Politechnika Łódzka

Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Narzędzie wspierające zarządzanie warsztatem samochodowym wykorzystujące szkielet aplikacji Spring

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr inż. Arkadiusz Tomczyk
Dyplomant: Tomasz Trębski
Nr albumu: 165535
Kierunek: Informatyka
Specjalność: Inżynieria oprogramowania i analiza danych

Łódź, 26.06.2014

Instytut Informatyki

90-924 Łódź, ul. Wólczańska 215, budynek B9

tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

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 szkieletów aplikacji, 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:

- **Warsztat NS**,
- **Warsztat 24**,
- **Warsztat 3.4**,
- **Asystent Warsztat**

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 ze znalezionych rozwiązań to programy instalowane na komputerach docelowych, nie obsługiwane poprzez interfejs przeglądarki internetowej.

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 do-

stępną 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 związanych z tworzeniem nowych wizyt, rejestracją klientów, pojazdów i mechaników oraz utrzymaniem ich kartoteki oraz skomplikowanego modelu danych. Kompleksowy model danych będzie musiał dostarczać wszelkich możliwych informacji odnośnie obiektów, które opisuje. Przykładowo, kluczowymi atrybutami opisującymi samochód, nie będą jedynie **numery rejestracyjny**, **marka**, czy też **model**. Ważne stanie się opisania danego pojazdu parametrami takimi jak: **numer VIN**, **rok produkcji**, **spalane paliwo** oraz **parametry silnika**. Ostatecznie, wzięte pod uwagę zostanie utrzymanie historii własności pojazdu. Ponadto dla niektórych obiektów modelu danych zostanie dostarczone przeglądanie historii zmian ich atrybutów.

Drugim celem pracy jest dostarczyć wiedzy o projektowaniu oraz implementacji aplikacji biznesowych w języku Java przy wykorzystaniu Spring'a. Szczególnie ważny stanie się tutaj aspekt samego projektowania aplikacji, realizowany poprzez wykorzystanie standardowych, gotowych rozwiązań dostarczonych przez wybrany szkielet aplikacji. Mowa tutaj o takich elementach jak praca z danymi (operacje zapisu i odczytu, walidacja), separacja modelu danych i logiki biznesowej, efektywne mapowanie żądań klienta do serwera. Ponadto na potrzeby aplikacji demonstracyjnej zaprojektowane i zaimplementowane zostaną generyczne moduły wspierające wyświetlanie informacji o obiektach modelu danych, tabeli oraz tworzenia dynamicznych szablonów dla raportów biznesowych. Wspomniane moduły będą miały za zadanie wesprzeć dalszy rozwój aplikacji poprzez uproszczenie i automatyzację następujących przypadków: wprowadzenie nowego typu danych i dodanie dla niego stron wyświetlającej jego atrybuty, utworzenie nowej tabeli (zarówno dla nowego typu danych, ale także wyświetlającej informacje niezwiązane z modelem biznesowym), modyfikacja lub tworzenie nowego szablonu dla raportu.

2. Wymogi funkcjonalne aplikacji

2.1. Wymogi biznesowe

2.1.1. Główna funkcjonalność biznesowa

Analiza istniejących rozwiązań pozwoliła ustalić, że minimalna funkcjonalność programu realizującego wsparcie dla przedsiębiorstwa prowadzącego warsztat samochodowy nie powinna się ograniczać jedynie do wspomagania procesów: wymiany płynów eksploatacyjnych czy też napraw. Prowadzenie 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 informacji z poprzedniej wizyty klienta i sprawdzić, co zostało wykonane. Funkcjonalność programu dla warsztatów samochodowych to między innymi wsparcie w następujących obszarach:

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

2.1.2. Przewodniki - kreatory nowych obiektów

Biorąc pod uwagę funkcjonalność w zakresie realizacji celów biznesowych jednym z nadrzędnych aspektów 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.

2.1.3. 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.

2.2. Wymogi techniczne

2.2.1. Architektura aplikacji

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

[1] Model-View-Controller

2.2.2. 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.

2.2.3. 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 taki moduł może działać dla dowolnych danych wejściowych, zgodnych z jego specyfikacją, i nie ma potrzeby dostosowywania kodu do różnych typów danych. Moduły te są główną częścią aplikacji wspierającą realizację wymogów biznesowych.

Strony obiektów domenowych

Strony obiektów domenowych realizować mają koncepcję dostępu do informacji, atrybutów danego obiektu, należącego do warstwy modelu danych, poprzez pojedynczą stroną internetową. Dla każdego obiektu przypisany będzie format adresu złożony z unikatowego klucza (innego dla każdego obiektu domenowego), klucza głównego (odpowiadającemu kluczowi głównego encji w bazie danych) i numeru wersji (dla obiektów wersjonowanych).

Adres służyć będzie pobraniu definicji oraz informacji dla danej strony. Struktura opisywać będzie to, jakie atrybuty, i w jakiej postaci (jako wartość, tabela czy też hiperłącze), mają zostać zaprezentowane użytkownikowi. Dane, z drugiej strony, będą musiały być zgodne z wybraną dla nich reprezentacja.

Generowanie definicji i danych dla tabel

Generowanie danych oraz definicji tabel będzie czynnością podobną, w kontekście algorytmu, do procesu prezentowania użytkownikowi strony obiektu domenowego. Główną różnicą będzie to, w jakim formacie zwracany będzie zbiór danych. Podczas, gdy dla pojedynczej strony, relacja komponent - strona to relacja jeden do jednego, tabela wyświetlać będzie wiele obiektów tego samego typu.

Generator szablonów dla raportów

Szablony raportów będą grupą obiektów, gdzie każdy z nich zawierać będzie informacje niezbędne do utworzenia nowego raportu. Głównym wymogiem dla tego modułu będzie możliwość wybrania jednej z czterech możliwych reprezentacji: PDF, CSV, strona internetowa lub plik Excel. Ponadto, podczas tworzenia szablonu, można będzie wybrać jakie obiekty, oraz jakie atrybuty tych obiektów, znaleźć się mają w nowym szablonie. Dzięki temu, wybranego szablonu będzie można użyć w dowolnej chwili, dla dowolnego zbioru danych.

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.

W pierwszej kolejności wybrany został szkielet aplikacji, który w założeniu stanowić miał solidny fundament, na którym oparta została by aplikacja. Spośród wielu dostępnych rozwiązań 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 szkielet, co odciąża programistę od konieczności kontrolowania tych aspektów.

Po analizie rozwiązań takich jak **JBoss Seam Framework**, **Google Guice**, **PicoContainer** i **Spring Framework**, okazało się, że wszystkie z nich wspierają wymienione techniki. Niemniej stopień w jakim można by użyć jednego z nich do kompleksowego wsparcia aplikacji praktycznej nie był 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. Dzięki przyjętemu, przez szkielet, modelowi programistycznemu, opierającemu się o wykorzystanie interfejsów, zamiast konkretnych ich implementacji, **Spring** jest rozwiązaniem wysoce konfigurowalnym. Nic nie stoi na przeszkodzie, aby do gotowej aplikacji dołączyć dowolną zewnętrzną bibliotekę, czy też zmienić sposób, w jaki dany moduł działa, poprzez dostarczenie własnej implementacji pewnego algorytmu.

3.2. Słownik pojęć

Poniższa tabela przedstawia definicję pojęć technicznych wykorzystanych w pracy dyplomowej.

Tabela 1: Adnotacje Spring opisujące poziom abstrakcji cache

Pojęcie	Znaczenie
Dependency injection	Wstrzykiwanie zależności - jest to technika programistyczna, w której dana klasa nie jest odpowiedzialna za tworzenie obiektów od których zależy. Zależność jest umieszczana w danym obiekcie, po jego utworzeniu, przez specjalnego zarządcę, posiadającego wiedzę o wszystkich obiektach istniejących w działającej aplikacji. Pozwala to jednocześnie na obniżenie ścisłej zależności między dwoma klasami, a także zmniejsza zapotrzebowania na zasoby systemowe. Jest to możliwe, ponieważ, bardzo często, obiekty, będące zależnościami dla innych, istnieją w kontekście danego programu, jako **singletony**.
Singleton	**Singleton** jest wzorcem programistycznym, zakładającym istnienie tylko jednego obiektu danej klasy w całej aplikacji. Najczęściej taka funkcjonalność, realizowana jest poprzez utworzenie klasy, która nie posiada publicznego konstruktora, a jej obiekt uzyskiwany jest poprzez statyczną metodę, zwracającą obiekt tej klasy. W kontekście szkieletów aplikacji, wspierających **wstrzykiwanie zależności**, wzorzec singletonu realizowany jest na poziomie zarządcy.
Zależność przechodnia	Taka zależność pojawia się w momencie kiedy aplikacja korzysta z biblioteki, która z kolei korzysta z innych. Skutkiem tego jest, że aplikacja staje się zależna od bibliotek od których bezpośrednio nie zależy. Staje się to problematyczne, w momencie kiedy programista zaczyna wykorzystywać funkcjonalność zdefiniowaną w nich, nie wiedząc o tym. Usunięcie bezpośredniej zależności, skutkować będzie błędami kompilacji.
Obiekt domenowy	Klasy takich obiektów zdefiniowane są w modelu danych. Innymi słowy dostarczają one 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
Następna strona...	

Tabela 1 – kontynuacja...

Pojęcie	Znaczenie
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.
AOP	**Aspect Oriented Programming** jest paradygmatem programistycznym zwiększającym skalowalność, zmniejszającym kohezję klas i tym samym eliminującym tak zwane *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 zdefiniowany 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.
ORM	Z angielskiego, **Object/Relational Mapping**, jest to technika programistyczna, w założeniu implementująca proces konwertowania danych, które nie są obiektami, jak krotki w bazie danych, na model zaprojektowany, zgodnie z wytycznymi programowania obiektowego. W wyniku tego, programista uzyskuje dostęp do wirtualnej bazy danych.
Getter/Setter	Zwyczajowe pojęcia opisujące metody dostępowe klasy służące do pobierania lub ustawienia wartości pól obiektów tej klasy
API	Ściśle określony zbiór reguł i metod, dzięki którym program może się komunikować ze sobą lub z innym programem.
JMS	**Java Message Service** to część języka Java, która pozwala dwóm i więcej programom komunikować się poprzez jednolity interfejs oraz format wiadomości [7].
JMX	**Java Management Extensions** jest technologią, która jest częścią standardowej biblioteki Java. Dzięki JMX można kontrolować stan aplikacji i urządzeń oraz dynamicznie wpływać na ich stan [8].
Best practice	Zalecane i pożądane sposoby realizacji często spotykanych problemów, które można napotkać, podczas projektowania aplikacji
DRY	**Dont Repeat Yourself** - zasada negująca implementacje rozwiązań problemów, które już zostały rozwiązane.
Artefakt	Artefakt, w kontekście programowania obiektowego w Java, jest pojęciem jednocześnie opisującym takie elementy tego języka jak klasy, interfejsy oraz paczki (zbiór klas i interfejsów).

Następna strona...

Tabela 1 – kontynuacja...

Pojęcie	Znaczenie
Predykat	Predykat, coś co umożliwia ustalenie czegoś. Predykat można rozumieć jako kompozycję wyrażań logicznych lub specjalny rodzaj metody weryfikującej zadany problem pod kątem ustalenia wartości prawda/fałsz.
Classpath	**Classpath** - ścieżka klas, jest parametrem maszyny wirtualnej Java, który wskazuje na lokalizację folderu w systemie plików, gdzie znajduje się skompilowane klasy Javy.
Rewizja	Rewizja jest specjalnym numerem, który jest bezpośrednio związany z historią zmian obiektu domenowego. Każda modyfikacja takiego obiektu oznacza w praktyce utworzenie nowej rewizji.
Lokalizacja	Lokalizacja, w kontekście dowolnej aplikacji, należy rozumieć jako możliwość programu do wsparcia więcej niż jednego języka interfejsu użytkownika.
Przypadek użycia	Sposób opisu wymagań aplikacji na poziomie interakcji między użytkownikiem aplikacji, a nią samą lub konkretna sytuacja występująca w programie.

3.3. Szkielet aplikacji

3.3.1. 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**. Jest to szczególnie użyteczne ponieważ nierzadko zdarza się, że programista popełnia błąd 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 szkielet programistyczny 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 [9].

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.

Problemy szkieletów aplikacji

Mimo że szkielety aplikacji znacząco podnoszą jakość kodu oraz obniżają późniejsze koszty jej utrzymania, nie są doskonałym narzędziem. Większość trudności, jakie można napotkać podczas korzystania z nich, wynika z ich rozmiaru oraz ze złożoności. Złożoność należy rozumieć zarówno w kontekście tego, jak dany szkielet jest zaprojektowany, oraz w kontekście wielu obszarów funkcjonalnych, które wspiera. Poniższe zestawienie podsumowuje najczęściej spotykane problemy, z którymi można zetknąć się podczas korzystania ze szkieletów aplikacji:

- *złożoność modelu 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,

- *skupienie się na szczególe, pominięcie ogółu* - 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 kodu, zadaniem którego jest obejście problemu, a nie jego zrozumienie. Nie jest to jedynie aspekt dotyczący samego szkieletu, ale także aplikacji z niego korzystającej. Mimo wszystko nadmiarowy kod wciąż bazuje na klasach danej biblioteki. Zmiany zachodzące w platformie propagują do opartej o nią aplikacji, a działający wcześniej kod, przestaje działać. Programista aplikacji tworzy kolejny, jeszcze bardziej skomplikowany, aby uzyskać funkcjonalność, rzekomo nie dostarczoną przez szkielet programistyczny.

Rozwiązaniem tych trudności jest zmiana koncepcji, według której projektowany był szkielet aplikacji. Tradycyjne podejście, oparte o klasy, zostaje zastąpione podejściem opartym o role. Idea zakłada wykorzystanie istniejących klas, które grupowane są w bloki funkcjonalne. Każdy z takich bloków to inna rola, a każdy obiekt posiada swoją własną, podrzędną rolę. Dzięki temu szkielet staje się zbiorem funkcji, które z kolei składają się z wielu ról, współpracujących ze sobą dla uzyskania konkretnego wyniku. Tak zaprojektowana i podzielona platforma jest łatwiejsza w zrozumieniu i wykorzystaniu dla celów programów, opartych o nią [9].

Funkcjonalność szkieletów aplikacji

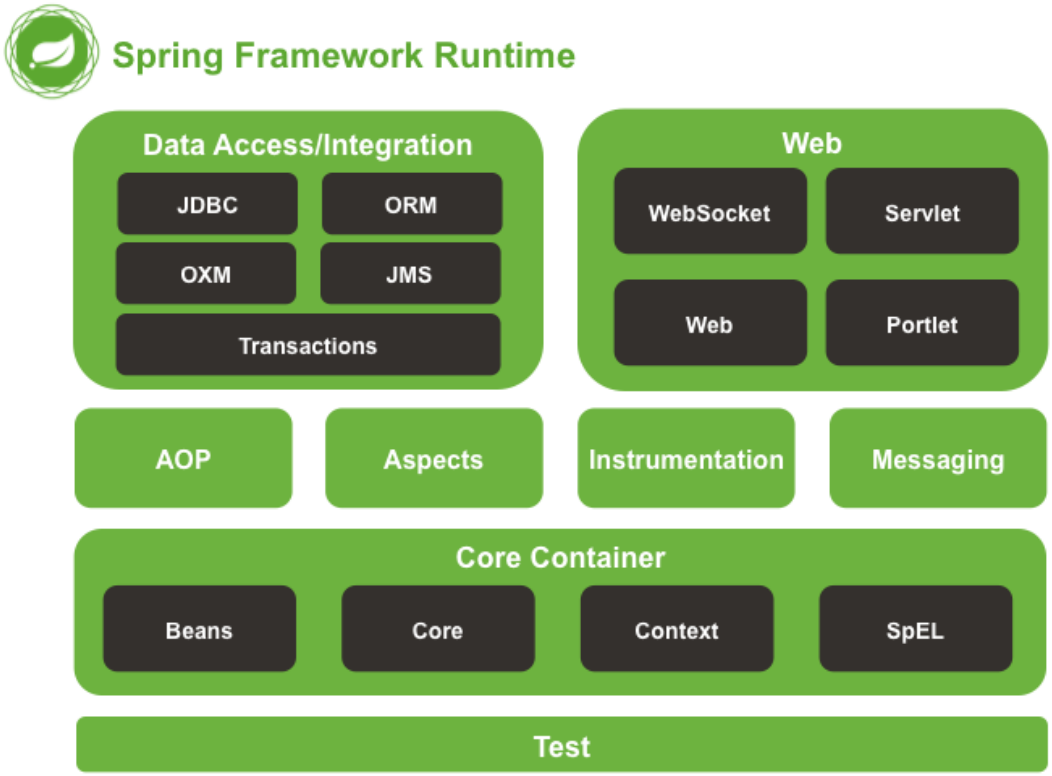
Szkielety aplikacji dostarczają jednolitego sposobu budowania programów o nie opartych. Zestawy najlepszych praktyk, komponenty realizujące powtarzalne operacje, a także sposoby realizacji najczęściej spotykanych problemów, to wspólny mianownik szkieletów programistycznych, nie tylko tych przeznaczonych dla aplikacji internetowych. Dla tego typu programów, istniejące rozwiązania, implementują:

- wsparcie dla wielojęzycznych aplikacji,
- wsparcie dla różnych rodzajów widoku (strony HTML, strony JSP, pliki PDF lub Excel),
- integrację z językiem szablonów pozwalającym, w strukturę strony HTML, dodawać elementy generujące dynamiczne treści,
- dostęp do danych oraz ich walidację,
- wsparcie dla komunikacji klient - serwer w kontekście mapowania akcji wykonywanych przez użytkownika,
- wsparcie dla formularzy internetowych,
- wsparcie dla technologii Ajax

3.3.2. 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

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. Ponadto oparty jest o koncepcję, gdzie poszczególne klasy powiązane są z konkretną rolą, jaką spełniają w programie. Dzięki temu cała platforma jest łatwa w zrozumieniu. Jest to także powód dla którego **Spring**'a, nie należy rozumieć jako biblioteki, która jednocześnie wspiera więcej niż jeden obszar, ale jako zbioru samodzielnych komponentów. Poprzez wzajemną integrację, każda część szkieletu to rozwiązania innego problemu. Programista może wybrać tę część najlepiej odpowiadającą jego potrzebom. **Spring** składa się z ponad 20 samodzielnych modułów:



Rysunek 1: *Kontener Spring wraz z modułami*
źródło: [12]

- **Core Container** - fundament szkieletu na którym oparte są pozostałe modułu. Definiuje on funkcjonalność **Dependency Injection** oraz odwróconego sterowania, a także posiada definicję obiektów takich jak **Bean**, **Context**. Jego częścią są również klasy niezbędne do ładowania zasobów (plików), lokalizacji oraz **Expression Language**[1],
- **Data Access/Integration** - określa sposób dostępu dla takich źródeł danych jak bazy danych, pliki XML, czy też zdalne źródła danych dostępne przez protokół JMS.

