak:

- markę oraz model,
- numer tablicy rejestracyjnej,
- rok produkcji,
- rodzaj paliwa,
- właściciela.



Rysunek 13: *Kreator nowego samochodu - numer VIN*

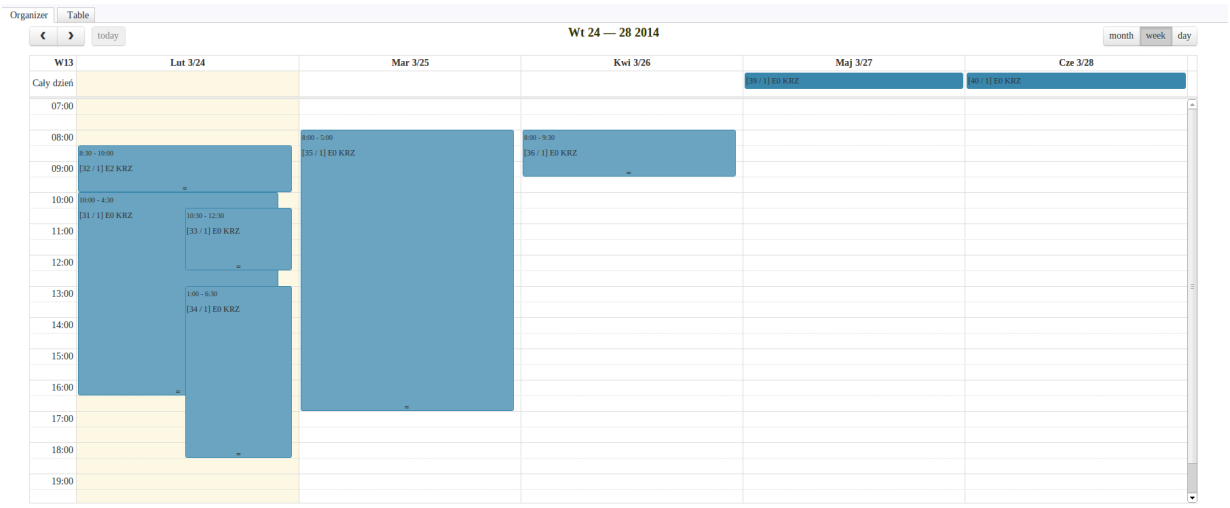


Rysunek 14: *Kreator nowego samochodu - pozostałe dane*

Organizator spotkań

Organizator spotkań jest komponentem wspierającym zarządzania tym typem obiektów. Pozwala na wgląd w listę wszystkich spotkań na 4 rożne sposoby:

- widok w kontekście wybranego dnia,
- widok w kontekście wybranego tygodnia,
- widok w kontekście wybranego miesiąca,
- widok tabelaryczny wszystkich spotkań.

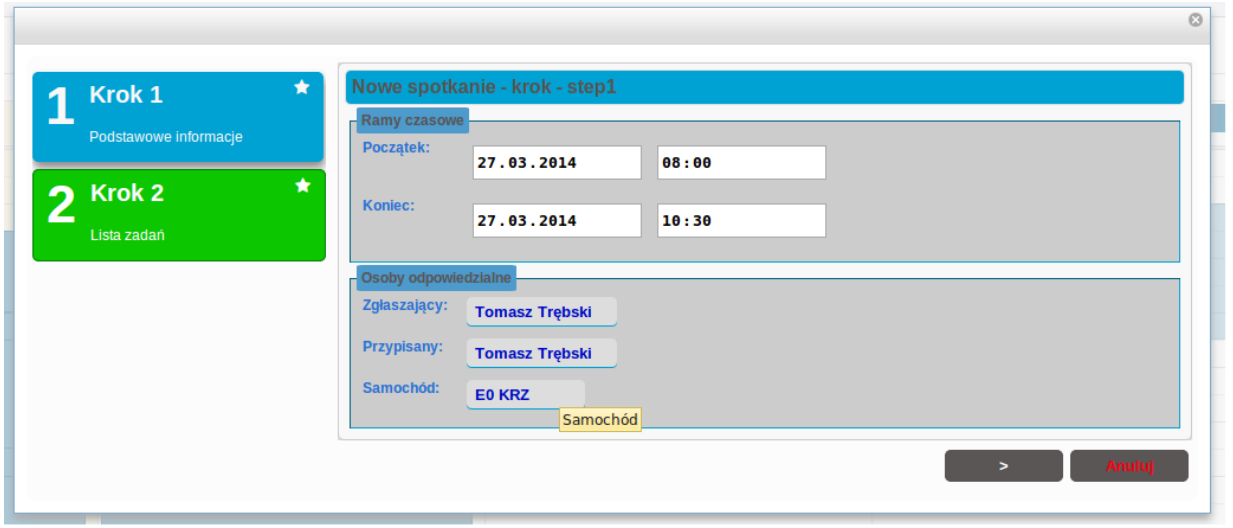


Rysunek 15: *Komponent kalendarza wspierający organizację spotkań - organizer*

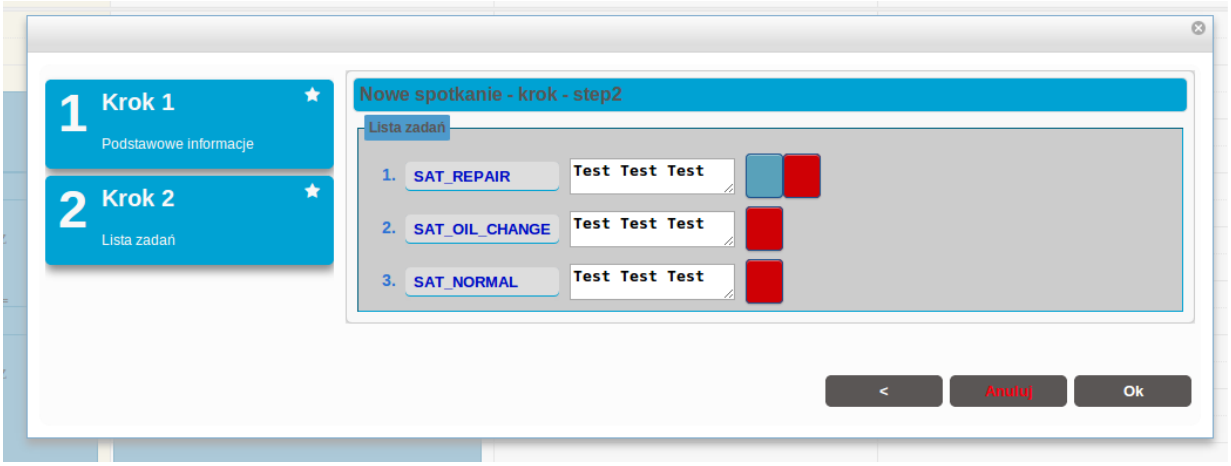
Rysunek 16: *Komponent kalendarza wspierający organizację spotkań - tabela*

Z poziomu organizera możliwe jest natomiast otwarcie strony domenowej dla wybranego spotkania, dostarczającej znacznie więcej informacji niż sam organizator oraz uruchomienie przewodnika utworzenia nowego spotkania. Ważną cechą tego przewodnika jest to, że zakres dat wybrany w organizatorze jest zachowany. Sam kreator został napisany w oparciu o technologię **Spring Web Flow**, dzięki czemu na poziomie jednego komponentu można wprowadzić dane takie jak:

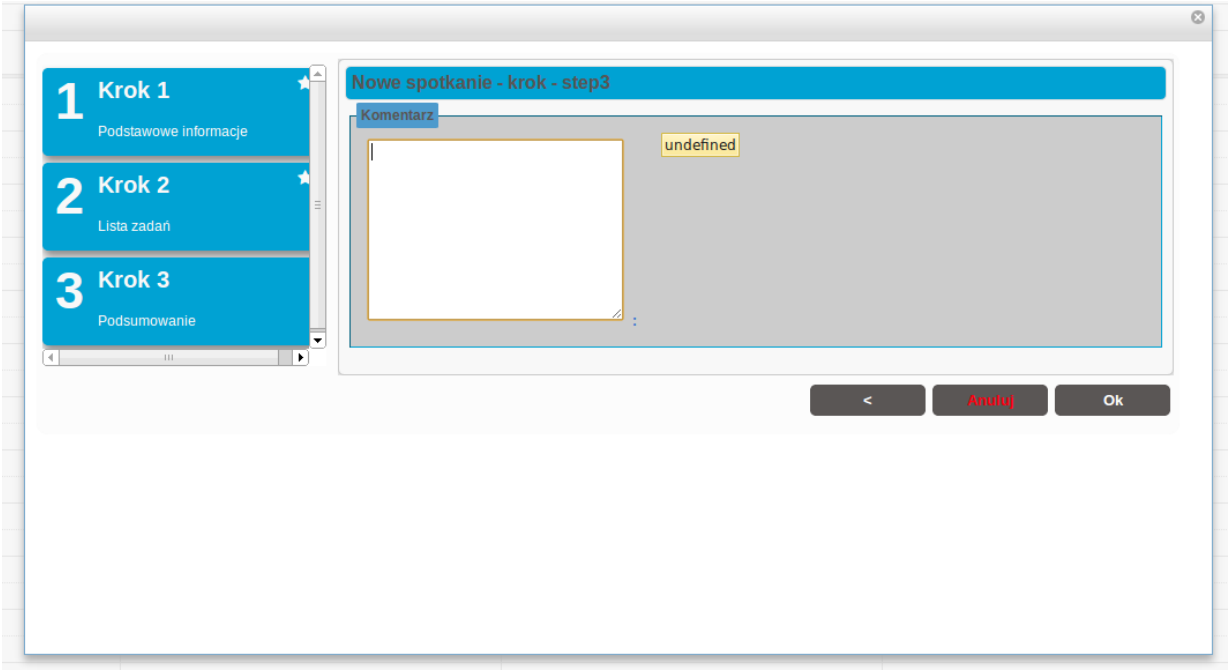
- mechanik, który będzie odpowiedzialny za wizytę,
- mechanik raportujący dane spotkanie, niemniej taką możliwość posiadają jedynie osoby uprawnione do tego,
- wybrany samochód,
- lista zadań do wykonania podczas wizyty,
- opcjonalny komentarz dla wizyty.



Rysunek 17: *Kreator nowego spotkania - krok 1*



Rysunek 18: *Kreator nowego spotkania - krok 2*



Rysunek 19: *Kreator nowego spotkania - krok 3*

Dane wejściowe są walidowane pod kątem logiki biznesowej:

- termin spotkania:
  - — podane godziny (oraz czas) muszą mieścić się w wybranym zakresie [7-21],
  - — koniec spotkania nie może nastąpić później niż jego początek,
  - — zbyt krótkie (30 minut) oraz zbyt długie (10 dni),
  - — pokrywa się z co najmniej jednym spotkaniem dla mechanika wybranego jako wykonawca
- wybrany samochód:
  - — jeśli z właścicielem samochodu powiązane są jakiekolwiek problematyczne informacje, wyświetlane jest ostrzeżenie

4.1.4. Architektura MVC

Model - Warstwa danych

Warstwa modelu danych została zaprojektowana przy użyciu adnotacji 3.4 **JPA**. Korzystanie ze standardu, stanowczo podnosi przenośność aplikacji i zwiększa możliwości modyfikacji kodu w późniejszych etapach rozwoju. Adnotacje byłyby bez znaczenie bez kolejnego szkielety aplikacji. **Hibernate** 3.5. Mimo że został on zaprojektowany do większej ilości zadań, niż jedynie zarządzania serializacją oraz deserializacją obiektów z/do bazy danych, jest wykorzystywany jedynie w tym kierunku. Dostarczona funkcjonalność do tworzenia zapytań bazodanowych niestety ma kilka wad, które w kontekście aplikacji były nie do akceptowania.

Po pierwsze tworzone kwerendy nie są typizowane, co wymusza konieczność rzutowania typów i niskopoziomowych operacji na zwracanych strukturach danych. Na dłuższą metę jest to rozwiązanie nieefektywne i generuje zbyt duża ilość zbędnego kodu, który można by zastąpić lepszym, pochodzącym z innych bibliotek. Ostatecznie problematyczne zarządzania sesją oraz błędy wynikające z przedwczesnego jej zamknięcia, generowałoby dalsze problemy w momencie serializacji danych do formatu przenośnego przez sieć internet, jak na przykład JSON. Niemniej czynnikiem decydującym był brak możliwości integracji z pozostałymi modułami szkieletu aplikacji **Spring**.

Model danych jest w niektórych przypadkach wersjonowany. Oznacza to, że dla wybranych obiektów, **Hibernate** trzyma ich historię zmian w bazie danych. Modyfikacja każdego lub jedynie wybranych pól (jest to zależne od konfiguracji danego obiektu domenowego), powoduje nie tyle zapisywanie zmian do bazy co utworzenie nowych rekordów w tabelach:

- **revinfo** - zawiera następujące kolumny:
  - — datę utworzenia rewizji,
  - — login użytkownika, który dokonał zmiany,
- **revchanges** - zawiera następujące kolumny:
  enumer rewizji, nazwę (pełna nazwa klasy) modelu,
- **nazwa_tabeli_history** - zawiera ona te pola, które został wybrane jako kandydaci do prowadzenia dziennika zmian, dla danego obiektu domenowego.

Model - Warstwa repozytoriów

Repozytoria w aplikacji demonstracyjnej są niczym więcej jak jedynie zdefiniowanymi w odpowiedni sposób interfejsami. Odpowiedni sposób oznacza, że dla każdego z nich, pierwszym interfejsem w drzewie dziedziczenia jest **Repository**. Jest to kluczowe ponieważ w ten sposób szkielet aplikacji **Spring** rozpoznaje repozytoria podczas skanowania **classpath**[4]. Programista jest zobowiązany jedynie, w przypadku chęci skorzystania, do utworzenia w swojej klasie pola typu interfejsu, a właściwa referencja do obiektu zostanie umieszczona poprzez **dependency injection**. Kod na listingu 12 pokazuję deklarację repozytorium.