[1] Manipulowania obiektami poprzez wyrażania zapisane czystym tekstem, które są później tłumaczone na odpowiednie wywołania programowe

Najważniejszą zaletą jest maksymalne wykorzystanie spójnych interfejsów do dostępu do danych i ukrycie ich źródła.
- **Web** - składa się z klas, które pozwalają na inicjalizację kontenera odwróconego sterowania w kontekście działającego serwletu. Definiuje również funkcjonalność **Spring MVC**, które z kolei jest praktyczną implementacją wzorca **MVC**,
- **AOP** - dostarcza sposoby oraz środki do programowania aspektowego.

Modularna budowa jest także 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, komunikacji protokołem JMS czy też dynamicznego kontrolowania aplikacji poprzez protokół JMX.

3.3.3. 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 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. Nie ma potrzeby redefiniowania modelu danych jako klas, których pola są prostymi typami danych, takimi jak liczby czy też łańcuchy znaków.

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

Głównymi zaletami **Spring MVC** 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**. 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 (listing 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.

```
1  public interface JpaRepository<T, ID extends Serializable>
2              extends PagingAndSortingRepository<T, ID> {
3      List<T> findAll();
4
5      List<T> findAll(Sort sort);
6
7      List<T> findAll(Iterable<ID> ids);
8
9      <S extends T> List<S> save(Iterable<S> entities);
10
11     void flush();
12
13     T saveAndFlush(T entity);
14
15     void deleteInBatch(Iterable<T> entities);
16
17     void deleteAllInBatch();
18
19     T getOne(ID id);
20 }
```

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**[2] 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.

[2] Java Enterprise Edition

Spring Web Flow

"Przemieszczenia się kogoś, czegoś, przekazywanie, obieg czegoś."[10]

SWF[3] 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** 3.3.3 oraz silnikiem walidacji i konwersji typów.

Flow [13] - 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.

```
1  <view-state id="entity" view="ui.wizard.NewReportWizard.PickEntity" popup="true">
2      <on-render>
3          <evaluate expression="pickEntityFormAction.setupForm"/>
4      </on-render>
5      <transition on="next" bind="true" validate="true" to="pickColumns">
6          <evaluate expression="pickEntityFormAction.bindAndValidate"/>
7      </transition>
8      <transition on="cancel" to="cancel" history="invalidate"/>
9  </view-state>
```

Listing 2: Deklaratywna deklaracja stanu - kroku dla przepływu w rozumieniu **Spring Web Flow**, źródło: opracowanie własne

Przykład 2 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.

```
1  @Override
2  public Event setupForm(final RequestContext context) throws Exception {
3      final MutableAttributeMap<Object> scope = this.getFormObjectScope().getScope(context);
4      final Set<RBuilderEntity> entities = this.getReportableEntities(context);
5      scope.put(ENTITIES, entities);
6      scope.put(ASSOCIATION_INFORMATION, this.getReportableAssociations(entities));
7      return super.setupForm(context);
8  }
```

Listing 3: *setupForm* - metoda *setupForm* wykorzystywana w definicji kroku 2 do umieszczenia danych w kontekście przepływu, źródło: opracowanie własne

[3] Skrót od Spring Web Flow

3.4. Technologie wspierające

Mimo, że **Spring** jest doskonałym fundamentem dla aplikacji internetowej, nie dostarcza on bezpośredniego wsparcia, ani też nie narzuca, w jaki sposób aplikacja ma się komunikować z bazą danych, czy też z jakiego silnika bazodanowego ma korzystać. Nie istnieje też żaden komponent, który zarządzałby połączeniem z samą bazą, zajmował się kolejkowaniem żądań i rozdzielaniem zasobów. Zarówno to, że **Spring** jest wysoce konfigurowalnym rozwiązaniem oraz, że jest niezwykle popularny, nie ma żadnego problemu z wyborem technologii, które wspierały by pewną funkcjonalność czy też całkowicie zastępowały tę, którą wybrany szkielet programistyczny oferuje.

3.4.1. Hibernate - Object Relational Mapping

Hibernate to otwarto źródłowy projekt, zaprojektowany dla zarządzania danymi trwałymi w języku Java. Nie jest to problem nowy, ani też łatwy do rozwiązania. Potwierdza to szeroka gama rozwiązań podobnych do **Hibernate**: **jOOQ**, **MyBatis**, **SimpleORM**. Wspólną ich cechą jest efektywne mapowanie relacji i tabel znajdujących się w bazie danych, na zrozumiały i łatwy w utrzymaniu model danych. Różnice natomiast to, to jak szczegółowo można zamodelować kompleksowy model danych, jak i ile różnych silników bazodanowych jest wspierane oraz jak rozwiązany jest problem definiowania nowych typów danych.

Wybrany silnik **ORM** rozwiązuje powyższe problemy odpowiadając na pytania: co jeśli baza danych MySQL zostanie zamieniona na bazę danych Oracle, jak efektywnie zaprezentować pojedynczą krotką jako obiekt, czy też zrealizować koncepcję zaprezentowania pojedynczych atrybutów takiej krotki, jako liczb, łańcuchów znakowych, typu wyliczeniowego czy tez innych obiektów. Ostatni z problemów znany jest pod angielską nazwą *object/relational paradigm mismatch*.

Dla złożonej tabeli, posiadającej kilkanaście atrybutów oraz powiązań z innymi, bardzo łatwo wyodrębnić takie, które, z punktu widzenia programowania obiektowego, najlepiej przedstawić jako w postaci odrębnej klasy. Bardzo dobrym przykładem jest tutaj adres. Na adres składają się takie informacje jak ulica, numer domu i/lub mieszkania, kraj oraz numer pocztowy. Nawet na poziomie bazy danych, zalecane jest, aby te informacje przechowywane były w relacji jeden do jednego, pojedynczy atrybut - pojedyncza kolumna. Zarówno przypadek, kiedy adres jest oddzielną tabelą, czy też kiedy jest to grupa atrybutów innej tabeli, jest obsługiwany przez elastyczny model mapowania **Hibernate'a**. To co wyróżnia to rozwiązanie, a co przykłada się na jakość kodu obiektowego, to spełnienie zasady pojedynczego celu wybranej klasy. Adres, zdefiniowany jako samodzielny artefakt, staje się złożonym typem danych, którego można użyć w więcej niż jednym kontekście. Alternatywą dla takiego rozwiązania problemu niespójności danych są definiowane przez użytkownika, na poziomie bazy danych, własne typy. Niemniej jest to funkcjonalność, której implementacja jest unikatowa dla poszczególnych systemów bazodanowych. Skutkiem tego jest brak prostego sposobu na przeniesienie takiego typu danych w przypadku migracji do innej bazy danych. Nie jest to jednak kwestia dyskusyjna, trudna do zrealizowania dla **Hibernate**. Wsparcie dla wielu baz danych, daje możliwość tworzenie kodu obiektowego, łatwiejszego w przenoszeniu, mniej podatnego na zmiany, w przypadku migracji do innego silnika bazodanowego. Ostatecznie, podobnie jak wybrany szkielet aplikacji, nie wymusza

on korzystania ze wszelkich możliwych funkcjonalności. Z drugiej strony dodanie kolejnego modułu, takiego jak walidacja danych na poziomie modelu obiektowego, jest bardzo prosta i sprowadza się do dodania nowej biblioteki do aplikacji, a następnie opatrzenia odpowiednimi adnotacjami wybranych atrybutów. Adnotacje określają ograniczenie, takie jak zakres liczb dla danych numerycznych lub długość łańcuchów znakowych.

3.4.2. QueryDSL

Wykorzystanie **Spring Data JPA** (3.3.3) w znaczący sposób uprościło warstwę dostępu do danych. Niemniej w dalszym ciągu nie istniała możliwość budowania bardziej skomplikowanych zapytań, w których można by pobrać dane po więcej niż tylko kluczu głównym. Spośród przeanalizowanych rozwiązań, zapewniających tę funkcjonalność, takich jak **JPA Criteria**, **Hibernate Criteria** i **QueryDSL**, wybrany został ostatni z wymienionych projektów. Najważniejszym czynnikiem, pod kątem którego ocenione zostały rozwiązania, było generowanie silnie typizowanych zapytań. Niestety **Hibernate Criteria**, pozwalając tworzyć zapytania zorientowane na konkretną klasę modelu danych, jednocześnie nie dawał możliwości uzyskania wyniku zgodnego z oczekiwanym. W dalszym ciągu wymaganą operacją było rzutowanie na pożądany typ danych. Z drugiej strony **JPA Criteria**, zostało odrzucone z uwagi na problematyczne tworzenie zapytania. Koncepcja oparta o tworzenie klas, których zadaniem było zwracanie jego specyfikacji, wciąż wymagała napisania od kilkunastu do kilkudziesięciu linii kodu.

```
1   @Override
2   public Collection<Owner> getOwners() {
3       final Set<Owner> ownerSet = Sets.newHashSet();
4
5       final QSUser user = QSUser.sUser;
6       final QSCar car = QSCar.sCar;
7
8       final EnumPath<SRole> role = user.roles.any().pk.authority.role;
9
10      BooleanExpression predicate = role.in(SRole.ROLE_CLIENT, SRole.ROLE_MECHANIC)
11              .and(user.credentials.username.ne("SYSTEM"))
12              .and(user.accountNonLocked.eq(true))
13              .and(user.accountNonExpired.eq(true)
14              .and(user.enabled.eq(true)));
15      try {
16          final Iterable<SUser> users = this.userRepository.findAll(predicate);
17
18          for (final SUser userFound : users) {
19              final Iterable<SCar> all = this.repository.findAll(QSCar.sCar.owner.eq(userFound));
20              ownerSet.add(
21                  new Owner()
22                      .setOwner(userFound)
23                      .setCars(Sets.newHashSet(all))
24              );
25          }
26
27          return ownerSet;
28      } catch (Exception exp) {
29          LOGGER.fatal("Failure in retrieving possible owners", exp);
30      }
31
32      return null;
33  }
```

Listing 4: Specyfikacja zapytania w rozumieniu biblioteki **QueryDSL**, źródło: opracowanie własne

QueryDSL przerzuca odpowiedzialność za utworzenie takiej specyfikacja na siebie. W tym przypadku, to jak zapytanie będzie wyglądać, jakie atrybuty danego modelu, czyli kolumny tabeli, zostaną uwzględnione, kontrolowane jest przez odpowiednie użycie meta modelu danej klasy modelu danych. Wspomniany meta model to klasa języka Java. Wynikająca z tego korzyść polega na tym, że wraz z nim, 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[4], jakby to miało miejsce podczas pisania identycznego zapytania opartego o łączenie łańcuchów znakowych, czy też z wykorzystaniem **Hibernate Criteria** lub **JPA Criteria**. Listing 4 pokazuje tworzenie zapytania, gdzie pobrane obiekty klasy **SUser** muszą posiadać dwie konkretne role, ich nazwa użytkownika musi różnić się od **"SYSTEM"**, konto nie może być zablokowane oraz nieważne. Podczas tworzenia takiego zapytania szczególnie istotne jest to, że oprócz silnego typizowania wyniku, takie wsparcie jest obecne również podczas definiowania specyfikacji. Przykładowo, nie możliwe jest podanie liczby jako argumentu dla atrybuty **accountNotLocked**, który w klasie **SUser**, a tym samym jego meta modelu, zdefiniowany został jako zmienna logiczna.

3.4.3. c3p0

c3p0 jest łatwą do użycia biblioteką zaprojektowaną dla **Java**, której głównym zadaniem jest realizacja postulatów zdefiniowanych przez specyfikacje **JDBC 3**. 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 zasobami, zajmowanie oraz 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].

3.4.4. 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 [14].

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

3.4.5. Apache Tiles

Apache Tiles to biblioteka umożliwiająca dekompozycję widoku aplikacji na wiele niezwiązanych ze sobą bezpośrednio elementów - płytek [5]. 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 3.3.3, gdzie jedną z preferowanych technologii widoku jest właśnie Apache Tiles, a także możliwość prostszego wsparcia dla **partial rendering**[6] 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.

3.4.6. 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 w aplikacji internetowej. 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.

3.4.7. 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.

3.4.8. FullCalendar

Full Calendar jest graficznym komponentem zaprojektowanym na podstawie biblioteki **jQuery**. Jest on odpowiedzialny za wyświetlenie kalendarza, działającego w formie

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

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

terminarza. 3 różne tryby widoku: dzienny, tygodniowy i miesięczny dają możliwość dostosowania terminarza do aktualnych potrzeb. Komponent pobiera zdarzenia korzystając z zapytań Ajax. Sam komponent można dostosować do dowolnego źródła danych [11].

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.

4.1.1. Funkcjonalność aplikacji

W części praktycznej zrealizowana została następująca funkcjonalność:

1. zarządzanie dostępem do poszczególnych stron, z wykorzystaniem uprawnień użytkowników:
   - aplikacja rozpoznaje czy użytkownik jest zalogowany, dostosowując ilość dostępnych funkcji, w zależności od grupy (grup), w których użytkownik się znajduje,
   - weryfikacja jest jednoetapowa,
   - proces weryfikacji opiera się o adres internetowy,
2. zarządzanie spotkaniami:
   - przeglądanie terminarza spotkań na poziomie dnia, tygodnia oraz miesiąca,
   - tworzenie nowych spotkań, dostępne z poziomu terminarza,
3. tworzenie nowych użytkowników:
   - nazwa użytkownika oraz hasło,
   - uprawnienia,
   - dane kontaktowe,
4. przeglądanie obiektów domenowych jako pojedynczych stron internetowych,
5. przeglądanie danych w formie tabelarycznej,
6. tworzenie nowych szablonów raportów biznesowych, zapisywanie ich oraz późniejsze generowanie wyniku

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

4.2. Architektura MVC

4.2.1. Model - Warstwa danych

Na warstwę modelu danych składają się klasy, zwane dalej obiektami domenowymi. Zdefiniowane w aplikacji praktycznej klasy, należące do tej warstwy, opisują obiekty rzeczywiste, związane ze specyfikacją działalności warsztatu samochodowego, a także takie, dzięki którym możliwe jest zarządzanie użytkownikami aplikacji oraz ich uprawnieniami.

W wybranych przypadkach (tabela 2), obiekty domenowe są wersjonowane. Oznacza to, że **Hibernate** trzyma 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:
  - — numer rewizji,
  - — nazwę (pełna nazwa klasy) modelu,
- **{nazwa_tabeli}_history** - zawiera ona te pola, które zostały wybrane, co powoduje utworzenie nowej rewizji obiektu domenowego.

Aby wskazać konkretny obiekt domenowy, dla którego wymagana jest wiedza o historii jego modyfikacji, należy użyć adnotacji **@Audited**. Jeśli zostanie nią opatrzona cała klasa, będzie to równoznaczne z tym, że wszystkie atrybuty tej klasy będą kandydatami do utworzenia nowej rewizji. Z drugiej strony możliwe jest podanie jedynie konkretnych pól. Fragment klasy **SCar** (listing 5) pokazuje użycie adnotacji nad polem **licencePlate**.

```
1        @Audited
2        @NotNull
3        @Column(nullable = false, length = 45, name = "licencePlate", unique = true)
4        @NaturalId(mutable = true)
5        private              String     licencePlate      = null;
```

Listing 5: Użycie adnotacji **@Audited**, źródło: opracowanie własne

Tabela 2: Lista obiektów domenowych

Obiekt domenowy	Wersjonowany	Opis
SAppointment	Nie	Pojedyncza wizyta danego pojazdu w warsztacie samochodowym, która wydarzyła się w konkretnym momencie czasu. Wizyta powiązana jest z konkretnym samochodem, będącym jej podmiotem oraz zawiera informacje o osobie, która zarejestrowała zgłoszenie i była jego wykonawcą (mogą to być inne osoby), a także listę czynności, jakie należało wykonać.
Następna strona...		

Tabela 2 – kontynuacja...

Obiekt domenowy	Wersjonowany	Opis
SAppointmentTask	Nie	Pojedyncza czynność, wykonana podczas wizyty. Czynność opisana jest przez jej typ (może to być, na przykład, wymiana oleju) oraz komentarz (dla wymiany oleju może to być informacja o tym, jaki olej został wymieniony).
SAppointmentIssue	Nie	Opisuje problemową sytuację, związaną z danym spotkaniem. Dla pojedynczej wizyty sytuacją wyjątkową może być fakt nieodbycia się spotkania, ponieważ klient warsztatu, do który należy pojazd (podmiot wizyty), nie stawił się na umówiony termin. Jednocześnie klasa **SAppointmentIssue** jest rozszerzeniem klasy **SIssue**, dziedziczy więc wszystkie jej atrybuty.
SCarMaster	Nie	Zawiera informacje opisujące samochód, które nie zależą od jego konkretnej rewizji. Są to atrybuty takie jak marka, model, producent i kraj, z którego samochód pochodzi.
SCar	Tak	Centralny obiekt domenowy systemu. Na jego opis składają się atrybuty zdefiniowane w klasie **SCarMaster**, a także numer rejestracyjny, numer VIN, rok produkcji, rodzaj spalanego paliwa. Pojedynczy samochód powiązany jest z użytkownikiem systemu, będącym tym samym jego właścicielem. Samochód jest wersjonowany, a atrybuty, których zmiana powoduje utworzenie nowej rewizji to: właściciel, numer rejestracyjny oraz rodzaj paliwa.
SIssue	Nie	**SIssue** opisuje sytuację wyjątkową, związaną z użytkownikiem systemu. Pojedynczy obiekt tego typu opisany jest przez użytkownika systemu, który zgłosił problem oraz użytkownika będącego podmiotem problematycznej sytuacji.