[4] **Classpath scanning** - przeszukiwania wszystkich załadowanych klas

```
@Qualifier(SCarMasterRepository.REPO_NAME)
@RepositoryRestResource(itemResourceRel = SCarMasterRepository.REST_REPO_REL, path = SCarMasterRepository.REST_REPO_PA
public interface SCarMasterRepository
        extends SBasicRepository<SCarMaster, Long> {
    String REPO_NAME      = "CarMasterRepository";
    String REST_REPO_REL  = "repo.carmaster";
    String REST_REPO_PATH = "car_master";

    @RestResource(rel = "byBrandAndModel", path = "brandAndModel")
    SCarMaster findByManufacturingDataBrandAndManufacturingDataModel(
            @Param("brand") final String brand,
            @Param("model") final String model
    );

    @RestResource(rel = "byBrand", path = "brand")
    Page<SCarMaster> findByManufacturingDataBrand(
            @Param(value = "brand") final String brand,
            final Pageable pageable
    );

    @RestResource(rel = "byModel", path = "model")
    Page<SCarMaster> findByManufacturingDataModel(
            @Param(value = "model") final String model,
            final Pageable pageable
    );

    @RestResource(rel = "byBrandOrModelContaining", path = "brandOrModel_contains")
    @Query(name = "byBrandOrModelContaining", value = "select cm from SCarMaster as cm where cm.manufacturingData.brand like %
    List<SCarMaster> findByManufacturingDataBrandContainingOrManufacturingDataModelContaining(
            @Param("arg") final String searchArgument
    );
}
```

Listing 12: **SCarMasterRepository** - interfejs repozytorium pokazujący użycie metod mapowanych na kwerendy oraz bardziej skomplikowanego zapytania z użyciem adnotacji **Query**

Repozytoria zostały dostosowane do wymagań aplikacji, celem umożliwiania wyszukiwania obiektów w konkretnych rewizjach lub ich zakresach. Z tego powodu zostało wprowadzone abstrakcyjne repozytorium **SRepository**.

```
public interface SRepository<T, ID extends Serializable, N extends Number & Comparable<N>>
        extends SBasicRepository<T, ID>,
                RevisionRepository<T, ID, N> {
    /**
     * {@code findInRevision} returns {@link Revision} of the target underlying target entity in the given revision
     * described in {@code revision} param
     *
     * @param id
     *          id of the entity {@link org.springframework.data.domain.Persistable#getId()}
     * @param revision
     *          the revision number
     *
     * @return {@link Revision} for passed arguments
     *
     * @throws EntityInRevisionDoesNotExists
     * @see SRepository#findInRevisions(java.io.Serializable, Number[])
     */
    Revision<N, T> findInRevision(final ID id, final N revision);

    /**
     * {@code findInRevisions} does exactly the same job but for multiple possible {@code revisions}.
     *
     * @param id
     *          id of the entity {@link org.springframework.data.domain.Persistable#getId()}
     * @param revisions
     *          varargs with revision numbers
     *
     * @return {@link Revisions}
     */
    @SuppressWarnings("unchecked")
    Revisions<N, T> findInRevisions(final ID id, final N... revisions);

    Revisions<N, T> findRevisions(final ID id, final DateTime dateTime, final Operators before);

    /**
     * Returns how many revisions exists for given {@link org.springframework.data.domain.Persistable#getId()} instance
     *
     * @param id
     *          the id
     *
     * @return revisions amount
     */
    long countRevisions(ID id);
}
```

Listing 13: **SRepository** - abstrakcyjne repozytorium wspierające dostęp do rewizji

Z uwagi na to, że funkcjonalność tego interfejsu nie jest dostępna w klasach dostarczonych przez **Spring Data JPA**, musiałaby być ona zaimplementowana. Niemniej nie odniosło by to skutku bez wprowadzenie dodatkowo elementu - fabryki, która nadpisywała proces tworzenia repozytoriów. Poniższa klasa **SRepositoriesFactoryBean** działa wybiórczo, tj. dobiera konkretne implementacje, zależnie od faktycznego typu repozytorium. Rozstrzygającym kryterium jest to, czy dane repozytorium jest przeznaczone dla obiektów wersjonowanych.

```
public class SRepositoriesFactoryBean<T extends SBasicRepository<S, ID>, S, ID extends Serializable>
        extends JpaRepositoryFactoryBean<T, S, ID> {

    private Class<?> revisionEntityClass;

    public void setRevisionEntityClass(Class<?> revisionEntityClass) {
        this.revisionEntityClass = revisionEntityClass;
    }

    @Override
    protected RepositoryFactorySupport createRepositoryFactory(final EntityManager entityManager) {
        return new SRepositoryFactory(entityManager, this.revisionEntityClass);
    }

    private static class SRepositoryFactory
            extends JpaRepositoryFactory {
        private static final Logger LOGGER = Logger.getLogger(SRepositoryFactory.class);
        private final Class<?>                      revisionEntityClass;
        private final RevisionEntityInformation     revisionEntityInformation;
        private final LockModeRepositoryPostProcessor lockModePostProcessor;

        public SRepositoryFactory(final EntityManager entityManager,
                                  final Class<?> revisionEntityClass) {
            super(entityManager);
            this.lockModePostProcessor = LockModeRepositoryPostProcessor.INSTANCE;

            this.revisionEntityClass = revisionEntityClass == null ? AuditedRevisionEntity.class : revisionEntityClass;
            this.revisionEntityInformation = DefaultRevisionEntity.class
                    .equals(revisionEntityClass) ? new AuditingRevisionEntityInformation() : new ReflectionRevisionEntityInformation(this.revisio

            if (LOGGER.isTraceEnabled()) {
                LOGGER.trace(String.format("Created %s with arguments=[em=%s,rec=%s,rei=%s]",
                        SRepositoryFactory.class.getSimpleName(),
                        entityManager,
                        this.revisionEntityClass,
                        this.revisionEntityInformation));
            }
        }

        @Override
        @SuppressWarnings("unchecked")
        protected <T, ID extends Serializable> JpaRepository<?, ?> getTargetRepository(final RepositoryMetadata metadata, final EntityMar
            final JpaEntityInformation<T, Serializable> entityInformation = (JpaEntityInformation<T, Serializable>) getEntityInformation(me
                    .getDomainType());
            final Class<?> repositoryInterface = metadata.getRepositoryInterface();
            SimpleJpaRepository<T, ID> repository;
            if (!ClassUtils.isAssignable(SRepository.class, repositoryInterface)) {
                repository = (SimpleJpaRepository<T, ID>) new SBasicRepositoryImpl<>(entityInformation, entityManager);
            } else {
                repository = (SimpleJpaRepository<T, ID>) new SRepositoryImpl<>(entityInformation, revisionEntityInformation, entityManag
            }
            repository.setLockMetadataProvider(this.lockModePostProcessor.getLockMetadataProvider());
            return repository;
        }

        @Override
        protected Class<?> getRepositoryBaseClass(final RepositoryMetadata metadata) {
            final Class<?> repositoryInterface = metadata.getRepositoryInterface();
            if (!SRepository.class.isAssignableFrom(repositoryInterface)) {
                return SBasicRepositoryImpl.class;
            }
            return SRepositoryImpl.class;
        }
    }
}
```

Listing 14: **SRepositoriesFactoryBean** - fabryka repozytoriów dla implementacji własnej funkcjonalności

Model - Warstwa serwisów

Serwisy stanowią w aplikacji element realizujący logikę biznesową. Nie odnoszą się one jednak jedynie do operacji na modelu danych. Również inne moduły aplikacji korzystają z własnych serwisów, jako miejsc gdzie funkcjonalność została zebrana i jest gotowa do użycia.

```
public interface SPersonService
      extends SService<SPerson, Long, Integer> {

   SContact<SPerson> newContactData(
         @NotNull
         final String contact,
         final long assignTo,
         @NotNull
         final SContact assignToContact) throws EntityDoesNotExistsServiceException;

   List<SPersonContact> findAllContacts(final Long idClient);

   List<SPerson> findByFirstName(
         @NotNull
         final String firstName);

   List<SPerson> findByLastName(
         @NotNull
         final String lastName);

   SPerson findByEmail(
         @NotNull
         final String email);
}
```

Listing 15: **SPersonService** - interfejs serwisu dla modelu SPerson

Serwisy zostały zaprojektowane aby korzystać z repozytoriów danych. Ma to swoje pozytywne skutki i nie jest wcale oznakę nadmiarowości kodu, czy też jego duplikowania. Z uwagi na fakt, ze serwisy odwołują się do danych poprzez interfejsy repozytorii, podnosi to znacząco możliwości późniejszych zmian w postaci silnika bazy danych. Dodatkową korzyścią jest zwiększenie możliwości testowania interesujących funkcji bez konieczności posiadania działającego połączenia z bazą danych.