Następna strona...

Tabela 2 – kontynuacja...

Obiekt domenowy	Wersjonowany	Opis
SPerson	Tak	Ten obiekt domenowy jest pojedynczą osobą w systemie. Obiekt tego typu nie jest tożsamy z użytkownikiem systemu. Atrybuty opisujące ten typ obiektu to: imię, nazwisko oraz dane kontaktowe. Zmiana któregokolwiek z tych atrybutów równoznaczna jest z utworzeniem nowej rewizji.
SPersonContact	Nie	Obiekt tego typu odnosi się do danych kontaktowych, które związane są osobą. Taki obiekt opisany jest przez atrybut, który odnosi się do tego, czy jest to numer telefonu komórkowego lub email. Drugim atrybutem jest wartość danego kontaktu.
SReport	Nie	Obiekt będący częścią komponentu **RBuilder**. Dostarcza informacji o położeniu plików opisujących zarówno strukturę szablonu, jak i pomocnych w procesie generowania raportu.
SAuthority	Nie	Obiekty tej klasy opisują uprawnienia jakie przypisane są pojedynczemu użytkowniki systemu. Dzięki nim możliwe jest kontrolowanie do jakich obszarów funkcjonalnych, pojedynczy użytkownik ma dostęp, a które rejony są dla niego niedostępne.
SUser	Tak	**SUser** jest faktycznym użytkownikiem aplikacji. Pojedynczy obiekt opisany jest przez: login, hasło, zestaw zmiennych logicznych pozwalających określić stan konta (konto nieaktywne, zablokowane lub nieważne). Rola użytkownika, to kim on jest w kontekście systemu, opisane jest przez zbiór uprawnień. Modyfikacja atrybutów odnoszących się do nazwy użytkownika lub hasła skutkuje utworzeniem nowej rewizji obiektu.
SUserNotification	Nie	Powiadomienie jest wiadomością jaka jest wysyłana użytkowniki systemu. Składa się z treści wiadomości oraz daty jej wysłania. Dodatkowym atrybutem jest zmienna logiczna odnosząca się do faktu, czy adresat odczytał powiadomienie.

4.2.2. 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**. Programista jest zobowiązany jedynie, w przypadku chęci skorzystania, do utworzenia w swojej klasie pola typu tego interfejsu, a właściwa referencja do obiektu zostanie umieszczona poprzez **dependency injection**. Kod na listingu 6 pokazuję definicje repozytorium.

```
1  public interface SCarMasterRepository
2          extends SBasicRepository<SCarMaster, Long> {
3      String REPO_NAME      = "CarMasterRepository";
4      String REST_REPO_REL  = "repo.carmaster";
5      String REST_REPO_PATH = "car_master";
6
7      @RestResource(rel = "byBrandAndModel", path = "brandAndModel")
8      SCarMaster findByManufacturingDataBrandAndManufacturingDataModel(
9              @Param("brand") final String brand,
10             @Param("model") final String model
11     );
12
13     @RestResource(rel = "byBrand", path = "brand")
14     Page<SCarMaster> findByManufacturingDataBrand(
15             @Param(value = "brand") final String brand,
16             final Pageable pageable
17     );
18
19     @RestResource(rel = "byModel", path = "model")
20     Page<SCarMaster> findByManufacturingDataModel(
21             @Param(value = "model") final String model,
22             final Pageable pageable
23     );
24
25     @RestResource(rel = "byBrandOrModelContaining", path = "brandOrModel_contains")
26     @Query(
27         name = "byBrandOrModelContaining",
28         value = "select cm from SCarMaster as cm " +
29                 "where cm.manufacturingData.brand " +
30                 "like %:arg% or cm.manufacturingData.model like %:arg%"
31     )
32     List<SCarMaster> findByManufacturingDataBrandContainingOrManufacturingDataModelContaining(
33             @Param("arg") final String searchArgument
34     );
35 }
```

Listing 6: **SCarMasterRepository** - interfejs repozytorium pokazujący użycie metod mapowanych na kwerendy oraz bardziej skomplikowanego zapytania z użyciem adnotacji **Query**, źródło: opracowanie własne

Rozszerzenie natywnej funkcjonalności repozytoriów

Funkcjonalność repozytoriów dostarczona przez moduł **Spring Data JPA** (3.3.3) została rozbudowana o kilka dodatkowych funkcji. Rozszerzenie dotyczyło operacji wykonywanych na obiektach wersjonowanych, a zdefiniowane zostało w klasie **SRepository**, widocznej na listingu 7. Dodatkowe funkcje pozwalają na wyszukiwanie obiektów domenowych według następujących kryteriów:

- konkretny klucz główny oraz konkretna rewizja (metoda **findInRevision(ID,N)**),

- konkretny klucz główny oraz konkretne rewizje (metoda **findInRevisions(ID,N...)**),
- konkretny klucz, gdzie modyfikacja nastąpiła w pewnym momencie czasu (metoda **findRevisions(ID,DateTime,TimeOperator)**),
- obliczenie ilości modyfikacji (metoda **countRevisions(ID)**

```
1  public interface SRepository<T, ID extends Serializable, N extends Number & Comparable<N>>
2              extends SBasicRepository<T, ID>,
3              EnversRevisionRepository<T, ID, N> {
4
5          Revision<N, T> findInRevision(final ID id, final N revision);
6
7          Revisions<N, T> findInRevisions(final ID id, final N... revisions);
8
9          Revisions<N, T> findRevisions(final ID id, final DateTime dateTime, final TimeOperator before);
10
11         long countRevisions(ID id);
12
13         /**
14
15
16         public static enum TimeOperator {
17                 BEFORE,
18                 AFTER,
19                 EQ
20         }
21 }
```

Listing 7: **SRepository** - abstrakcyjne repozytorium wspierające dostęp do rewizji, źródło: opracowanie własne

Uwzględnienie rozszerzenia nie byłoby możliwe bez modyfikacji procesu, podczas którego moduł **Spring Data JPA** tworzył obiekty implementujące funkcjonalność repozytoriów, zdefiniowaną poprzez interfejsy. Wprowadzony został dodatkowy element - fabryka, która nadpisywała oryginalny algorytm dostarczając implementacji zdefiniowanych w aplikacji praktycznej. Listing 8 pokazuje wycinek kodu klasy **SRepositoriesFactoryBean** realizujący tą czynność.

```
1      @Override
2      protected <T, ID extends Serializable> JpaRepository<?, ?> getTargetRepository(
3        final RepositoryMetadata metadata,
4        final EntityManager entityManager) {
5          final JpaEntityInformation<T, Serializable> entityInformation =
6            (JpaEntityInformation<T, Serializable>) getEntityInformation(metadata.getDomainType());
7          final Class<?> repositoryInterface = metadata.getRepositoryInterface();
8
9          SimpleJpaRepository<T, ID> repository;
10       if (!ClassUtils.isAssignable(SRepository.class, repositoryInterface)) {
11             repository = this.newNonVersionedRepo(entityManager, entityInformation);
12         } else {
13             repository = this.newVersionedRepo(entityManager, entityInformation);
14         }
15         repository.setLockMetadataProvider(this.lockModePostProcessor.getLockMetadataProvider());
16         return repository;
17     }
```

Listing 8: **SRepositoriesFactoryBean** - fabryka repozytoriów dla implementacji własnej funkcjonalności, źródło: opracowanie własne

Lista repozytoriów
Poniższa tabela zawiera listę repozytoriów zdefiniowanych w aplikacji demonstracyjnej, nazwę odpowiadającego jej obiektu domenowego oraz informację o tym, czy repozytorium wspiera dostęp do dziennika zmian tego obiektu.

Tabela 3: Lista repozytoriów danych

Repozytorium	Obiekt domenowy	Dziennik zmian
SAppointmentRepository	SAppointment	Nie
SAppointmentIssueRepository	SAppointmentIssue	Nie
SAppointmentTaskRepository	SAppointmentTask	Nie
SCarMasterRepository	SCarMaster	Nie
SCarRepository	SCar	Tak
SIssueRepository	SIssue	Nie
SPersonContactRepository	SPersonContacat	Nie
SPersonRepository	SPerson	Tak
SReportRepository	SReport	Tak
SUserAuthorityRepository	SUserAuthority	Nie
SUserNotificationRepository	SUserNotification	Nie
SUserRepository	SUser	Tak

4.2.3. Model - Warstwa serwisów

Serwisy stanowią w aplikacji element realizujący logikę biznesową, jednak nie odnoszą się one 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.

```
1  public interface SPersonService extends SService<SPerson, Long, Integer> {
2
3      SContact<SPerson> newContactData(@NotNull final String contact, final long assignTo,
4              @NotNull final SContact assignToContact) throws EntityDoesNotExistsServiceException;
5
6      List<SPersonContact> findAllContacts(final Long idClient);
7
8      List<SPerson> findByFirstName(@NotNull final String firstName);
9
10     List<SPerson> findByLastName(@NotNull final String lastName);
11
12     SPerson findByEmail(@NotNull final String email);
13 }
```

Listing 9: **SPersonService** - interfejs serwisu dla modelu SPerson, źródło: opracowanie własne

Serwisy zostały zaprojektowane aby korzystać z repozytoriów danych. Ma to swoje pozytywne skutki i nie jest wcale oznaką nadmiarowości kodu, czy też jego duplikowania. Z uwagi na fakt, ze serwisy odwołują się do danych poprzez interfejsy repozytoriów, 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.

4.2.4. Widok - Warstwa widoku

Warstwa widoku została zaprojektowana z wykorzystaniem standardowej biblioteki tagów **JSTL**, plików **JSP** oraz biblioteki **Apache Tiles** 3.4.5. 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.

```
<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 10: Definicja *tile* - podstawowy element widoku w rozumieniu technologi Apache Tiles, źródło: opracowanie własne

Listing 10 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.

```
      <div class="content">
         <tiles:insertAttribute name="content"/>
      </div>
```

Listing 11: Fragment pliku **JSP** umieszczający w nim atrybut **content**, źródło: opracowanie własne

Generyczny kontroler zwracający widok

Aby zminimalizować ilość kodu i pisania kontrolera posiadającego metody zwracające nazwy widoków dla każdego możliwego adresu, w aplikacji praktycznej istnieje tylko jeden kontroler wspierający to zadania. **SVTilesViewControler** działa w oparciu o plik, zawierający wpisy opisujące mapowania adresów na nazwy widoków. Adres jest tutaj kluczem, z którego kontroler buduje obiekt klasy **UriTemplate**. **UriTemplate** działa podobnie do wyrażenia regularnego i pozwala na stwierdzenie, czy adres pobrany z żądania, pasuje do szablonu. Listing 12 pokazuje definicję metody realizującej mapowanie, natomiast listing 13 to lista adresów wraz z odpowiadającymi im widokami.

```
protected String getViewNameForRequest(final HttpServletRequest request) {
        final String path = this.getUrlPathHelper().getLookupPathForRequest(request);
        for (final UriTemplate template : this.templates.keySet()) {
                if (template.matches(path)) {
                        final String viewName = this.templates.get(template);
                        LOGGER.info(String.format("For path %s resolved view %s", path, viewName));
                        return viewName;
                }
        }
        LOGGER.warn(String.format("Failed to retrieve view for path %s", path));
        return null;
}
```

Listing 12: Generyczny kontroler zwracający nazwy widoków zdefiniowanych w pliku 13, źródło: opracowanie własne

```
/=springatom.tiles.index
/reports=springatom.tiles.reports
/about=springatom.tiles.about
/auth/login=springatom.tiles.auth.login
/auth/failed=springatom.auth.login.failed
/auth/access/denied=springatom.tiles.auth.access.denied
/auth/register=springatom.tiles.auth.register
/dashboard/cars=springatom.tiles.dashboard.cars
/dashboard/calendar=springatom.tiles.dashboard.calendar
/dashboard/reports=springatom.tiles.dashboard.reports
/dashboard/clients=springatom.tiles.dashboard.clients
/garage/lifts=springatom.tiles.garage.lifts
/garage/mechanics=springatom.tiles.garage.mechanics
/admin/db=springatom.tiles.admin.db
/admin/language=springatom.tiles.admin.language
/admin/settings=springatom.tiles.admin.settings
/ip/{objectClass}/{objectId}{?.objectVersion*}=springatom.tiles.ip.InfoPagePers‘
```

Listing 13: Mapowania adresów na nazwy widoków, źródło: opracowanie własne

4.2.5. 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**.

```
public <T extends Persistable> Converter<T, String> getConverterForSelector(final String key) {
    final Optional<PersistableConverter<?>> match = FluentIterable
            .from(this.converters)
            .firstMatch(new Predicate<PersistableConverter<?>>() {
                @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<>();
}
```

Listing 14: **PersistableConverterPicker** - koordynator selektywnej konwersji typów, źródło: opracowanie własne

PersistableConverterPicker posiada metody które pozwalają na wybór selektywnego konwertera do wykonania operacji transformacji. Zostało również zapewnione wsparcie dla istniejącej funkcjonalności, bez użycia selektora. Ta część realizowana jest w metodzie **getDefaultConverter(...)**.

```
public <T extends Persistable> Converter<T, String> getDefaultConverter(final TypeDescriptor sourceType) {
    final Optional<PersistableConverter<?>> match = FluentIterable
            .from(this.converters)
            .filter(new Predicate<PersistableConverter<?>>() {
                @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.class);
                    return annotation != null
                            && String.valueOf(
                                    AnnotationUtils.getValue(annotation, "selector")
                            ).isEmpty();
                }
            });
    if (match.isPresent()) {
        return (Converter<T, String>) match.get();
    }
    return new DefaultPickedConverter<>();
}
```

Listing 15: **PersistableConverterPicker** - pobranie domyślnego konwertera dla typu, źródło: opracowanie własne

Selektywne konwertery różnią się od normalnych jedynie użyciem specjalnej adnotacji **PersistableConverterUtility** (listing 16), która pozwala na ustalenie, że:

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