Widok - Warstwa widoku

Warstwa widoku została zaprojektowana z wykorzystaniem standardowej biblioteki tagów **JSTL**, plików **JSP** oraz biblioteki **Apache Tiles** 3.9. Dzięki **Tiles** udało się zminimalizować zbędny kod w plikach **JSP** definiujący elementy, takie jak nagłówek strony, element *<head>*. Do gotowego widoku można się było następnie odwoływać po unikatowej nazwie.

Listing 16 pokazuje deklarację abstrakcyjnej *płytki*. Abstrakcyjność jest tutaj kwestią umowną ponieważ można by tę *płytkę* zwrócić z dowolnego kontrolera i została by ona zrenderowana do poprawnego widoku HTML. Elementem, który jest w tej definicji zadeklarowany, lecz nie zdefiniowany jest *content* - faktyczna zawartość danej strony. Mimo to plik **JSP** odpowiadający tej płytce zawiera kod, który umieści ją w ostatecznej strukturze DOM.

```
<tiles-definitions>

   <definition name="ui.core.Page" template="/ui/core/page.jsp">
      <put-attribute name="head" value="ui.meta.Head"/>
      <put-attribute name="css" value="ui.meta.head.CSS"/>
      <put-attribute name="js" value="ui.meta.head.JS"/>
      <put-attribute name="navigator" value="ui.nav.Navigator"/>
      <put-attribute name="header" value="/ui/core/c-header.jsp"/>
      <!-- to override by concrete page -->
      <put-attribute name="content" value=""/>
      <!-- to override by concrete page -->
   </definition>

</tiles-definitions>
```

Listing 16: Definicja *tile* - podstawowy element widoku w rozumieniu technologi Apache Tiles

```
<%@ page session="true" language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html>
<head>
   <tiles:insertAttribute name="head"/>
   <tiles:insertAttribute name="css" flush="true"/>
   <tiles:insertAttribute name="js" flush="true"/>
</head>
<body class="tundra">
<div id="page">
   <div class="content-wrapper">
   <header id="header" class="main">
         <s:message code="label.dashboard.header" htmlEscape="true" var="headerLabel"/>
         <p>${headerLabel}</p>

         <div id="authentication-control">
            <security:authorize access="isFullyAuthenticated()" var="userAuthenticated"/>
            <c:choose>
               <c:when test="${!userAuthenticated}">
                  <tiles:insertDefinition name="springatom.tiles.auth.action.login" flush="true"/>
               </c:when>
               <c:otherwise>
                  <tiles:insertDefinition name="springatom.tiles.auth.action.logout" flush="true"/>
               </c:otherwise>
            </c:choose>
         </div>
      </header>
      <tiles:insertAttribute name="header"/>
      <div class="content">
         <tiles:insertAttribute name="content"/>
      </div>
      <footer id="footer" class="main">
         <p>Footer goes here</p>
      </footer>
   </div>
   <tiles:insertAttribute name="navigator"/>
</div>
</body>
```

Listing 17: Plik **JSP odpowiadający płytce 16**

Globalna - Warstwa konwersji

Warstwa konwersji typów jest modułem **Spring**, który wykorzystywany jest w momencie, kiedy z obiektu typu A programista chce uzyskać obiekt typu B. Dobrym przykładem jest najczęściej moment, w którym w kodzie JSP umieszczamy bezpośrednio nasz obiekt, korzystając z biblioteki tagów dostarczonej przez **Spring**. W tym momencie uruchamiany jest proces, mający na celu konwersję obiektu do reprezentatywnej postaci łańcucha znakowego, który można będzie wkomponować do drzewa DOM. Jest to rozwiązanie efektywne, niemniej posiadające jedno uchybienie. W przypadku, gdy programista chciałby uzyskać selektywny sposób, zależny od kontekstu, w jakim znajduje się obiekt lub też chęci uzyskania wartości jednego z atrybutów, nie jest on w stanie osiągnąć zamierzonego rezultatu, z uwagi na sposób, w jaki działa konwersja typów. Znalezienie pierwszego konwertera, który jest w stanie przeprowadzić żądaną transformację kończy proces wyszukiwania. Nie oznacza to wcale, że uzyskany wynik będzie zgodny z oczekiwanym. Z tego powodu aplikacja demonstracyjna rozszerza istniejącą funkcjonalność przez umożliwienie wybiórczego konwertowania między poszczególnymi typami. Na obecną chwilę zostało to zaimplementowane dla obiektów domenowych, a klasą kontrolującą selektywny proces jest **PersistableConverterPicker**. **PersistableConverterPicker** posiada metody które po-

```
public <T extends Persistable> Converter<T, String> getConverterForSelector(final String key) {
        final Optional<PersistableConverter<?>> match = FluentIterable.from(this.converters).firstMatch(new Predicate<PersistableCo
                @Override
                public boolean apply(@Nullable final PersistableConverter<?> input) {
                        assert input != null;
                        final PersistableConverterUtility annotation = input.getClass().getAnnotation(PersistableConverterUtility.class);
                        return annotation != null && AnnotationUtils.getValue(annotation, "selector").equals(key);
                }
        });
        if (match.isPresent()) {
                return (Converter<T, String>) match.get();
        }
        return new DefaultPickedConverter<>();
}

@SuppressWarnings("unchecked")
public <T extends Persistable> Converter<T, String> getDefaultConverter(final TypeDescriptor sourceType) {
        final Optional<PersistableConverter<?>> match = FluentIterable
```

Listing 18: **PersistableConverterPicker** - koordynator selektywnej konwersji typów

zwalają na wybór selektywnego konwertera do wykonania operacji transformacji. Zostało również zapewnione wsparcia dla istniejącej funkcjonalności, bez użycia selektora. Ta część realizowana jest w metodzie **getDefaultConverter(...)**.

Selektywne konwertery różnią się od normalnych jedynie użyciem specjalnej adnotacji, która pozwala na ustalenie, że:

- konwerter jest domyślny, jeśli nie został zdefiniowany klucz,
- konwerter jest selektywny, jeśli istnieje zdefiniowany klucz.

```
                        @Override
                        public boolean apply(@Nullable final PersistableConverter<?> input) {
                                assert input != null;
                                return input.matches(sourceType, TypeDescriptor.valueOf(String.class));
                        }
                })
                .firstMatch(new Predicate<PersistableConverter<?>>() {
                        @Override
                        public boolean apply(@Nullable final PersistableConverter<?> input) {
                                assert input != null;
                                final PersistableConverterUtility annotation = input.getClass().getAnnotation(PersistableConverterUtility
                                return annotation != null && String.valueOf(AnnotationUtils.getValue(annotation, "selector")).isEmpty(
                        }
                });
        if (match.isPresent()) {
                return (Converter<T, String>) match.get();
        }
        return new DefaultPickedConverter<>();
}

private class DefaultPickedConverter<T extends Persistable>
                extends PersistableConverterImpl<T> {

        @Override
        public boolean matches(final TypeDescriptor sourceType, final TypeDescriptor targetType) {
                return true;
        }

        @Override
        public String convert(final Persistable source) {
                if (source == null) {
```

Listing 19: **PersistableConverterPicker** - pobranie domyślnego konwertera dla typu

4.1.5. Plany rozwojowe

Aplikacja demonstracyjna jest obecnie na etapie dalszego rozwoju. Posiada szeroko zdefiniowane moduły zaprojektowane, aby wspierać takie rejony jak:

- generyczna warstwa operacji bazodanowych,
- warstwa logiki biznesowej udostępnionej przez serwisy,
- moduł komponentów dla budowy tabel oraz stron obiektów domenowych,
- selektywna warstwa konwersji,
- tagi oddelegowane dla przewodników opartych o Spring Web Flow,
- model danych wraz z jego abstrakcyjną warstwą (interfejsy) do użytku zewnętrznego,
- moduł obsługujący funkcjonalność raportowania.

W większości przypadków uzyskana funkcjonalność jest jednak na etapie implementacji i niektóre z niedopracowanych elementów ulegną zmianie, celem uproszczenia zarządzania złożonością projektu oraz usunięcia nadmiarowych klas oraz słabo konfigurowalnych części. Plany rozwojowe aplikacji zostały przedstawione w tabeli 3.

Tabela 3: Plany rozwojowe aplikacji demonstracyjnej

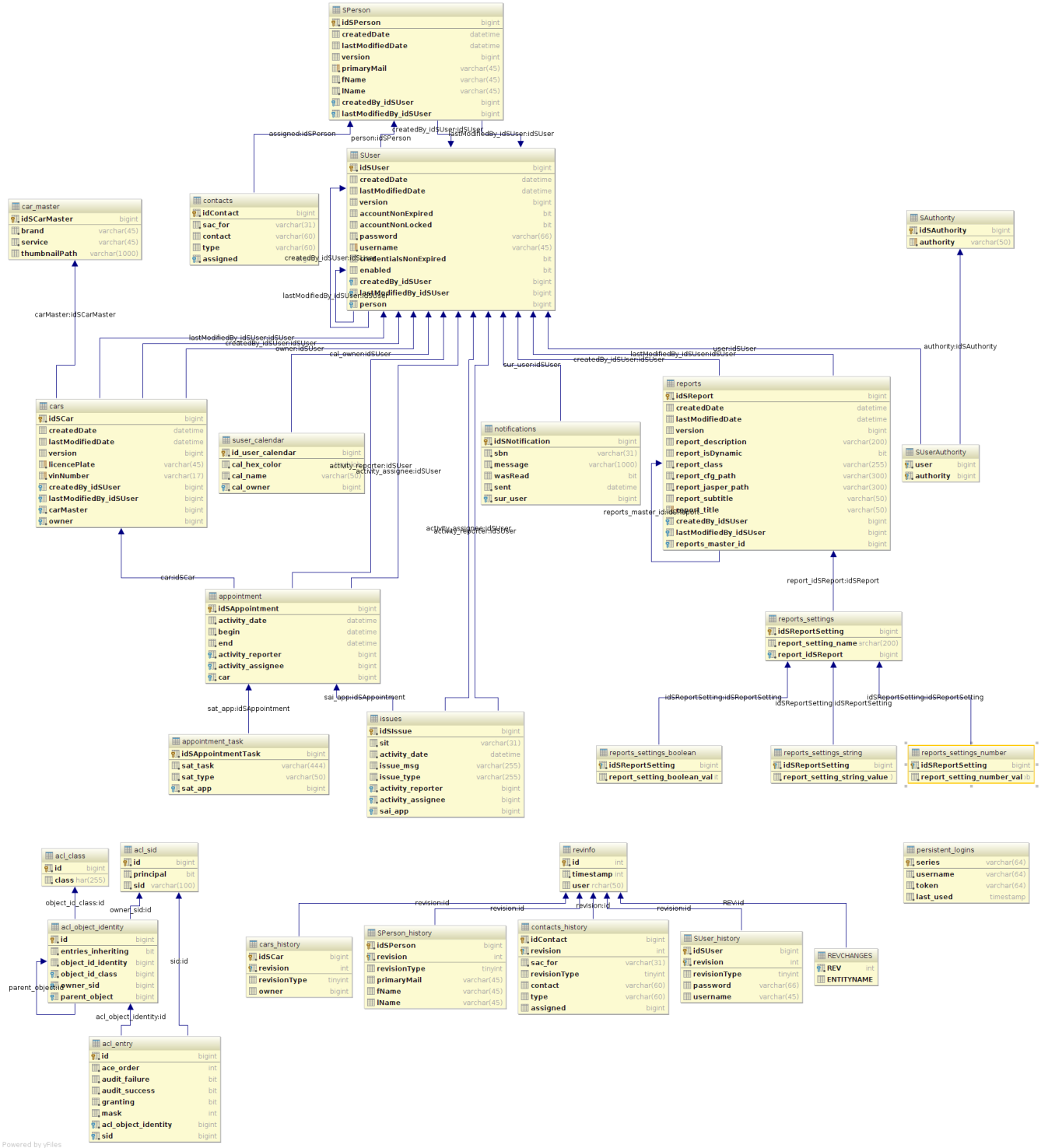
LP	Moduł	Opis zmian
1	**ComponentBuilder**	• optymalizacja kodu oraz wsparcie dla cache'owania raz załadowanych • przeniesienie definicji stron do deklaratywnego języka XML. Będzie to możliwe po usprawnieniu działania selektywnych konwerterów oraz zwiększy możliwość zmian zgodnie z wymaganiami, bez konieczności zmian w plikach Java, • prawdopodobne przeprojektowanie operacji ładowania strony obiektu domenowego. Na obecną chwilę ładowany jest gotowy widok, jednakże z uwagi na możliwie duży narzut przesłanych danych bardziej optymalnym rozwiązaniem jest załadowanie pojedynczo każdego z panelu zdefiniowanego na danej stronie, • przeniesienie kodu widoku stron domenowych do łatwiejszego w zarządzaniu oraz utrzymaniu kodu ExtJS
2	**ComponentBuilder**	Połączenie kontrolerów odpowiedzialnych za obsługę żądań pochodzących od komponentów typu **tabele** oraz **strony domenowe**. Celem tego działania jest ujednolicenie adresów sugerujące, że oba komponenty służą podobnemu celowi.
3	**ComponentBuilder**	Wprowadzenie automatycznego mechanizmu generującego przekierowania do stron obiektów domenowych
4	**WebFlow**	Zaprojektowanie biblioteki wspierającej funkcjonalność **Spring Web Flow** dla biblioteki ExtJS. Decyzja podyktowana jest chęcią zminimalizowania użycia różnorodnych bibliotek JavaScript. Gotowa biblioteka byłaby udostępniona na licencji OpenSource.
5	**Selektywne konwertery**	Rozszerzenie możliwości selektywnych konwerterów o obiekty inne niż domenowe oraz wsparcie dla możliwości definiowana powtarzających się selektorów - kluczy w kontekście globalnym, ale unikatowych w kontekście danego obiektu podlegającego konwersji.
6	**Kod ogólny**	Usunięcie pozostałości nadmiarowego kodu, który pozostał po uaktualnieniu aplikacji, do korzystania z najnowszej wersji (4.0.0) szkieletu aplikacji **Spring**.
	Następna strona...	