```
1  public @interface PersistableConverterUtility {
2      /**
3       * The value may indicate a suggestion for a logical component name,
4       * to be turned into a Spring bean in case of an autodetected component.
5       *
6       * @return the suggested component name, if any
7       */
8      String value() default "";
9
10     /**
11      * Defines the selector for which selector may be used. For instance it can be a name of property of given {@link
12      * org.springframework.data.domain.Persistable}
13      * <p/>
14      * If this value remains {@code empty} it means that default converter will be used
15      *
16      * @return selector
17      */
18     String selector() default "";
19 }
```

Listing 16: **PersistableConverterUtility** - adnotacja opisująca selektywny konwerter

4.3. Przewodniki tworzenia nowych obiektów

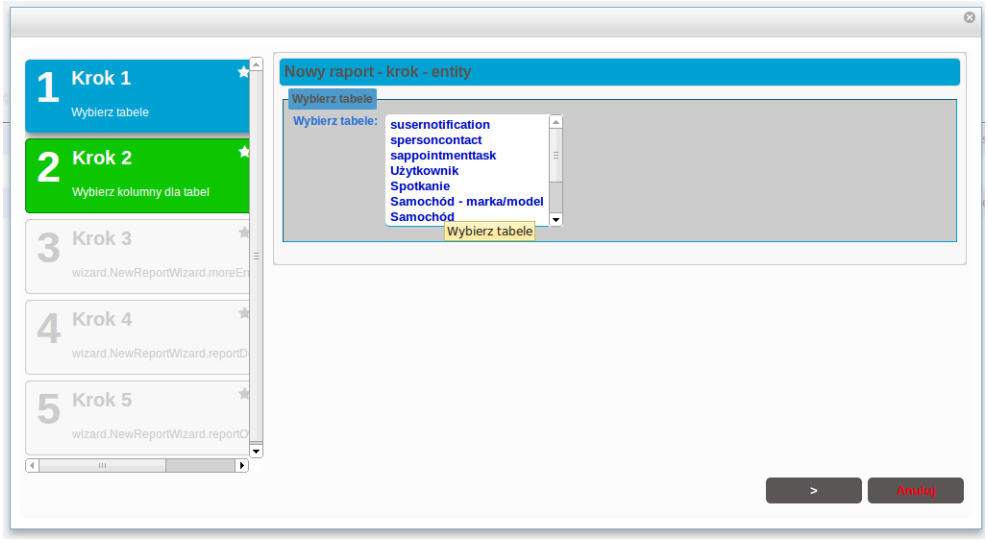
Przewodniki tworzenia nowych obiektów zostały oparte o bibliotekę **Spring WebFlow** (3.3.3). Każdy z kreatorów opisany jest na trzech płaszczyznach. Jedną z nich jest warstwa widoku. Widok składa się z plików **JSP**, gdzie każdy z nich zawiera formularz pozwalający na wprowadzenie danych. Relacja między nimi a przewodnikiem opisana jest przez relację jeden do jednego - jeden formularz przypada na jeden plik **JSP**. Definicja pełnej sekwencji kroków zapisana jest w pliku **XML**, zrozumiały dla biblioteki **Apache Tiles**. Po drugiej stronie istnieje natomiast opis przepływu. Przepływ, w rozumieniu biblioteki **Spring WebFlow**, to sekwencja kolejnych widoków, prezentowanych użytkownikowi oraz możliwych przejść z danego kroku do innych. Trzecia płaszczyzna to klasy języka Java, w których zdefiniowana została logika dla każdego z kroków. Zadaniem tych klas jest przygotowanie danych, które przekazane zostaną do widoku tuż przed jego wyświetleniem, pozwalającym tym samym pokazać elementy takie jak listy rozwijane. Umożliwiają również przetworzenie surowych danych, które użytkownik wprowadził w formularzu, na zrozumiałą dla aplikacji postać. Współpraca wszystkich trzech płaszczyzn zapewniona jest przez bibliotekę **Spring WebFlow**, która efektywnie mapuje żądania pochodzące od klienta, na, w pierwszej kolejności właściwy moment przepływu, a w dalszej, na klasy obsługujące ten moment.

4.3.1. Kreator nowego szablonu raportu

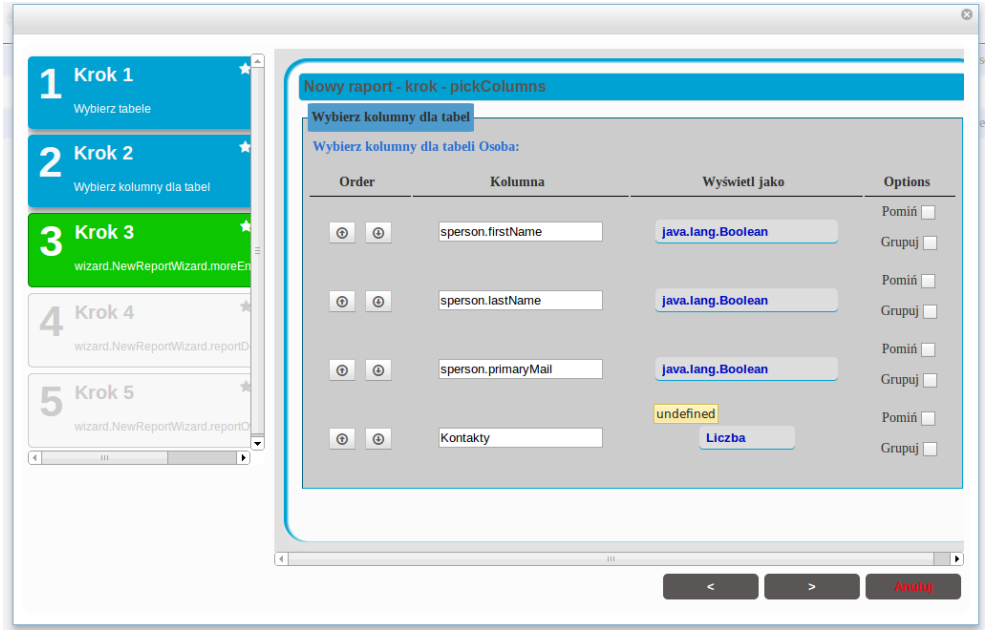
ReportBuilder jest narzędziem będącym obecnie w fazie rozwoju, niemniej pozwalającym już teraz na tworzenie raportów biznesowych. Dedykowany dla użytkownika, daje mu możliwość wybrania zbioru interesujących go tabel, kolumn które zebrane razem stanowią

logiczny zbiór używany w dalszej kolejności do konstrukcji zapytania do bazy danych i utworzenia gotowego raportu. Przewodnik składa się z 3 kroków:

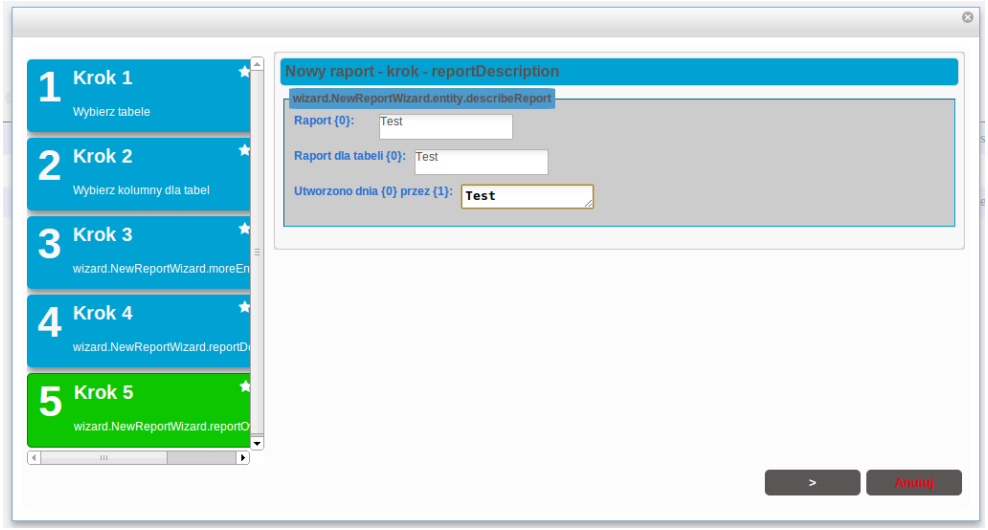
1. wybranie tabeli lub tabel, dla której wygenerowany zostanie raport
2. dostosowanie formatowania kolumn,
3. podanie danych opisujących raport: tytuł, podtytuł, komentarz



Rysunek 2: *Kreator nowego raportu - krok 1, źródło: opracowanie własne*



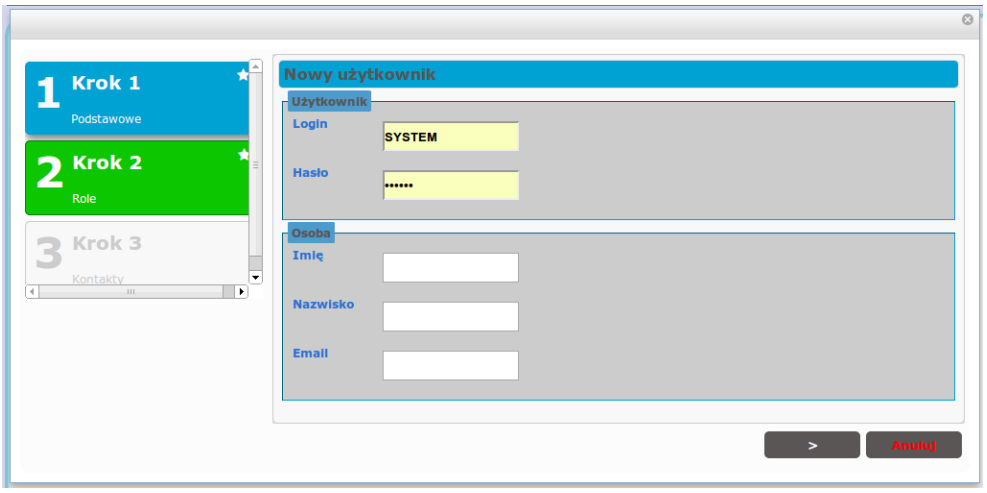
Rysunek 3: *Kreator nowego raportu - krok 2, źródło: opracowanie własne*



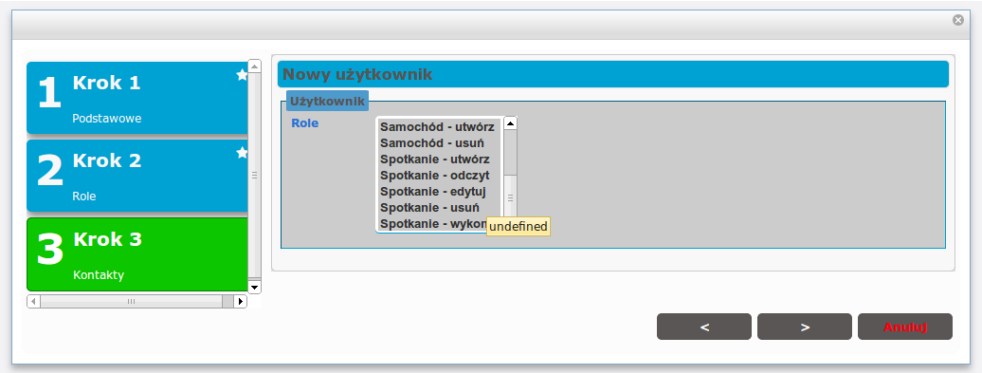
Rysunek 4: *Kreator nowego raportu - krok 3, źródło: opracowanie własne*

4.3.2. 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 wybiera się zestaw ról. Role, do których przypisany jest użytkownik, stanowią późniejszą bazę do weryfikacji dostępności funkcji systemu dla poszczególnych użytkowników.



Rysunek 5: *Kreator nowego użytkownika - dane podstawowe, źródło: opracowanie własne*



Rysunek 6: *Kreator nowego użytkownika - uprawnienia użytkownika, źródło: opracowanie własne*

Rysunek 7: *Kreator nowego użytkownika - dane kontaktowe, źródło: opracowanie własne*

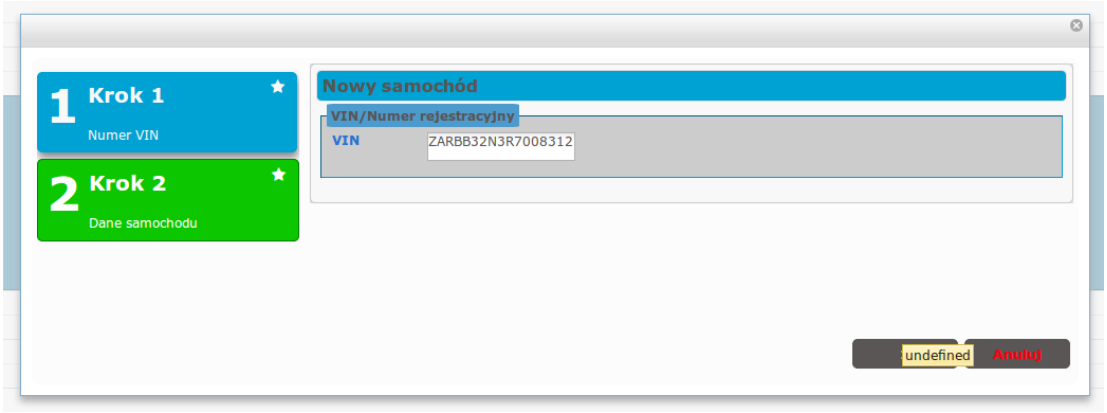
4.3.3. Kreator nowego samochodu

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:

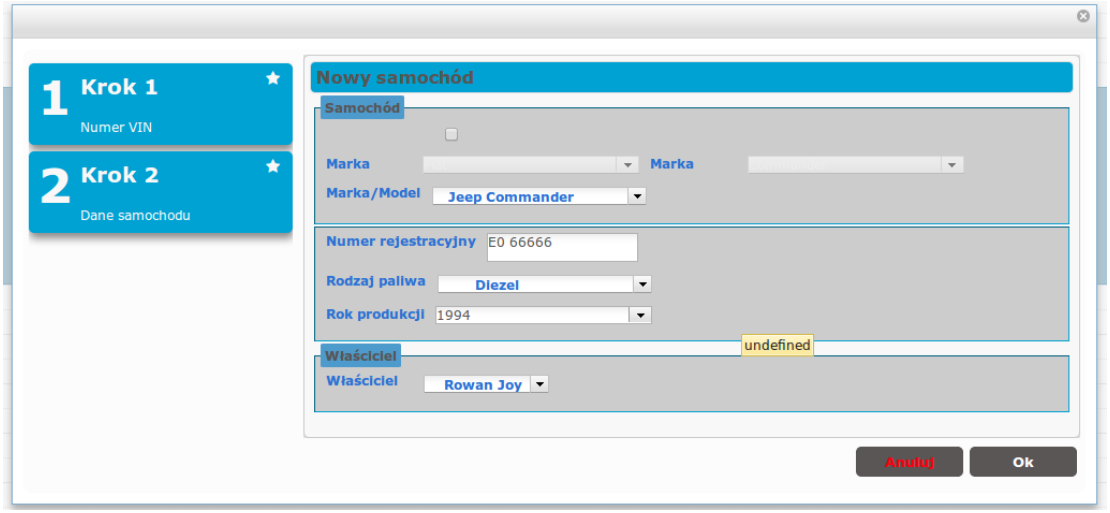
- rok produkcji, zwracany jako lista lat w których samochód mógł być wyprodukowany,
- kraj, w którym samochód został wyprodukowany.

W drugim kroku kreator nowego samochodu podaje takie informacje jak:

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

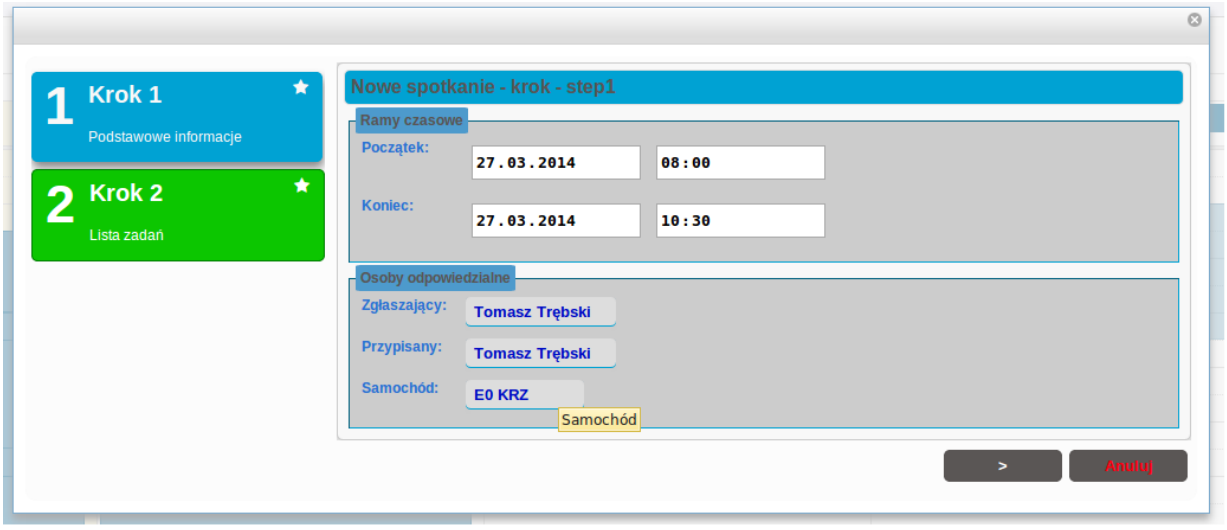


Rysunek 8: *Kreator nowego samochodu - numer VIN, źródło: opracowanie własne*

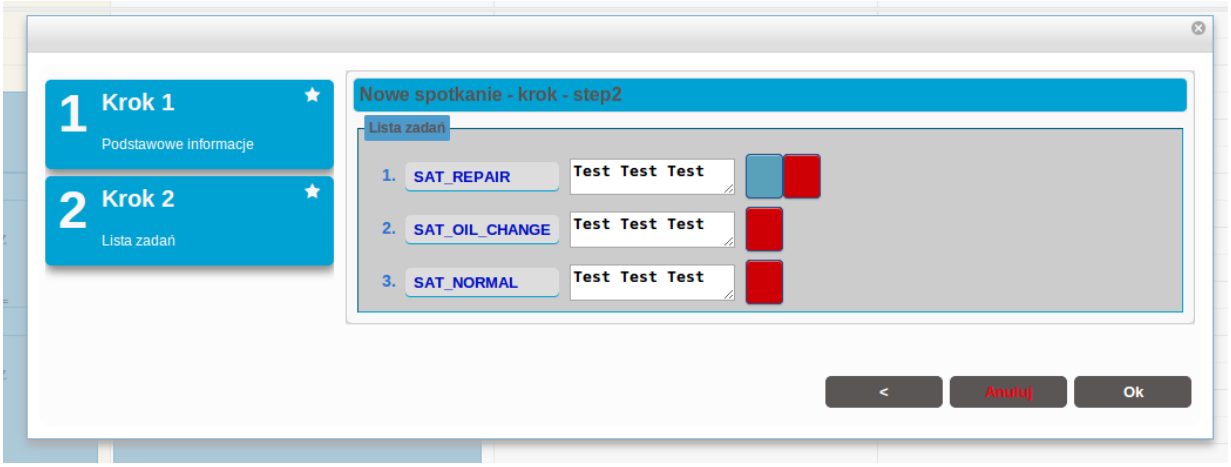


Rysunek 9: *Kreator nowego samochodu - pozostałe dane, źródło: opracowanie własne*

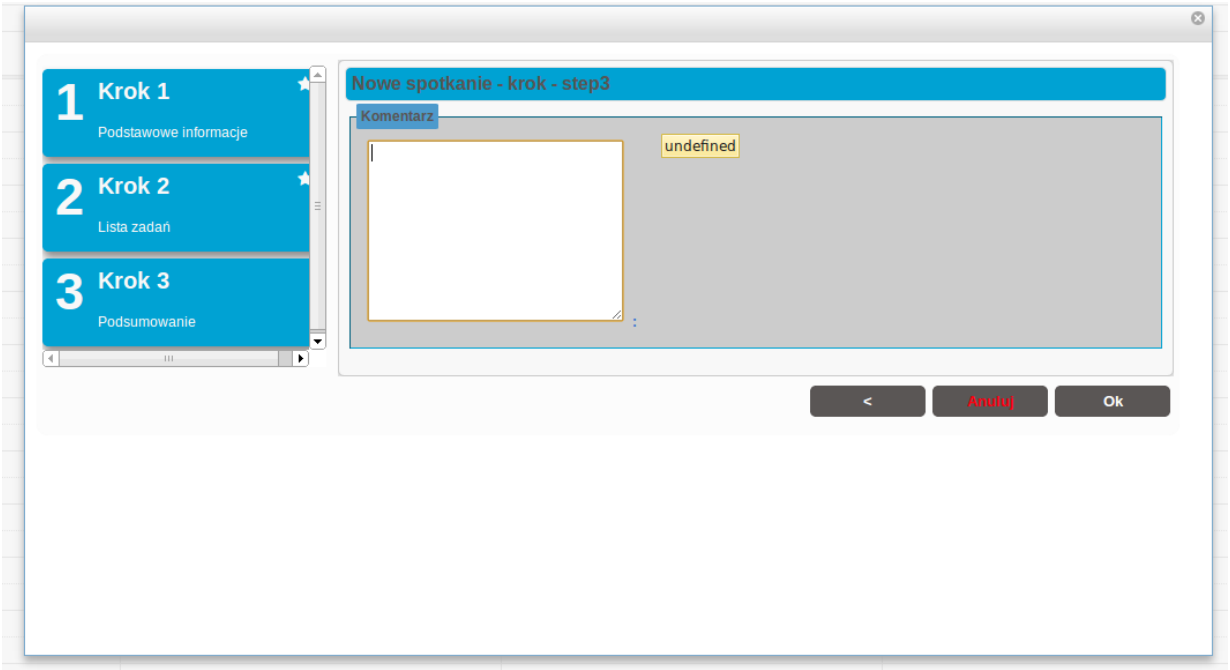
4.3.4. Kreator nowego spotkania



Rysunek 10: *Kreator nowego spotkania - krok 1, źródło: opracowanie własne*



Rysunek 11: *Kreator nowego spotkania - krok 2, źródło: opracowanie własne*



Rysunek 12: *Kreator nowego spotkania - krok 3, źródło: opracowanie własne*

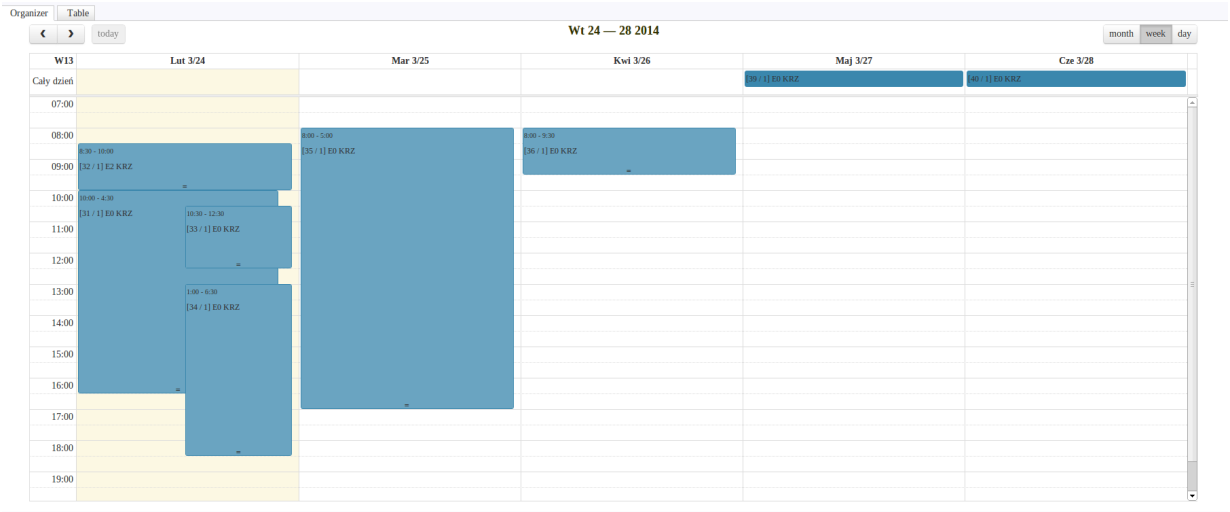
Dane wejściowe są walidowane pod kątem logiki biznesowej:

- 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

4.4. Terminarz spotkań

Organizator spotkań jest komponentem wspierającym zarządzanie 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 13: *Komponent kalendarza wspierający organizację spotkań - organizer, źródło: opracowanie własne*

Rysunek 14: *Komponent kalendarza wspierający organizację spotkań - tabela, źródło: opracowanie własne*

Z poziomu organizera możliwe jest natomiast otwieranie 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.

4.5. Generyczne moduły

Praca nad niewidoczną dla użytkownika końcowego częścią generycznych modułów zaowocowała opracowaniem 3 niezależnych modułów. To czym jest generyczny moduł, najłatwiej opisać na przykładzie jednej z metody programowania - programowania uogólnionego. Koncepcja ta oparta jest o założenie, że wiele fragmentów kodu jest niepotrzebnie powtarzanych tylko dlatego, że służą do przetwarzania różnych danych w identyczny sposób. Podejście generyczne pozwala na tworzenie ogólnych algorytmów, gdzie wymagania dotyczące danych, są nieokreślone lub zdefiniowane na poziomie abstrakcyjnych typów danych.

4.5.1. Strony obiektów domenowych oraz tabele

Podczas analizy wymagań warstwy biznesowej aplikacji demonstracyjnej stało się jasne, że podejście do reprezentacji danych w formie tabel, czy też wyświetlenia informacji o pojedynczym obiekcie domenowym, nie będzie operacją trywialną. Kluczowe było wybranie efektywnego sposobu reprezentacji obiektu lub obiektów domenowych w wybranym formacie. Z uwagi na przyjęte wymaganie funkcjonalne oraz w myśl zasady **DRY** (1), tworzenie kolejnych klas z kodem odpowiedzialnym za powstawanie struktury tabeli lub strony internetowej było niedopuszczalne. Nie udało się również znaleźć żadnej biblioteki wspierającej taką funkcjonalność. Strony obiektów domenowych, zwane dalej **info page**, oraz tabele, to moduły generyczne ponieważ:

- algorytm pobierania danych jest niezależny od danych na poziomie typów danych, tj. możliwe było jego zaimplementowanie w abstrakcyjnej klasie generycznej, nie posiadającej na etapie kompilacji żadnej informacji o konkretnym typie obiektów, jakie klasa będzie przetwarzać,
- algorytm zdefiniowany jest raz,
- algorytm jest elastyczny,
- komponenty są w stanie samodzielnie skonwertować dane do typu prostego (liczba, wartość logiczna lub łańcuch znakowy), który można bezproblemowo zaprezentować po stronie klienta,
- finalne implementacje odpowiedzialne są jedynie za:
  - dostarczenie informacji o budowie komponentu,
  - dostarczenie wartości dla atrybutów dynamicznych, nie związanych z typem przetwarzanych danych

Artefaktem definiującym powyższe założenie, będącym korzeniem całej hierarchii klas tych komponentów, jest interfejs **ComponentBuilder** (listening 17). To właśnie ten interfejs jest następnie używany w kontrolerach odpowiedzialnych za mapowanie żądań pobrania definicji lub danych z poszczególnych komponentów. Same kontrolery nie zależą bezpo-

średnio od konkretnych implementacji, ale opierają się na **API** wspólnym dla wszystkich obiektów typu **ComponentBuilder**. Szczególnie ważne są tutaj następujące metody:

- **getDefinition()** - zadaniem tej metody jest zwrócenie struktury, definicji danego komponentu,
- **getData()** - zadaniem tej metody jest zwrócenie danych danego komponentu.