Tabela 3 – kontynuacja...

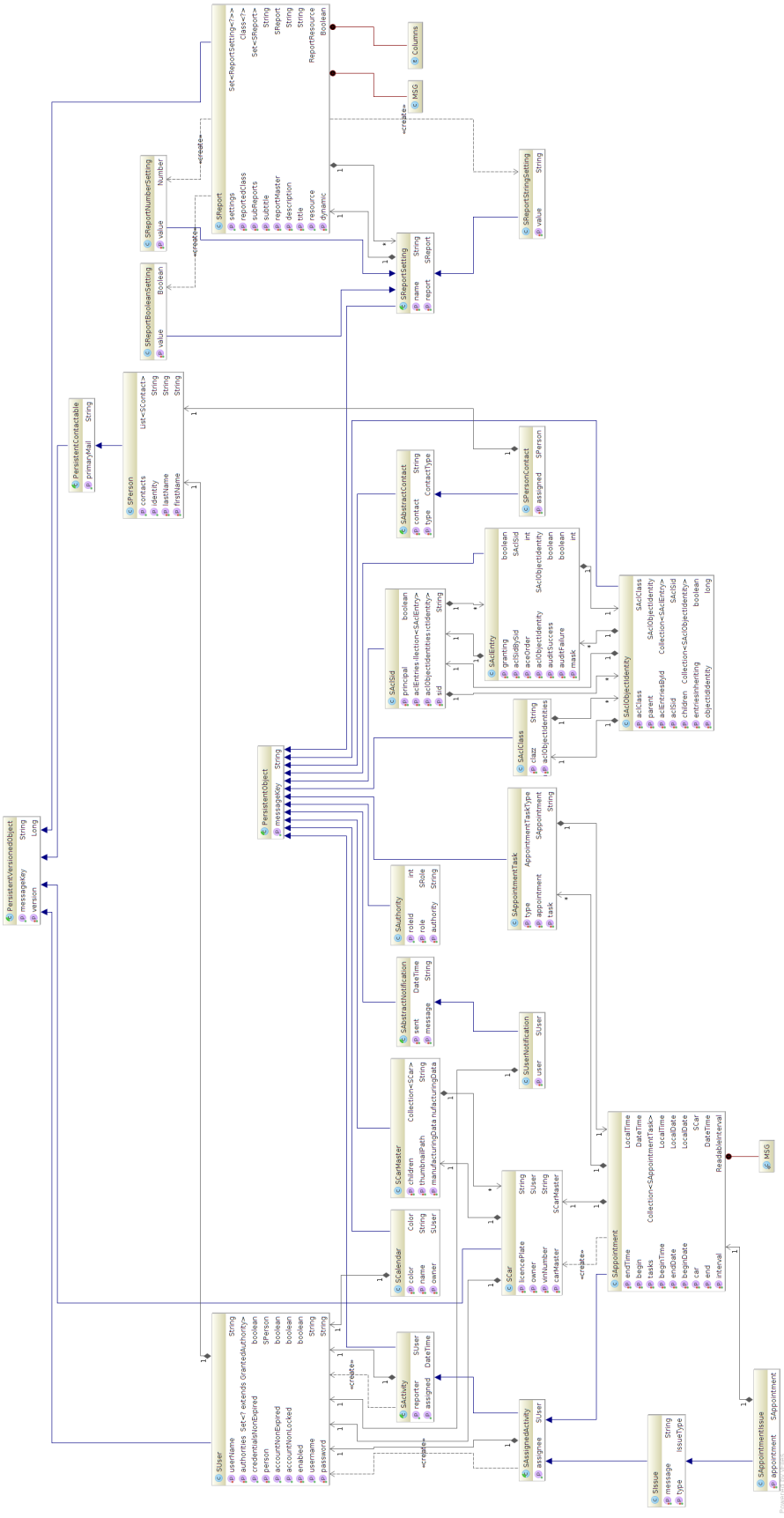
LP	Moduł	Opis zmian
7	**Widok**	Wsparcia dla ExtJS - migracja istniejących komponentów warstwy widoku użytkownika do szkieletu aplikacji ExtJS. Prawdopodobnym wyborem będzie wykorzystanie ExtJS w wersji 4.x.x z uwagi na lepszą wydajność i większe możliwości biblioteki.
8	**ReportBuilder**	Zrezygnowania z ciężkiego w użyciu oraz utrzymaniu sposobu generowania raportów w aplikacji poprzez bibliotekę **DynamicJasper**. Przeniesienie bezpośredniego generowania raportów do tabelek oraz inwestygacja możliwych technologii do wykorzystania w przypadku eksportowania raportu do formatów takich jak HTML, CSV, PDF oraz XLS.
9	**Przewodniki**	Ukończenie wszystkich wymaganych kreatorów służących zarówno do modyfikacji, jak i tworzenia nowych obiektów domenowych
10	**Terminarz spotkań**	Ukończenie prac nad terminarzem spotkań. Obecna funkcjonalność pozwala na tworzenie obiektów, które następnie można przeglądać z użyciem kalendarza (podobny do używanego w programie Outlook).
11	**Dekodowane numeru VIN**	Rozszerzenie obecnych możliwości dekodera numeru VIN o pobieranie informacji, takich jak: • wytwórca samochodu, • marka oraz model, • typ nadwozia, • typ paliwa, • typ silnika

4.2. Diagramy UML, schemat bazy danych

4.2.1. Schemat bazy danych

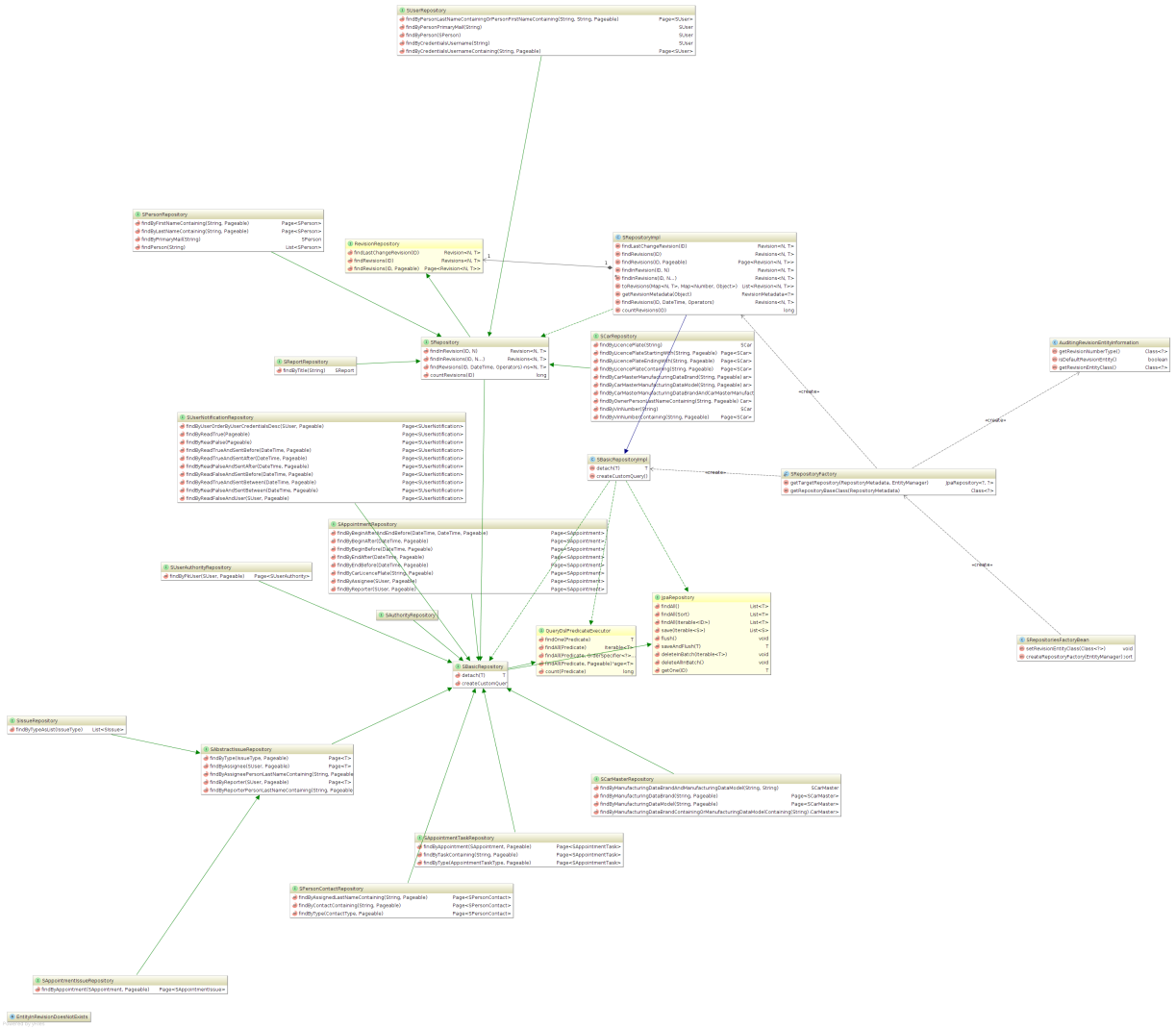


Rysunek 20: *Schemat bazy danych aplikacji demonstracyjnej*



Rysunek 21: *Obiektowy model danych odpowiadający który odpowiada schematowi bazy danych*

4.2.2. Schemat warstwy repozytoriów



Rysunek 22: *Schemat repozytoriów aplikacji demonstracyjnej*

Add the rest of UMLs

4.3. Metryki kodu

Kod aplikacji jest zbiorem funkcjonalności zaimplementowanych zarówno dla części serwerowej, jak i klienckiej. Z tego powodu liczba linii kodu została podzielona na odpowiednie grupy, zgodne z językami użytymi do stworzenia aplikacji demonstracyjnej.

4.3.1. Liczba linii kodu aplikacji

Tabela 4: Liczba linii kodu według języka programowania

Język	LOC	LOC (SC)	LOC (C)	LOC (EL)
Java	29555	16628	9136	3791
JS	1044	?	?	?
JSP	2122	?	?	?
Razem	32721	16628	9136	3791

LOC	Całkowita liczba linii kodu
LOC (SC)	Liczba linii kodu źródłowego
LOC (C)	Liczba linii komentarzy
LOC (EL)	Liczba linii pustych

4.3.2. Java

Tabela 4 obrazuje złożoności projektu składającego się z 5 głównych modułów:

- **AOP** - funkcjonalność opierająca się o *Aspect Oriented Programming* zakresem obejmująca całą aplikację,
- **Core** - zbiór artefaktów (klas, klas abstrakcyjnych, interfejsów oraz typów wyliczeniowych) przeznaczonych do wykorzystania,
- **Server** - zadaniem klas tego modułu jest dostarczenie modelu danych, wsparcia jego walidacji oraz wersjonowania, definicji interfejsów repozytoriów, serwisów odpowiedzialnych za logikę biznesową,
- **Web** - artefakty tego modułu oferują definicje obiektów opisujących strony domenowe, tabele, przewodniki, raporty. Zawierają również klasy odpowiedzialne za dynamiczny i elastyczny model akcji.
- **WebMVC**

Strukturalne metryki kodu

Tabela 5: Liczba klas / Liczba linii kodu modułów

Moduł	LK	LOC	CD	D	TD
Aop	3/3	192/192	0	0	0
Core	13/1.86	628/89.71	0	0.25	0.25
Server	187/3.07	10806/183.15	0.82	3.32	23.48
Web	188/2.89	10803/166.20	0.3	3.28	14.01
WebMVC	37/2.85	2262/174	0	4.27	37.73

LK	Liczba klas/Średnia liczba klas
LOC	Liczba linii kodu/Średnia liczba linii kodu
D	Średnia liczba cyklicznych zależności
CD	Średnia liczba zależności
TD	Średnia liczba zależności przechodnich

Metryka Chidamber-Kemerer

Zadaniem metryki jest analiza następujących właściwości kodu [5]:

- **WMC** - liczba metod zdefiniowanych w klasie,
- **DIT** - głębokość drzewa dziedziczenia,
- **SUB** - liczba bezpośrednich potomków w hierarchii dziedziczenia,
- **CBO** - stopień zależności od pozostałych artefaktów,
- **RFC** - ilość metod obiektu danej klasy, które mogę być wywołane w odpowiedzi na wywołania jednej metody tej klasy,
- **LCOM** - współczynnik kohezji, im wyższy, tym większa jest zależność między poszczególnymi artefaktami

Tabela 6: Metryka Chidamber - Kemerer

Moduł	CBO	DIT	LCOM	RFC	SUB	WMC
Aop	0	1.00	2.33	12.00	0	6
Core	2.25	1.50	1.00	34.60	0.62	5.12
Server	6.50	2.35	1.64	282.30	0.77	7.16
Web	5.78	2.03	1.74	194.33	0.64	7.47
WebMVC	5.00	2.94	1.33	213.22	0.18	5.03
Razem	5.78	2.22	1.65	222.44	0.62	6.98

Na uwagę zasługują w tym miejscu niskie wartości takich współczynników jak **DIT**, gdzie średnia wartość nie przekroczyła wartości 3, począwszy od korzenia wszystkich klas definiowanych w języku Java - **Object**. Przyjmuje się, że wartość graniczna dla większości aplikacji wynosi 5. Niemniej wartość średnia nie oddaje pojedynczych przypadków nadużyć. Większość takich sytuacji, występujących aplikacji demonstracyjnej, gdzie przekroczono graniczną wartość, odnosi się do klas rozszerzających standardowe możliwości szkieletu aplikacji, celem dostosowania ich do konkretnych przypadków użycia. Ponadto ważna jest głębokość drzewa dziedziczenia, przekraczająca przyjętą wartość w klasach opisujących biznesowy model danych. Fakt ten można pominąć z uwagi na to, że wspomniane artefakty służą wsparciu dla dziedziczenia wspólnych atrybutów dla konkretnych gałęzi klas oraz tym, że są to w klasy definiujące, prócz wspomnianych już pól, metody dostępowe, popularnie nazywane **gettery** oraz **settery**. W nielicznych przypadkach część funkcjonalności biznesowej została zamknięta w obiektach domenowych z uwagi na rozbieżność w sposobie przechowywania danych, a tym, jak są one udostępniane innym klasom.