```
1  public interface ComponentBuilder<COMP extends Serializable> {
2      String getId();
3
4      Class<?> getBuilds();
5
6      ComponentBuilds.Produces getProduces();
7
8      ComponentDataResponse<?> getData() throws ComponentException;
9
10     COMP getDefinition() throws ComponentException;
11
12     void init(ComponentDataRequest componentDataRequest);
13 }
```

Listing 17: **ComponentBuilder** - korzeń hierarchii modułu komponentów, który opisuje rolę tego rodzaju obiektów w systemie, źródło: opracowanie własne

Problem wielu zapytań AJAX

Aplikacja internetowa korzysta z zapytań do serwera aby pobierać treść, dane oraz zasoby, które mogą zostać zaprezentowane użytkownikowi. Z każdym zapytaniem wiążą się jednak koszty, których nie sposób jest uniknąć. Narzut związany z inicjacją zapytania, odpowiedzią serwera, jej wielkością oraz samym czasem odpowiedzi jest kwestią, którą należy uwzględnić. Problemem, który pojawił się podczas projektowania algorytmu renderowania **info page** oraz tabeli wynikał z uogólnionej koncepcji według, której miały one działać. Niemniej, bez wiedzy o typie obiektu domenowego, którego atrybuty miały być przedstawione na stronie lub którego obiekty miały być wyświetlone w tabeli, nie było możliwości wstępnego przygotowania struktury takiego komponentu. Jedną z możliwości rozwiązania tego problemu było wykonanie dwóch oddzielnych zapytań do serwera, najpierw po definicję, a następnie po dane. Inną ewentualnością było wystosowanie pojedynczego zapytania, gdzie odpowiedź miałaby zawierać informacja o charakterystyce i danych. W drugim przypadku kwestią dyskusyjną stawał się rozmiar odpowiedzi, a także format w jakim przedstawiona musiałaby być definicja, aby możliwe było jej efektywne przetworzenie na odpowiednią strukturę DOM.

Wyjściem z sytuacji było skorzystanie z możliwości dynamicznego definiowania zawartości stron HTML, budowanych poprzez pliki JSP. Idea zakładała wykorzystanie mechanizmów dostępnych tylko i wyłącznie po stronie serwera, takich jak **tagi JSTL** i serwisy aplikacji, do utworzenia struktury DOM i odesłania gotowego dokumentu w odpowiedzi na żądania. Alternatywą byłoby budowanie kolejnych elementów strony przez **JavaScript**, co, nawet przy wykorzystaniu bibliotek takich jak **jQuery**, byłoby operacją mozolną i podatną na błędy. W kontekście omawianych komponentów, wybrane podejście zostało

zastosowane zarówno dla stron obiektów domenowych, jak i tabel. Jedynie w przypadku **info page**, również dane, umieszczane są w strukturze dokumentu po stronie serwera.

Problem źródła danych

Główna funkcjonalność tej części aplikacji demonstracyjnej sprowadza się do wydajnej i rozszerzalnej zmiany reprezentacji pewnej informacji na inną. Nie byłoby to możliwe bez źródła danych. Problemem z repozytorium, które byłoby w stanie sprostać zadaniu pobrania obiektów konkretnego typu, sprowadzał się właśnie do typu. W myśl idei jaka przyświeca programowaniu uogólnionemu, to nie dane są ważne, ale algorytm ich przetwarzania. Niemniej, dzięki wsparciu biblioteki **Spring Data JPA** (3.3.3), kwestia ta została łatwo rozwiązania. Wspomniane repozytoria, w kontekście **Spring Data JPA**, są niczym innym jak generycznymi interfejsami. Wykorzystując ten fakt oraz własność typów generycznych języka Java, gdzie typ uogólniony, znajdujący się w deklaracji klasy, może zostać użyty w polach oraz metodach tej klasy (listining 18), repozytoria zdefiniowane zostały jako pola poszczególnych komponentów. Ostatecznie, korzystając z funkcjonalność szkieletu aplikacji **Spring**, pozwalając na wstrzykiwania zależności generycznych klas. Zachowane zostało bezpieczeństwo typów, a generyczne algorytmy uzyskały możliwość korzystania ze źródła danych.

```
1  abstract public class TableComponentBuilder<COMP extends TableComponent, Y extends Persistable<?>>
2                  extends DefaultComponentBuilder<COMP>
3                  implements EntityAware<Y> {
4
5          private static final String CACHE_NAME       = "org_agatom_springatom_tableBuilders";
6          private static final long   serialVersionUID = -6830218035599925554L;
7          @Autowired
8          protected SBasicRepository<Y, Long> repository;
```

Listing 18: Typy generyczne w deklaracji oraz polach klasy w języku Java, źródło: opracowanie własne

Problem identyfikacji ComponentBuilder

Komponenty zostały tak zaprojektowane, aby zminimalizować poziom zależności między konkretnymi instancjami **ComponentBuilder** oraz odpowiadającym im elementów interfejsu użytkownika. Problematyczne okazało się zdefiniowanie powiązania pozwalającego nawigować między oboma artefaktami. Rozwiązanie oparte zostało na adnotacjach języka Java oraz zdolności szkieletu aplikacji **Spring** do wybierania obiektów opatrzonych konkretnymi adnotacjami spośród wszystkich zdefiniowanych.

```
1  public @interface ComponentBuilds {
2      String id();
3
4      Class<?> builds();
5
6      Produces produces() default Produces.PAGE_COMPONENT;
7
8      public static enum Produces {
9          PAGE_COMPONENT,
10         TABLE_COMPONENT
11     }
12
13 }
```

Listing 19: **ComponentBuilds** - adnotacja opisująca **ComponentBuilder**, źródło: opracowanie własne

ComponentBuilds pozwala na zdefiniowanie następujących informacji:

- unikatowy klucz pod którym obiekt istnieje w aplikacji,
- typ budowanego komponentu: strona lub tabela

Dzięki **EntityBased** dany komponent zostaje ściśle powiązany z pewną klasą biznesowego modelu danych, jako artefakt zdolny utworzyć reprezentację obiektu lub obiektów tej klasy. Wszystkie te informacje pozwalają na pobranie obiektu **ComponentBuilder** skonkretyzowanego na przygotowanie tabeli, której kolejne wiersze odpowiadają konkretnym obiektom domenowym lub gotowego na utworzenie definicji strony domenowej.

```
1 @Inherited
2 @Documented
3 @Target(value = {ElementType.TYPE})
4 @Retention(value = RetentionPolicy.RUNTIME)
5 public @interface EntityBased {
6     Class<?> entity();
7 }
```

Listing 20: **EntityBased** - adnotacja opisująca klasę obiektu domenowego, z którą związana jest konkretny **ComponentBuilder**, źródło: opracowanie własne

4.5.2. InfoPage - strony obiektów modelu danych

InfoPage jest komponentem budującym reprezentację obiektu domenowego jako strony internetowej. Składają się na niego artefakty opisujące strukturę strony, skonkretyzowana, pod kątem przetwarzania danych, implementacja interfejsu **ComponentBuilder** (17), klasy pomocnicze, usprawniające proces budowania struktury i pozyskiwania definicji komponentów na podstawie unikatowego klucza, klasy obiektu domenowego związanej z daną stroną lub adresu wpisanego w oknie przeglądarki.

Definicja komponentu InfoPage

Pełna definicja **InfoPage** składa się z dwóch elementów: rozszerzenia klasy **EntityInfoPageComponentBuilder** i **SEntityInfoPage**. **SEntityInfoPage** (listining 21) dostarcza metadanych o:

- klasie obiektu domenowego,
- unikatowym identyfikatorze strony,
- unikatowym kluczu będącym częścią adresu danej strony

Podczas startu, aplikacja pobiera konkretne implementacje **SEntityInfoPage**, które istnieją w programie jako obiekty, dzięki wsparciu szkieletu aplikacji **Spring**, który automatycznie tworzy instancje opatrzonych odpowiednią adnotacją (**Component**, **Service** lub **Repository**). Dzięki temu procesowi możliwe jest odwoływanie się do meta obiektu **InfoPage** poprzez dowolny atrybut opisujący go.

```
public interface SEntityInfoPage
        extends SInfoPage {
    Class<?> getDomain();

    String getPath();

    String getRel();
}
```

Listing 21: **SEntityInfoPage** - interfejs opisujący metadane związane ze stroną domenową, źródło: opracowanie własne

Algorytm pobrania danych dla strony domenowej

EntityInfoPageComponentBuilder jest implementacją **ComponentBuilder** (17) zawierającą skonkretyzowany algorytm pobierania wartości atrybutów, będących częścią danej strony. Jego zadaniem jest również przetworzenie danych na reprezentacją, którą będzie można wyświetlić po stronie klienta, jako prosty łańcuch znakowy lub liczbę. Widoczna w interfejsie **ComponentBuilder** metoda **getDefinition** pozostaje na tym poziomie nie zaimplementowana. Przyczyna tego leży w odmienności poszczególnych stron obiektów domenowych, sprowadzającej się do różnej ich struktury, różnych atrybutów, które będą widoczne i ostatecznie do innej klasy biznesowego modelu danych, skojarzonej z komponentem **InfoPage**.

Rozwiązanie problemu reprezentacji danych

W przypadku strony domenowej atrybutami, które wymagały rozwiązania niespójności typów danych, były relacje klucz obcy - klucz główny. Sytuacja kiedy obiekt, będący podmiotem danej strony, zależy od innego obiektu, została rozwiązania jako przedstawienie takiej zależności w postaci hiperłącza do strony domenowej, jeśli takowa została zdefiniowana (innymi słowy, jeśli w systemie można było znaleźć definicję klasy **SEntityInfoPage** oraz odpowiadającej jej implementacji **EntityInfoPageComponentBuilder**. Odwrotna sytuacja miała miejsce, kiedy podmiot strony był zależnością dla innych obiektów domenowych. Rozwiązana została ona poprzez przekazanie, jako wartości dla atrybutu, informacji niezbędnych do załadowania tabeli. Analogicznie do pierwszego przypadku, również w tym momencie, koniecznie było najpierw ustalenie, czy w aplikacji istnieje odpowiednia implementacja klasy **TableComponentBuilder**. Problem nieczytelnej wartości typów wyliczeniowych został rozwiązany z wykorzystaniem specjalnego pliku, zawierającego listę wszystkich wartości wszystkich typów wyliczeniowych wraz z ich czytelną wartością.

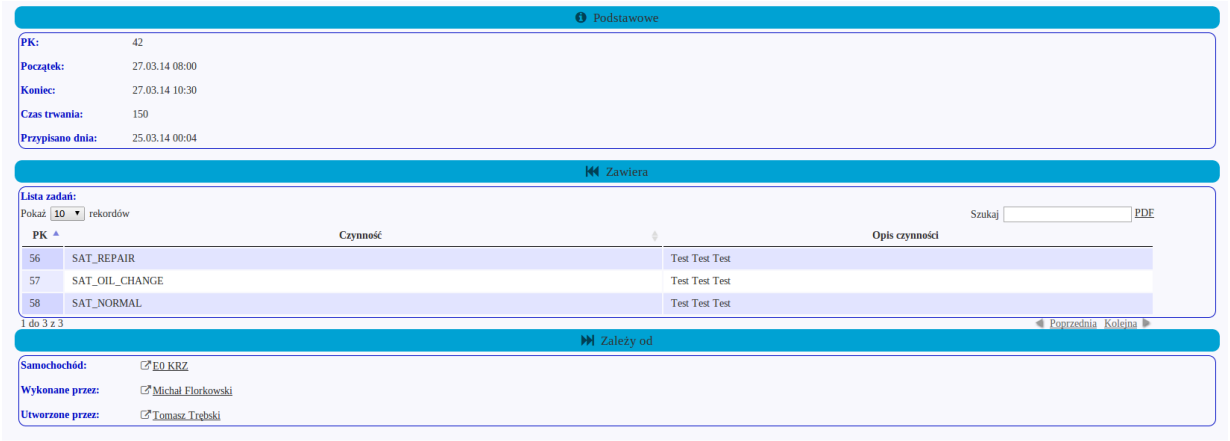
```
    @Override
    protected ComponentDataResponse<?> buildData(final ComponentDataRequest dataRequest)
            throws ComponentException {
        final Long objectId = dataRequest.getLong("objectId");
        final Y object = this.repository.findOne(objectId);

        this.logger.trace(
                String.format("processing object %s=%s",
                        ClassUtils.getShortName(object.getClass()),
                        object.getId()
                )
        );

        return new EntityInfoPageResponse() {
            private static final long serialVersionUID = 5093004619980504166L;

```

```
17                  @Override
18                  public Object getValueForPath(final String path) throws ComponentException {
19                          logger.trace(String.format("processing path %s", path));
20                          Object value = InvokeUtils.invokeGetter(object, path);
21
22                          if (value != null) {
23                                  final InfoPageAttributeComponent attributeComponent = getAttributeForPath(path);
24
25                                  switch (attributeComponent.getDisplayAs()) {
26                                          case INFOPAGE: {
27                                                  value = buildInfoPageLink(path, value);
28                                          }
29                                          break;
30                                          case TABLE: {
31                                                  value = buildTableBuilderLink(path, value, object);
32                                          }
33                                          case VALUE: {
34                                                  // update only if value is Enumeration
35                                                  if (ClassUtils.isAssignableValue(Enum.class, value)) {
36                                                          value = messageSource.getMessage(
37                                                                          ((Enum<?>) value).name(),
38                                                                          LocaleContextHolder.getLocale()
39                                                          );
40                                                  }
41                                          }
42                                  }
43                          }
44
45                          value = verifyAgainstNullAndSetDefaultIfSo(path, value, object);
46
47                          logger.trace(String.format("processed path %s to %s", path, value));
48                          return value;
49                  }
50          }.setValue(object);
51      }
52
```

Listing 22: **EntityInfoPageComponentBuilder** - implementacja pozyskania danych dla strony obiekty domenowego, źródło: opracowanie własne

Utworzenie nowej strony sprowadza się sprowadza się do zaimplementowania **EntityInfoPageComponentBuilder**, oznaczenia nowej klasy adnotacjami: **ComponentBuilds** (listing 19) i **EntityBased** (listing 20) oraz utworzenia nowej implementacji interfejsu **SEntityInfoPage**.

Proces renderowania strony domenowej



Rysunek 15: *Strona domenowa dla spotkania, źródło: opracowanie własne*

Na rysunku 15 pokazana została strona domenowa obiektu opisującego pojedyncza wizytę w warsztacie samochodowym. Zdefiniowane zostały 3 panele z atrybutami. Panel podstawowy oraz dwa panele 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 23 przedstawia fragmentu kodu Java odpowiedzialny za zbudowania struktury strony.

```
1   protected InfoPageComponent buildDefinition() {
2       final InfoPageComponent cmp = new InfoPageComponent();
3       this.populateBasicPanel(helper.newBasicPanel(cmp, LayoutType.VERTICAL));
4       this.populateTablePanel(helper.newOneToManyPanel(cmp, LayoutType.VERTICAL));
5       this.populateInfoPagePanel(helper.newManyToOnePanel(cmp, LayoutType.VERTICAL));
6       return cmp;
7   }
8
9   private void populateInfoPagePanel(final InfoPagePanelComponent panel) {
10      this.helper.newLinkAttribute(panel, "car", this.getEntityName());
11      this.helper.newLinkAttribute(panel, "assignee", this.getEntityName());
12      this.helper.newLinkAttribute(panel, "reporter", this.getEntityName());
13  }
14
15  private void populateBasicPanel(final InfoPagePanelComponent panel) {
16      this.helper.newAttribute(panel, "id", "persistentobject.id", AttributeDisplayAs.VALUE);
17      this.helper.newValueAttribute(panel, "begin", this.getEntityName());
18      this.helper.newValueAttribute(panel, "end", this.getEntityName());
19      this.helper.newValueAttribute(panel, "interval", this.getEntityName());
20      this.helper.newValueAttribute(panel, "assigned", this.getEntityName());
21  }
22
23  private void populateTablePanel(final InfoPagePanelComponent panel) {
24      this.helper.newTableAttribute(panel, "tasks", this.getEntityName());
25  }
26 }
```

Listing 23: Strona domenowa dla spotkania - kod źródłowy, źródło: opracowanie własne

Proces, w wyniku którego wywołana została metoda z listingu 23, tworząca strukturę strony, oraz wywołujący metodą pokazaną na listingu 22, która zwraca dane dla konkretnych atrybutów, ilustruje poniższa tabela:

Tabela 4: Proces renderowania strony domenowej

Krok	Opis
1	Zdarzeniem inicjującym proces renderowania strony domenowej jest wpisanie w przeglądarce adresu, który zostaje rozpoznany przez aplikację jako wskazujący na komponent **InfoPage**. Jest to możliwe ponieważ adres komponentu posiadana określony format: **/ip/{klucz strony}/{klucz główny}/{wersja}**. Najważniejszym elementem adresu jest **klucz strony** (listining 21).
2	Żądanie zostaje zmapowane do metody **getInfoPageView** (listining 24). Jej zadaniem jest pobranie meta danych strony domenowej na podstawie unikatowego klucza strony, wspomnianego w punkcie 1. Obiekt klasy **SInfoPage** zostaje umieszczony w odpowiedzi razem z hiperłączem, pod którym dostępny będzie dokument HTML zawierający gotową stronę **InfoPage**.
3	Po stronie klienta, kod JavaScript zajmuje się przygotowaniem asynchronicznego żądania po gotowy widok strony, dostępny pod adresem z punktu 2.
4	Żądanie, wysłane z punkcie 3, zostaje zmapowane do metody **getInfoPageViewData** (listining 24). Metoda pobiera instancję klasy **ComponentBuilder**, która spełnia następujące wymagania: • buduje strukturę strony domenowej, • związana jest z typem obiektu domenowego Znaleziony **ComponentBuilder** jest następnie umieszczany w odpowiedzi razem z widokiem, będącym szablonem (listing 25) przyszłej strony. Szablon zaprojektowany został dla jednoczesnego tworzenia dokumentu HTML, zawierającego atrybuty oraz odpowiadające im wartości. To właśnie z poziomu pliku JSP, w przypadku stron obiektów domenowych, wywoływane są metody interfejsu **ComponentBuilder** pozwalające na pobranie jej struktury oraz danych.

```
1
2     @RequestMapping(value = "/{path}/{id}", method = RequestMethod.GET)
3     public ModelAndView getInfoPageView(@PathVariable("path") final String path,
4                                         @PathVariable("id") final Long id)
5                 throws InfoPageNotFoundException {
6         LOGGER.debug(String.format("/getInfoPageView/path=%s/id=%s", path, id));
7         final SInfoPage page = this.getInfoPageForPath(path);
8         if (page != null) {
9             LOGGER.trace(String.format("Resolved infoPage => %s for path => %s", page, path));
10            final ModelMap modelMap = new ModelMap();
11            modelMap.put(InfoPageConstants.INFOPAGE_PAGE, page);
12            modelMap.put(InfoPageConstants.INFOPAGE_VIEW_DATA_TEMPLATE_LINK,
13                         this.getViewDataTemplateLink());
14            return new ModelAndView(
15                    this.getViewForPage(page),
16                    modelMap
17            );
18        }
19        throw new InfoPageNotFoundException(path);
20    }
21
```

```
22      @RequestMapping( value = "/template/render", method = RequestMethod.POST,
23              produces = {MediaType.TEXT_HTML_VALUE},
24              consumes = {MediaType.APPLICATION_JSON_VALUE})
25      public ModelAndView getInfoPageViewData(@RequestBody final DataBean data, final WebRequest request)
26              throws InfoPageNotFoundException {
27          LOGGER.trace(String.format("/getInfoPageViewData -> data=%s, request=%s", data, request));
28          final String rel = data.get("infopage").getValue();
29          final SInfoPage page = this.infoPageMappings.getInfoPageForRel(rel);
30          if (page != null) {
31              final InfoPageComponentBuilder builder = (InfoPageComponentBuilder) this.builders
32                      .getBuilder(page.getClass(), data.toModelMap(), request);
33              return new ModelAndView(DATA_VIEW_NAME,
34                      new ModelMap(InfoPageConstants.INFOPAGE_BUILDER, builder));
35          }
36          throw new InfoPageNotFoundException(rel);
37      }
```

Listing 24: SVInfoPageController - kontroler obsługujący żądania komponentu **InfoPage**

```
1               <span class="x-ip-attr-value">
2                   <c:if test="${attr.valueAttribute}">
3                       <s:eval expression="data.getValueForPath(attr.path)"/>
4                   </c:if>
5                   <c:if test="${attr.tableAttribute}">
6                       <s:eval expression="data.getValueForPath(attr.path)" var="builderContextLink" scope="page"/>
7                       <script>
8                           $(function () {
9                               $('#' + '${attrHolderId}').find('span.x-ip-attr-value').loadBuilderView({
10                                  url : '${builderContextLink.link.href}',
11                                  data: {
12                                      contextKey  : '${builderContextLink.contextKey}',
13                                      contextClass: '${builderContextLink.contextClassName}',
14                                      builderId   : '${builderContextLink.builderId}'
15                                  }
16                              });
17                          });
18                      </script>
19                  </c:if>
20                  <c:if test="${attr.infoPageAttribute}">
21                      <s:eval expression="data.getValueForPath(attr.path)" scope="page" var="_link"/>
22                      <ip:renderInfoPageLink link="${_link}"/>
23                  </c:if>
24                  <c:if test="${attr.emailAttribute}">
25                      <s:eval expression="data.getValueForPath(attr.path)" scope="page" var="_link"/>
26                      <ip:renderEmail link="${_link}"/>
27                  </c:if>
28              </span>
```

Listing 25: Fragment szablonu strony obiektu domenowego, tworzący ciało tabeli

4.5.3. TableBuilder

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. Każda z nich opisana jest zarówno przez zbiór danych, jak i przez zbiór kolumn. Aby poprawnie zdefiniować tabelę wymagane jest podanie obu zbiorów. Istnieją dwie całkowicie odmienne koncepcje tego zagadnienia: statyczna oraz dynamiczna. W przypadku podejścia statycznego, struktura tabeli jest ręcznie umieszczana w strukturze DOM dokumentu HTML. Takie podejście sprawia jednak, że tabela jest stała, zarówno pod kątem tego co wyświetla,

jak również w jaki sposób to robi, każda modyfikacja zbioru danych, wymaga więc ręcznej aktualizacji. Dynamiczną realizację tej kwestii można podzielić na przypadki, gdzie:
1. struktura jest statyczna, źródło danych jest dynamiczne,
2. struktura oraz dane są dynamiczne

Problem reprezentacji danych
Problematyczne w obu podejściach nie okazuje się jednak ani wprowadzenie schematu mapującego poszczególne kolumny na odpowiadające im pola danych, ani też automatyzacja procesu wpisującego poprawną, w rozumieniu języka HTML, definicję tabeli do struktury DOM dokumentu. W przypadku pliku JSP rozwiązanie wymagałoby by ustalenia zbioru kolumn, pobrania danych z repozytorium, ręcznego wpisania statycznych elementów tabeli, takich jak na przykład nagłówek i utworzenia ciała tabeli (kolejnych wierszy) na podstawie zbioru danych. Inne podejście zakładałoby wykorzystanie biblioteki JavaScript dostarczającej dodatkowej warstwy abstrakcji nad procesem definiowania tabeli oraz zdolnej do wygenerowania zapytania Ajax pod wskazany adres, pod którym dostępny byłby zbiór danych. Problemem jest reprezentacja danych. Poniższe zestawienie opisuje wybrane przypadki, gdzie pokazane zostały potencjalne rozwiązania, a rzeczywista i oczekiwana reprezentacja danych jest od siebie różna:

Tabela 5: Zestawienie problemów reprezentacji danych dla tabel

Problem	Potencjalne rozwiązanie
Relacja klucz główny - klucz obcy	Wykonanie następnego zapytania do repozytorium danych, pobranie powiązanego obiektu. Kolejną niewiadomą w tym przypadku jest niestety to, jaki atrybut powiązanego obiektu wybrać, który jednoznacznie by go opisywał, a w pierwszej kolejności wiedza, które z atrybutów należy interpretować jako relacje.
Data jako znacznik czasowy	Przechowanie daty jako znaczników czasowych jest zalecane, ponieważ jest to rozwiązanie przenośne, pozwalające na uniknięcie problemu operowania na datach jako łańcuchach znakowych, w których format zapisu jest kwestią zależną od programisty i podatną na błędy. Utworzenie zrozumiałej reprezentacji wymagałoby by wiedzy o tym, które kolumny krotki bazy danych, to tak naprawdę daty.
	Następna strona...

Tabela 5 – kontynuacja...

Problem	Potencjalne rozwiązanie
Typy wyliczeniowe	**Typ wyliczeniowy**, w kontekście języka Java, to specjalny rodzaj klasy, pozwalający na tworzenie stałych w trakcie kompilacji programu, które, w sposób niezmienny, opisują pewną właściwość obiektu, z którą są związane. Na poziomie bazy danych możliwe jest przechowanie wartości typu wyliczeniowego jako łańcucha znakowego. Problemem jest tutaj format w jakim kolejne pozycje typu wyliczeniowego są definiowane. Najczęściej pisane dużymi literami, nie posiadając spacji nie są dobrym kandydatem do bezpośredniego zaprezentowania w tabeli. Ponownie, rozwiązanie tego problemu, wymagałoby wiedzy nie tyle o tym, które z atrybutów należy interpretować jako pozycje typu wyliczeniowego, ale także tego, o który typ wyliczeniowy chodzi oraz danych, opisujących jego wartość w sposób zrozumiały dla użytkownika.

TableComponentBuilder

TableComponentBuilder jest implementacją interfejsu **ComponentBuilder** (listing 19), który został zaprojektowany do budowany definicji oraz zbioru danych dla tabel. Rozwiązuje on problemy zdefiniowane w tabeli 5 oraz dostarcza sposobu wsparcia następujących aspektów pracy z tabelami:

- wsparcie dla sortowania po stronie serwera,
- wsparcie dla filtrowania po stronie serwera,
- wsparcie dla dynamicznych kolumn, nie związanych bezpośrednio z typem obiektów, które tabela wyświetla

Działa on w oparciu o bibliotekę **Dandelion Datatables** (3.4.7). Wykorzystanie biblioteki **Dandelion**, pozwoliło na ograniczenie ilości kodu niezbędnego do poprawnego wpisania definicji tabeli do struktury dokumentu HTML. Odpowiedzialność za ten proces została oddelegowana do wykorzystanej biblioteki, podobnie jak wykonanie asynchronicznego zapytania pod adres, pod którym znajduje się zbiór danych dla tabeli.

Algorytm budowania zbioru danych

Algorytm odpowiedzialny za utworzeniu zbioru danych jest podobny do tego, który zdefiniowano dla **InfoPage** (listing 22). Różnice polegają na formacie w jakim dane są zwracane oraz w sposobie generowania zapytania do repozytorium danych. Podczas gdy strona domenowa była związana z pojedynczym obiektem, tabela związana jest z pewną ich liczbą. Dlatego też algorytm został skonkretyzowany na zwrócenia danych jako kolekcji, gdzie kolejne jej elementy odpowiadają kolejnym wierszom tabeli. Uwzględnia on także stronnicowanie, które polega na zwróceniu jedynie pewnego wycinka danych, zamiast wszystkich możliwych rekordów z bazy danych.

```
final Predicate predicate = this.getPredicate(dc);
final long countInContext = predicate != null ?
                this.repository.count(predicate) :
                this.repository.count();

final List<Map<String, Object>> data = Lists.newArrayList();

if (countInContext != 0) {
        // some data to process
        final Page<Y> all = this.getAllEntities(dc, predicate, countInContext);

        for (final Y object : all) {
                final Map<String, Object> map = Maps.newHashMap();
                for (ColumnDef columnDef : dc.getColumnDefs()) {

                        final String path = columnDef.getName();
                        Object value = InvokeUtils.invokeGetter(object, path);
                        if (value == null) {
                                value = this.handleDynamicColumn(object, path);
                        }
                        value = verifyNullAndSetDefaultIfSo(object, path, value);
                        map.put(path, value);

                }
                if (!map.isEmpty()) {
                        data.add(map);
                }
        }

}

final ComponentDataResponse<DataSet<Map<String, Object>>> response = new ComponentDataResponse<>();
response.setValue(new DataSet<>(data, (long) data.size(), countInContext));

return response;
```

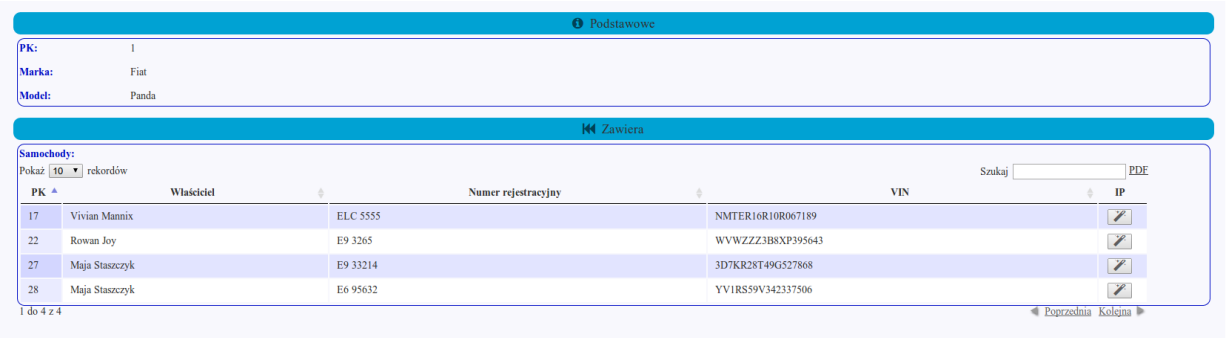
Listing 26: **TableComponentBuilder** - fragment implementacji algorytmu pozyskania zbioru danych dla tabel

Warto w tym miejscu wspomnieć o podwójnym trybie działania algorytmu. W momencie, gdy tabela działa w kontekście komponentu **InfoPage**, nadrzędnym obiektem, zwanym dalej kontekstem, jest obiekt z którym skojarzona jest strona obiektu domenowego. Dla tego przypadku, tabela zwraca zbiór danych, który dla wspomnianego kontekstu jest wynikiem transformacji zależności klucz główny - klucz obcy. Z drugiej strony, komponent tabeli może zostać umieszczony na podstronie, gdzie kontekst nie jest zdefiniowany. W takim wypadku, budowany zbiór danych nie jest zawężany do żadnej relacji między poszczególnymi obiektami. Widoczne na listingu 26 wywołanie metody **getPridacate** wykorzystuje funkcję **isInContext** (listing 27), która rozpoznaje czy kontekst jest obecny i tym samym decyduje o zachowaniu tabeli. Jeśli kontekst istnieje tworzony jest predykat. Zadanie to realizowane jest na poziomie finalnych implementacji klas.