W tym miejscu warto wspomnieć o wartości jaką uzyskano dla wskaźnika **SUB**, który jest blisko związany z poprzednio omawianym **DIT**. Podczas gdy **DIT** opisuje głębokość drzewa dziedziczenia, co przekłada się na zwiększenie zarówno ilości atrybutów, jak i metod będących kandydatami do ponownego wykorzystania (nadpisania), **SUP** odnosi się do szerokości drzewa dziedziczenia, czyli ilości dzieci będących bezpośrednimi potomkami analizowanej klasy. Przyjęto, że niska wartość **DIT** jest zdecydowanie lepsza od **SUB**. Tak też jest w przypadku aplikacji demonstracyjnej, gdzie wartości tych dwóch współ-

czynników charakteryzują się następującymi wartościami **DIT** równe 2.22, a **SUB** - 0.62. Widać wyraźnie, że zdecydowana większość klas definiuje swoją rolę poprzez mechanizm polimorfizmu.

Największym problem aplikacji okazała się wysoka wartość współczynnika **RFC**. Im jest ona wyższa, tym bardziej aplikacja narażona jest na błędy, a istniejąca złożoność utrudnia zrozumienie oraz testowanie aplikacji.

Metryka MOOD

Metryki **MOOD** zostały zaprojektowane do mierzenia jakości aplikacji realizowanych z użyciem technik programowania obiektowego [1]. Analiza projektu nimi jest szczególnie użyteczna dla obszernych projektów. Duża ilość klas oraz istniejące plany rozwojowe sugerujące dalszy wzrost ilości linii kodu sprawiają, że stanowi ona cenna źródło wiedzy o strukturze programu i możliwości poprawienia rejonów szczególnie ważnych w kontekście właściwego wykorzystania paradygmatu programowania obiektowego.

Tabela 7: Metryka MOOD

AHF	**AIF**	**CF**	**MHF**	**MIF**	**PF**
100.0%	0.00%	0.00%	53.85%	0.00%	100.0%

AHF	Współczynnik enkapsulkacji pól klas
AIF	Współczynnik dziedziczenia atrybutów
CF	Współczynnik powiązań
MHF	Współczynnik enkapsulkacji metod
MIF	Współczynnik dziedziczenia metod
PF	Współczynnik polimorfizmu

Na istotną uwagę zasługują dwa wskaźniki **PF=100%** oraz **MHF=53.85%**. Pierwszy z wyników odnosi się do polimorfizmu. Wynik z pewnością wskazuje na dobry projekt systemu wykazującego duży poziom abstrakcyjność, co usprawnia późniejsze modyfikacje na poziomie zmiany sposobów realizacji konkretnych bloków funkcjonalnych. Stałym artefaktem podczas takiej zmiany jest interfejs, definiujący kontrakt konkretnej gałęzi klas, podczas gdy zmiana zachodzi na poziomie poszczególnych jego implementacji.

Drugi ze wskaźników odnosi się do współczynnika enkapsulacji metod. Obliczany jest z następującego wzoru:

$$MHF = 1 - \frac{\sum MV}{(C-1)}$$

MV - liczba klas, gdzie dana metoda jest widoczna oraz **C** - ilość klas. Nie ma jednoznacznie przyjętej poprawnej wartości tej metryki, niemniej uznaje się, że im wyższa wartość tym jakość kodu jest większa, a potencjalne błędy skupione i łatwe do zlokalizowania. Z drugiej strony wysoka wartość oznacza wysoką specjalizację klas przy jednocześnie niskim poziomie funkcjonalności, która rozsiana jest między poszczególnymi elementami systemu.

5. Podsumowanie

Głównymi celami pracy było zaprojektowanie i przygotowanie aplikacji wspierającej warsztat samochodowy w zakresie zarządzania terminarzem wizyt, bazą klientów, informacjami o samym warsztacie oraz poznanie szkieletu aplikacji Spring, jako kompleksowego narzędzia wspierającego tworzenie rozbudowanych programów **Java EE**. Znaczenie oraz korzyści jakie przynosi korzystanie z tego typu aplikacji nie sposób nie zauważyć. Dobrze zaprojektowana jest cennym dodatkiem wspomagającym pracę przedsiębiorstwa w zakresie zarówno chwili obecnej, jak i analizy danych historycznych. Program obejmujący swym zasięgiem globalny aspekt misji danej firmy, przy jednoczesnym dostępie do poszczególnych elementów.

W kontekście aplikacji demonstracyjnej nie udało się zrealizować wszystkich zamierzonych postulatów. Niemniej część z brakujących elementów, z uwagi na pracę oraz czas poświęcony na zaprojektowanie niewidocznych dla użytkownika elementów, co wcale nie umniejsza ich znaczenie, będzie możliwa do szybkiego wprowadzenia do gotowego rozwiązania. Pozostałe braki wynikają głównie z problematyki rozwiązania poszczególnych problemów i dla celu aplikacji demonstracyjnej zostały pominięte.

Udało się natomiast dobrze poznać podstawowe prawa i reguły rządzące pisaniem rozbudowanego projektu, zarówno w sensie ogólnym oraz przy użyciu szkieletu aplikacji Spring. Rzeczy, które mogą się wydawać trywialne, jak dobry dobór bibliotek, projekt przed implementacją, testy jednostkowe, a które zostały zaczerpnięte z zestawu **best practices** dla Spring, pozwoliły zrozumieć na co zwracać szczególną uwagę oraz jak ważne mogą stać się małe pomyłki.

Bibliography

[1] Fernando Brito e Abreu. *MOOD Metrics*. URL: http://www.aivosto.com/project/help/pm-oo-mood.html.

[2] *c3p0 - JDBC3 Connection And Statement Pooling*. URL: http://www.mchange.com/projects/c3p0/.

[3] Neal Ford. *Art of Java Web Development*. Manning Publications, 2004.

[4] *Java Persistence Query Language*. URL: http://docs.oracle.com/javaee/7/tutorial/doc/persistence-querylanguage.htm.

[5] Chidamber Kemerer. *Chidamber Kemerer object-oriented metrics suite*. URL: http://www.aivosto.com/project/help/pm-oo-ck.html.

[6] Thomas Risber Jonathan L. Brisbin Michael Huner Mark Pollack Olivier Gierke. *Spring Data: Modern Data Access For Enterprise Java*. ISBN: 978-1-449-32395-0. O'Reilly, 2013.

[7] Dirk Riehle. "Framework Design: A Role Modelling Approach". PhD thesis. ETH Zürich, 2000.

[8] Wydawnictwo Naukowe PWN SA. *S?ownik J?zyka Polskiego*. URL: polish_dictionary.

[9] Pivotal Software. *Spring Framework Reference Documentation*. 2014. URL: http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/.

[10] Pivotal Software. *Spring Web Flow Reference Documentation*. 2014. URL: http://docs.spring.io/spring-webflow/docs/2.4.x/reference/htmlsingle/.

[11] Pivotal Software. *Understanding HATEOAS*. 2013. URL: http://spring.io/understanding/HATEOAS.

[12] Terracotta. *Ehcache - Documentation*. 2014. URL: http://ehcache.org/documentation/index.

Spis tabel

Spis rysunków

Kody źródłowe