```
private boolean isInContext(final ComponentTableRequest dc) {
        return (dc.getContextKey() != null && !dc.getContextKey().isEmpty()) && (dc.getContextClass() != null);
}

```

Listing 27: **isInContext** - metoda **TableComponentBuilder** określająca tryb pracy komponentu dla algorytmu budowania zbioru danych, źródło: opracowanie własne

Proces renderowania tabeli



Rysunek 16: *Tabela wyświetlająca wszystkie pojazdy danej samochodu* ***Fiat Panda****, źródło: opracowanie własne*

Tabela na rysunku 16 odpowiada implementacja klasy **TableComponentBuilder** widocznej na listingu 28. Posiada ona 5 kolumn, z których 4 utworzone zostały w metodzie **buildDefinition** klasy **CarsTableBuilder**. Ostatnia kolumna dodawana jest domyślnie do każdej tabeli, jeśli dla obiektu, stanowiącego de facto źródło danych pojedynczego wiersza, istnieje definicja komponentu **InfoPage**. Wspomniana kolumna zawiera akcję, której wywołanie prowadzi do strony obiektu domenowego.

```
1  @EntityBased(entity = SCar.class)
2  @ComponentBuilds(
3          id = CarsTableBuilder.BUILDER_ID,
4          builds = SCar.class,
5          produces = ComponentBuilds.Produces.TABLE_COMPONENT
6  )
7  public class CarsTableBuilder
8          extends TableComponentBuilder<DandelionTableComponent, SCar> {
9
10     protected static final String BUILDER_ID      = "carsTableBuilder";
11     private static final   String TABLE_ID        = String.format("%s%s", "table", StringUtils.uncapitalize(SCar.ENTITY_NAME));
12     private static final   Logger LOGGER          = Logger.getLogger(CarsTableBuilder.class);
13     private static final   long   serialVersionUID = 3079491907844336996L;
14
15     @Override
16     protected Object handleColumnConversion(final SCar object, final Object value, final String path) {
17         switch (path) {
18             case "owner": {
19                 return object.getOwner().getPerson().getIdentity();
20             }
21         }
22         return super.handleColumnConversion(object, value, path);
23     }
24
25     @Override
26     protected Predicate getPredicate(final Long id, final Class<?> contextClass) {
27         return QSCar.sCar.carMaster.id.eq(id);
28     }
29
30     @Override
31     protected Logger getLogger() {
32         return LOGGER;
33     }
34
35     @Override
36     protected DandelionTableComponent buildDefinition() {
```

```
37          final DandelionTableComponent component = this.helper.newDandelionTable(TABLE_ID, BUILDER_ID);
38          this.helper.newTableColumn(component, "id", "persistentobject.id");
39          this.helper.newTableColumn(component, "owner", "scar.owner");
40          this.helper.newTableColumn(component, "licencePlate", "scar.licenceplate");
41          this.helper.newTableColumn(component, "vinNumber", "scar.vinnumber");
42          return component;
43      }
```

Listing 28: *CarsTableBuilder* - klasa definiująca strukturę tabeli wyświetlającej listę samochodów, źródło: opracowanie własne

Proces, w wyniku którego tabela widoczna na rysunku 16, jest dostępna w interfejsie użytkownika, przedstawiony został w poniższej tabeli:

Tabela 6: Proces renderowania tabeli

Krok	Opis
1	Tabela istnieje w kontekście strony domenowej. Zdefiniowany na listingu 25 kod, obsługuje ten przypadek z wykorzystaniem technologii JavaScript. Po stronie klienta generowane jest asynchroniczne zapytanie, które zostaje zmapowane na wywołanie metody **getTableBuilderPost** (listing 29) kontrolera **SVTableBuilderController**. Metoda odszukuje instancję **TableComponentBuilder** na podstawie przekazanego w zapytaniu parametru **builderId**, a następnie umieszcza odnaleziony obiekt w odpowiedzi razem z nazwą widoku.
2	Wybrany w punkcie 2 widok to plik JSP, w którym, z przekazanej instancji obiektu **ComponentBuilder**, pobierana jest definicja tabeli. Uzyskane informacje są, w dalszej części, ustawiane jako atrybuty tagu **JSP** z biblioteki **Dandelion Datables**. Od tego momentu, proces budowania struktury DOM nowej tabeli jest obsługiwany przez wybraną biblioteką. Warto tutaj zwrócić uwagę na atrybut **url**, pod którym udostępnione będą dane dla tabeli. Adres tworzony jest na podstawie unikatowego identyfikatora obiektu **TableComponentBuilder**.
3	**Dandelion Datatables** po stworzeniu struktury tabeli generuje asynchroniczne zapytanie pod podany w punkcie 2 adres. Żądanie zostaje zmapowane na wywołanie metody **getBuilderData** (listing 29) kontrolera **SVTableBuilderController**. Metoda pobiera instancję **ComponentBuilder** i wywołuje metodę **getData**. Dane są następnie zwracane w postaci pliku **JSON** do tabeli po stronie klienta aplikacji.

```
1   @RequestMapping(value = "/inContext", method = RequestMethod.POST,
2               produces = {MediaType.TEXT_HTML_VALUE},
3               consumes = {MediaType.APPLICATION_JSON_VALUE}
4   )
5   public ModelAndView getTableBuilderPost(@RequestBody final DataBean data) {
6       final String builderId = data.get("builderId").getValue();
7       final ComponentBuilder<?> builder = this.builders.getBuilder(builderId);
8       final ModelMap modelMap = new ModelMap(InfoPageConstants.TABLE_COMPONENT_BUILDER, builder);
9       modelMap.addAttribute(InfoPageConstants.INFOPAGE_PARAMS, data.toModelMap());
10      return new ModelAndView(VIEW_NAME, modelMap);
11  }
12
13  @ResponseBody
14  @RequestMapping(value = "/data/{id}")
15  public DatatablesResponse<?> getBuilderData(
16              @PathVariable("id") final String builderId,
17              final ComponentTableRequest tableRequest,
18              final WebRequest request) throws ControllerTierException {
19      final ComponentBuilder<?> builder = this.builders.getBuilder(builderId,
20              new ModelMap(ComponentConstants.REQUEST_BEAN, tableRequest),
21              request
22      );
23      try {
24          if (builder != null && builder instanceof TableComponentBuilder) {
25              LOGGER.trace(String.format("Found builder %s:%s:%s",
26                                      builderId, builder.getId(), builder.getBuilds())
27              );
28              return DatatablesResponse.build(
29                      (DataSet<?>) builder.getData().getValue(), tableRequest.getCriterias()
30              );
31          }
32      } catch (Exception e) {
33          LOGGER.error("/getBuilderData threw exception", e);
34          throw new ControllerTierException(e);
35      }
36      return null;
37  }
```

Listing 29: SVTableBuilderController - kontroler obsługujący żądania komponentu **TableBuilder**, źródło: opracowanie własne

4.5.4. RBuilder - szablony raportów biznesowych

RBuilder jest modułem aplikacji praktycznej, realizującym koncepcję tworzenia szablonów raportów biznesowych, będących opisem struktury gotowego raportu. Generowanie raportów zostało zrealizowane z wykorzystaniem biblioteki **Dynamic Jasper** (3.4.6). Istniejące narzędzia do tworzenia szablonów, zrozumiałych przez tę bibliotekę, nie istnieją jako aplikacje działające w środowisku przeglądarki internetowej. Głównym założeniem funkcjonalnym komponentu **RBuilder** jest więc dostarczenie logiki oraz widoku pozwalających na przygotowanie szablonu w oknie przeglądarki, a następnie wygenerowanie gotowego raportu jako dokumentu **PDF**, **XLS**, **HTML** lub **CSV**.

Model danych - reprezentacja szablonu

Szablon raportu należy rozumieć jako obiekt domenowy **SReport** oraz klasę **ReportConfiguration**. Pierwszy z wymienionych obiektów zapisywany jest w bazie danych i pozwala na stwierdzenie gdzie zapisany został szablon oraz gdzie znajduje się zserializowany obiekt drugiej z wymienionych klas. **ReportConfiguration** dostarcza informacji trudnych do

efektywnego zamodelowania na poziomie bazy danych. Są to informacje opisujące źródło danych:
- wybrane obiekty domenowe,
- wybrane atrybutu obiektów domenowych

oraz strukturę nowego raportu, widoczną dla użytkownika:
- tytuł,
- podtytuł,
- kolumny,
- informacje o kolumnach grupujących, według których dane będą pogrupowane,
- wybrane reprezentacje, w których dane umieszczone w poszczególnych kolumnach, będą zaprezentowane w raporcie

Serwisy - przetwarzanie danych
Serwisy komponentu **RBuilder** dostarczają metod, dzięki którym możliwe jest zapisanie szablonu, wygenerowanie z niego raportu w wybranej przez użytkownika reprezentacji i dostarczenie danych do 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 7: Bloki funkcjonalne modelu serwisów **RBuilder**

Grupa	**Funkcjonalność**
Operation Management	Grupa **Operation Management** odpowiedzialna jest za tworzenie obiektu domenowego **SReport** w zależności od ilości tabel wybranych dla konkretnego raportu. Lista klas: • SingleEntityRBuilderCreateOperation - obsługuje przypadek, w którym źródłem danych dla nowego raportu jest pojedyncza tabela w bazie danych, • MultipleEntitiesRBuilderCreateOperation - obsługuje przypadek, w którym źródłem danych jest wiele tabel.
Następna strona...	

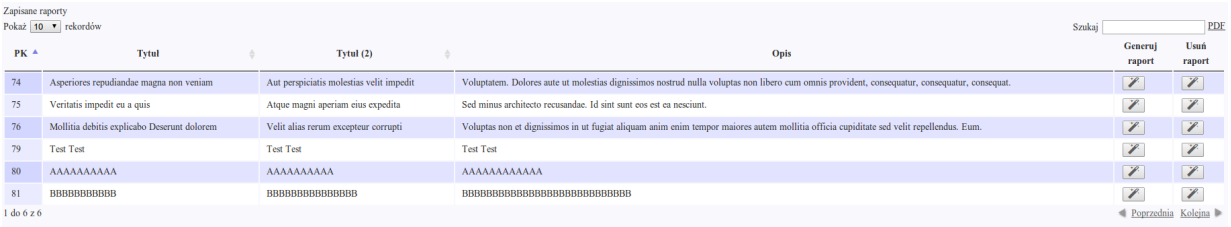
Tabela 7 – kontynuacja...

Grupa	Funkcjonalność
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 przechowywanych w klasie, w odpowiadającej jej polu oraz możliwe ich reprezentacje. Informacje tego typu udostępniane są poprzez interfejs *ReportableColumnResolver*, • 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 szablonu do pliku ***.jasper**. Obecnie wspiera ona przypadek pojedynczej tabeli, jako źródła danych raportu. Jej zadaniem jest utworzenie obiektu klasy **DynamicReport** poprzez ustawienie, na tym obiekcie, następujących informacji: • tytuł, • podtytuł, • opis, • język, • szerokość odpowiednich sekcji jak nagłówek, stopka itp., • lista kolumn, • lista kolumn według których dane mają być grupowane.
	Następna strona...

Tabela 7 – kontynuacja...

Grupa	Funkcjonalność
Generic helper	*ReportBuilderService* jest artefaktem serwisu, przekazującym sterowanie do modułu **Operation Management**, celem utworzenia instancji obiektu domenowego **SReport** oraz wsparcia 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 wybranym przez użytkownika formacie, 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, zrozumiałego przez bibliotekę **DynamicJasper**, na podstawie informacji takich jak lista kolumn, ich typ, wybrany typ reprezentacji danych w kolumnie

Warstwa widoku



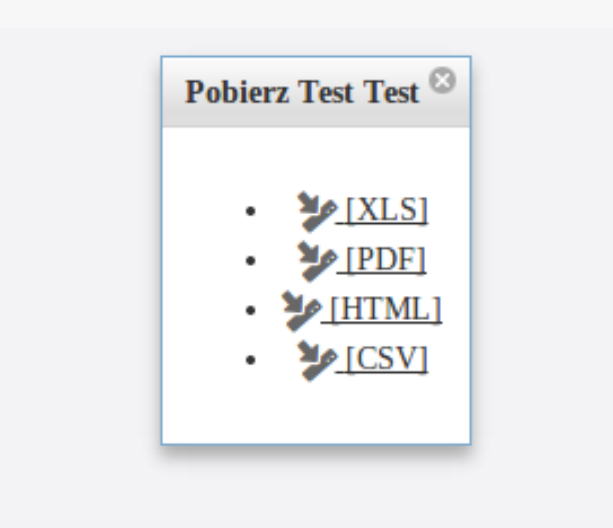
Rysunek 17: *Tabela wyświetlająca zapisane szablony, źródło: opracowanie własne*

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

- tabela z istniejącymi raportami,
- przewodnik tworzenia nowego raportu (opisany w rozdziale **ReportBuilder - tworzenie szablonów raportów biznesowych** 4.3.1)

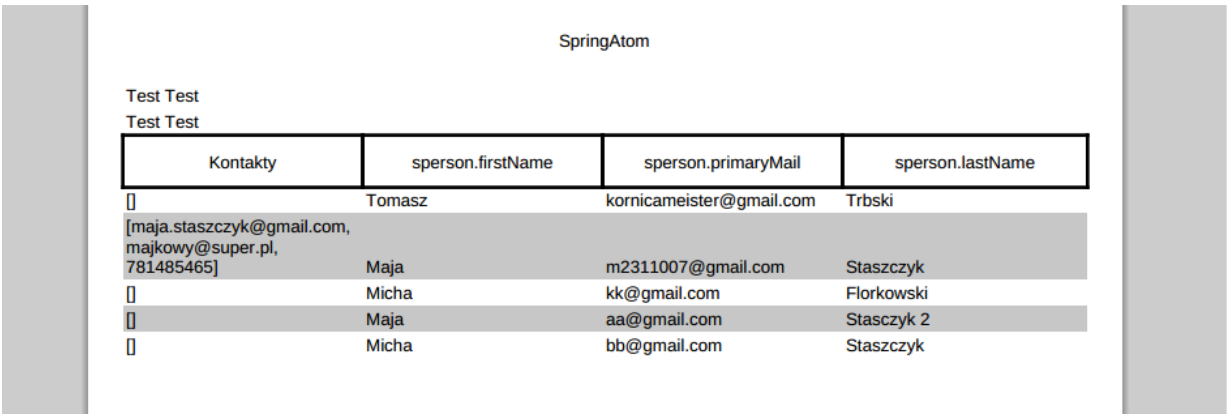
Tabela jest obiektem należącym do warstwy widoku, utworzonym z wykorzystaniem komponentu **TableBuilder** (26). W tabeli widoczne są wszystkie zapisane w systemie szablony. Dodatkowo do każdego z wierszy przypisane zostały akcje pozwalające na:

- generowanie nowego raportu,
- usunięcie szablonu z bazy danych.



Rysunek 18: *Generowanie nowego raportu - wybór docelowego formatu, źródło: opracowanie własne*

Usunięcie jest prostą, z punktu widzenia użytkownika, operacją, 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. Przykładowy raport wygląda w następujący sposób:



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 19: *Gotowy raport utworzony przez komponent **RBuilder**, źródło: opracowanie własne*

4.6. 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, jak także słabo konfigurowalnych części. Plany rozwojowe aplikacji zostały przedstawione w tabeli 8.

Tabela 8: Plany rozwojowe aplikacji demonstracyjnej

Moduł	Opis zmian
ComponentBuilder	Ogólne rozszerzenie i optymalizacja modułu **ComponentBuilder** dotyczyć będzie: • wsparcie dla cache'owania raz załadowanych komponentów, • 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, • przeniesienie kodu widoku stron domenowych do łatwiejszego w zarządzaniu oraz utrzymaniu kodu ExtJS, • wprowadzenie automatycznego mechanizmu generującego przekierowania do stron obiektów domenowych, • 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.
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.
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.
Następna strona...	

Tabela 8 – kontynuacja...

Moduł	Opis zmian
Kod ogólny	Usunięcie pozostałości nadmiarowego kodu, który pozostał po uaktualnieniu aplikacji, do korzystania z najnowszej wersji (4.0.0) szkieletu aplikacji **Spring**.
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, w porównaniu do poprzednich wersji.
RBuilder	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.
Przewodniki	Ukończenie wszystkich wymaganych kreatorów służących zarówno do modyfikacji, jak i tworzenia nowych obiektów domenowych.
Terminarz spotkań	Ukończenie prac nad terminarzem spotkań. Obecna funkcjonalność pozwala na tworzenie obiektów, które następnie można przeglądać z użyciem kalendarza (podobnego do używanego w programie Outlook).
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

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 znaczenia, będzie możliwa do szybkiego wprowadzenia do gotowego rozwiązania. Pozostałe braki wynikają głównie z problematyki rozwiązania aspektów technicznych 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, takie jak: prawidłowy dobór bibliotek, koncepcja przygotowania projektu przed jego 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*. dostęp: 2014-06-24. URL: http://www.aivosto.com/project/help/pm-oo-mood.html.

[2] *c3p0 - JDBC3 Connection And Statement Pooling*. dostęp: 2014-06-24. URL: http://www.mchange.com/projects/c3p0/.

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

[4] *Java Persistence Query Language*. dostęp: 2014-06-24. URL: http://docs.oracle.com/javaee/7/tutorial/doc/persistence-querylanguage.htm.

[5] Chidamber Kemerer. *Chidamber Kemerer object-oriented metrics suite*. dostęp: 2014-06-24. 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] Oracle. *Overview of the JMS API*. dostęp: 2014-06-24. URL: http://docs.oracle.com/javaee/7/tutorial/doc/jms-concepts001.htm.

[8] Oracle. *Overview of the JMX Technology*. dostęp: 2014-06-24. URL: http://docs.oracle.com/javase/tutorial/jmx/overview/index.html.

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

[10] Wydawnictwo Naukowe PWN SA. *Słownik Języka Polskiego*. dostęp: 2014-06-24. URL: http://sjp.pwn.pl/.

[11] Adam Shaw. *Full Calendar*. dostęp: 2014-06-24. URL: http://arshaw.com/fullcalendar/.

[12] Pivotal Software. *Spring Framework Reference Documentation*. dostęp: 2014-06-24. 2014. URL: http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/.

[13] Pivotal Software. *Spring Web Flow Reference Documentation*. dostęp: 2014-06-24. 2014. URL: http://docs.spring.io/spring-webflow/docs/2.4.x/reference/htmlsingle/.

[14] Terracotta. *Ehcache - Documentation*. dostęp: 2014-06-24. 2014. URL: http://ehcache.org/documentation/index.

Spis tabel

Spis rysunków

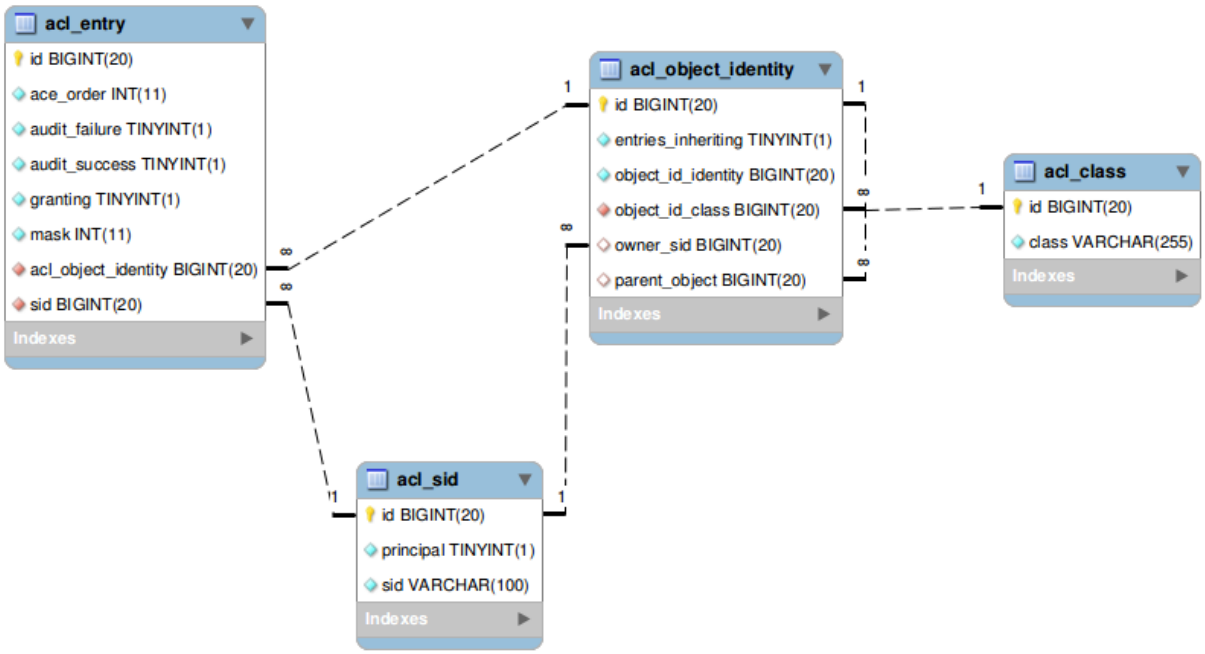
Kody źródłowe

A. Diagramy UML, schemat bazy danych

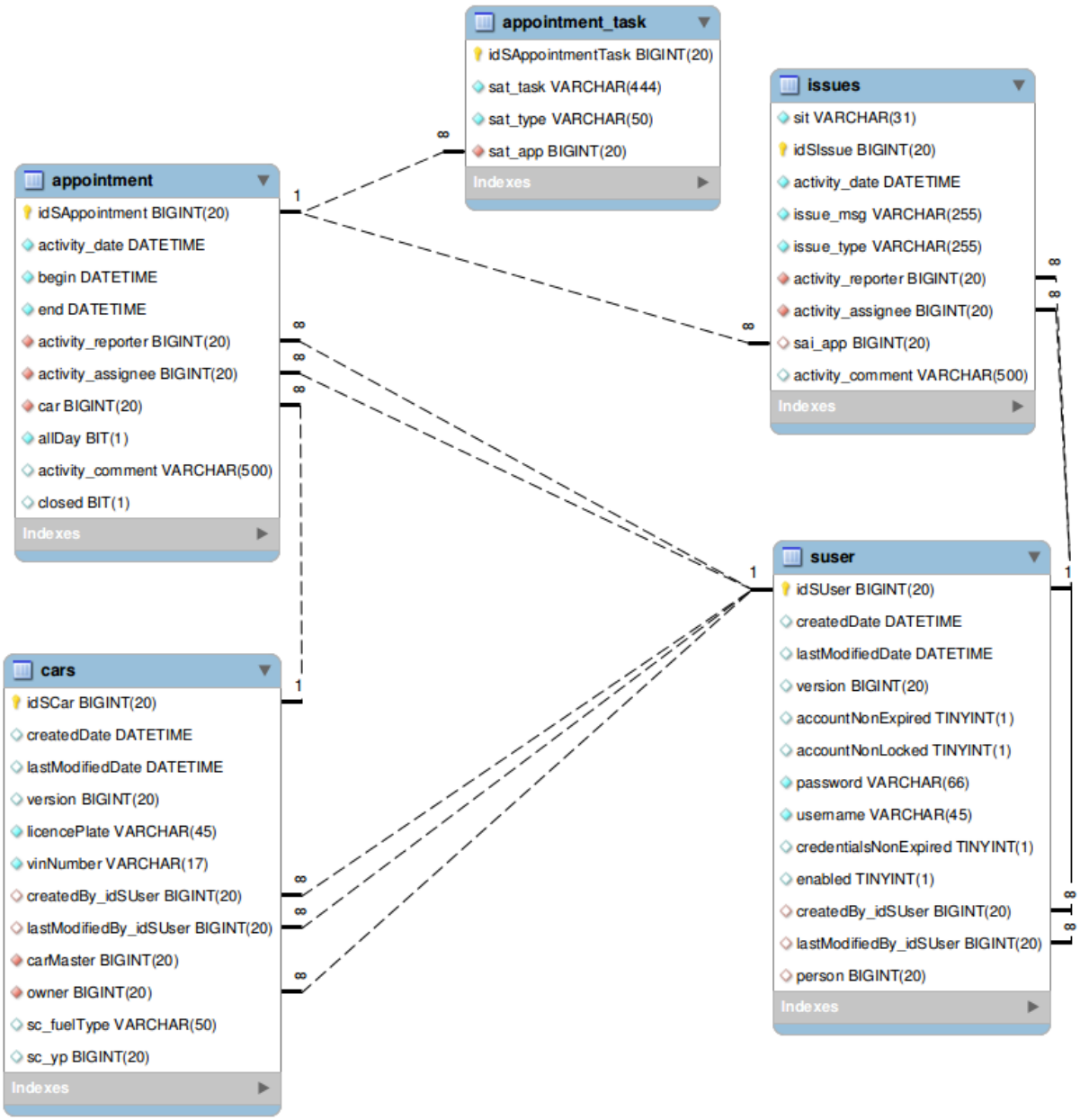
A.1. Schematy warstwy danych

A.1.1. Schemat tabel

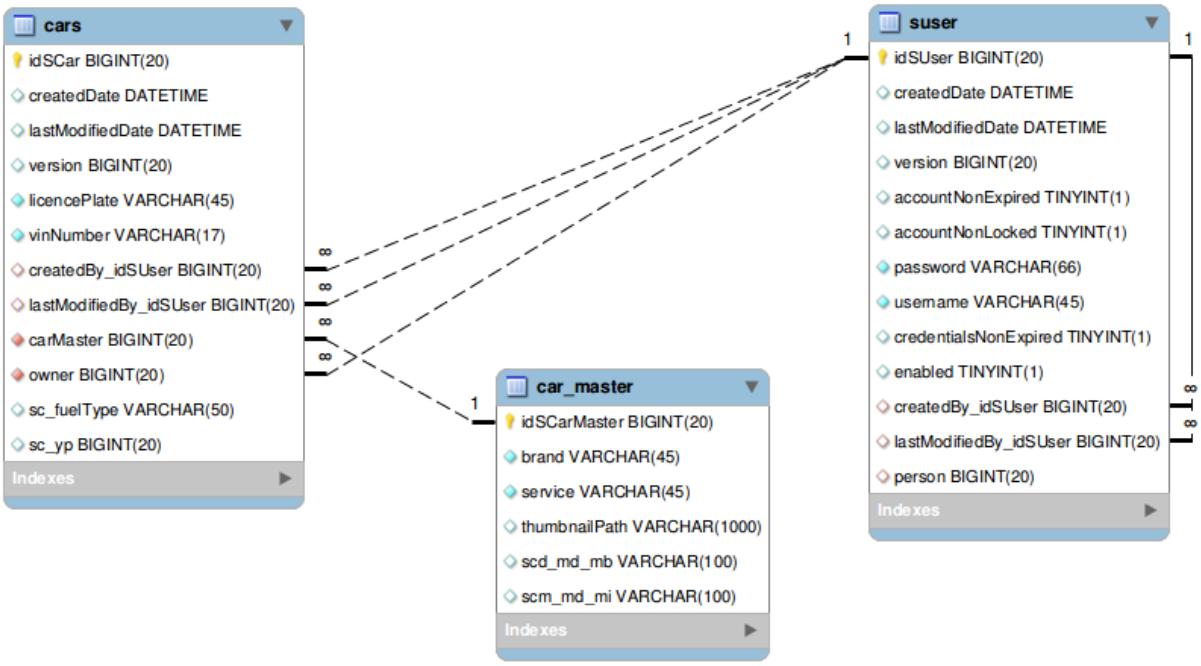
Diagramy poniżej przedstawiają schemat bazy danych. Dla większej czytelności zostały one podzielone na mniejsze rysunki. Każdy z nich przedstawia logicznie powiązane ze sobą tabele.



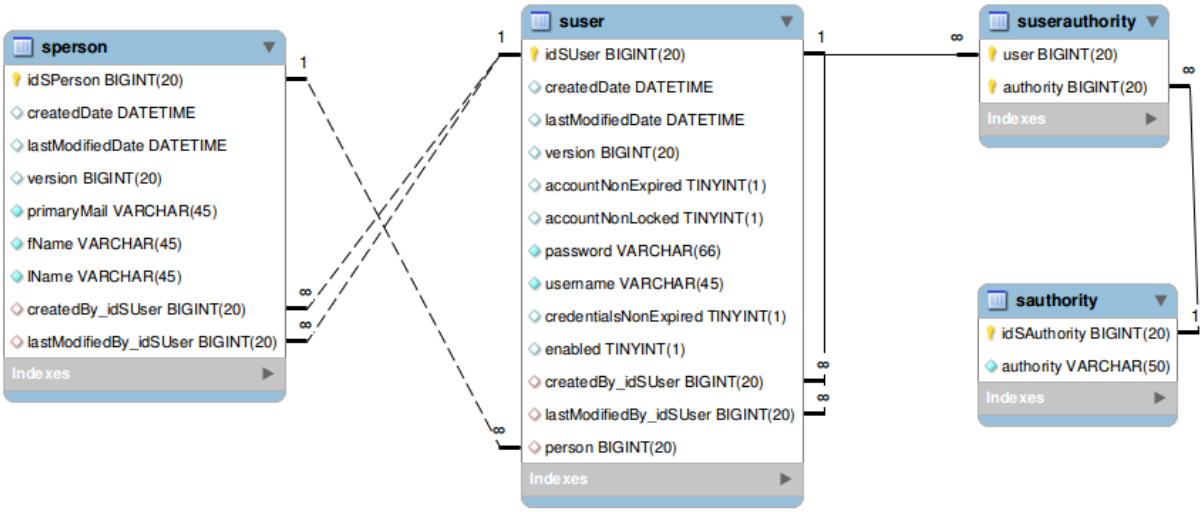
Rysunek 20: *Tabele grupy acl*



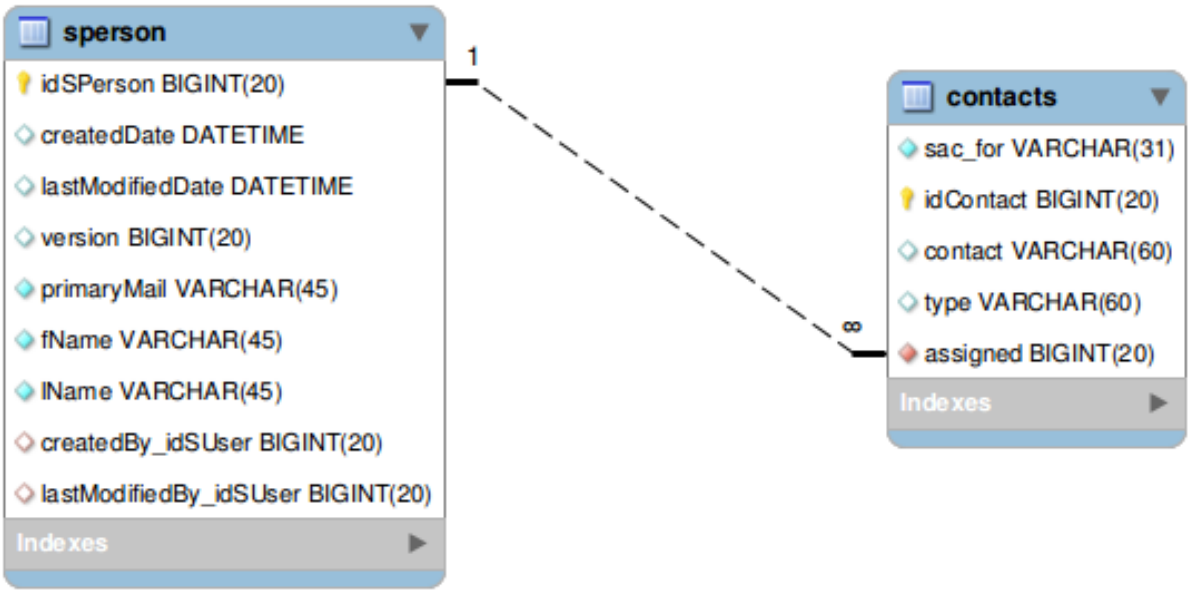
Rysunek 21: *Tabele grupy appointment*



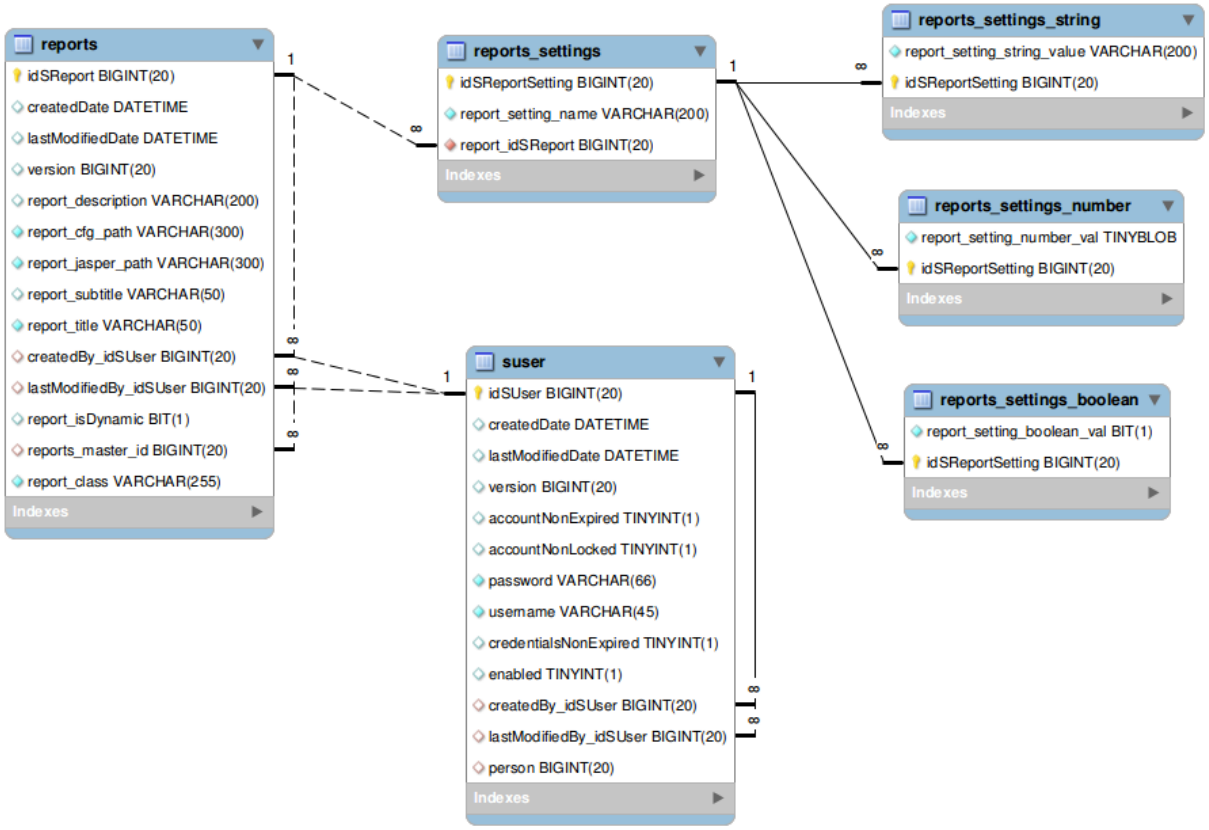
Rysunek 22: *Tabele grupy car*



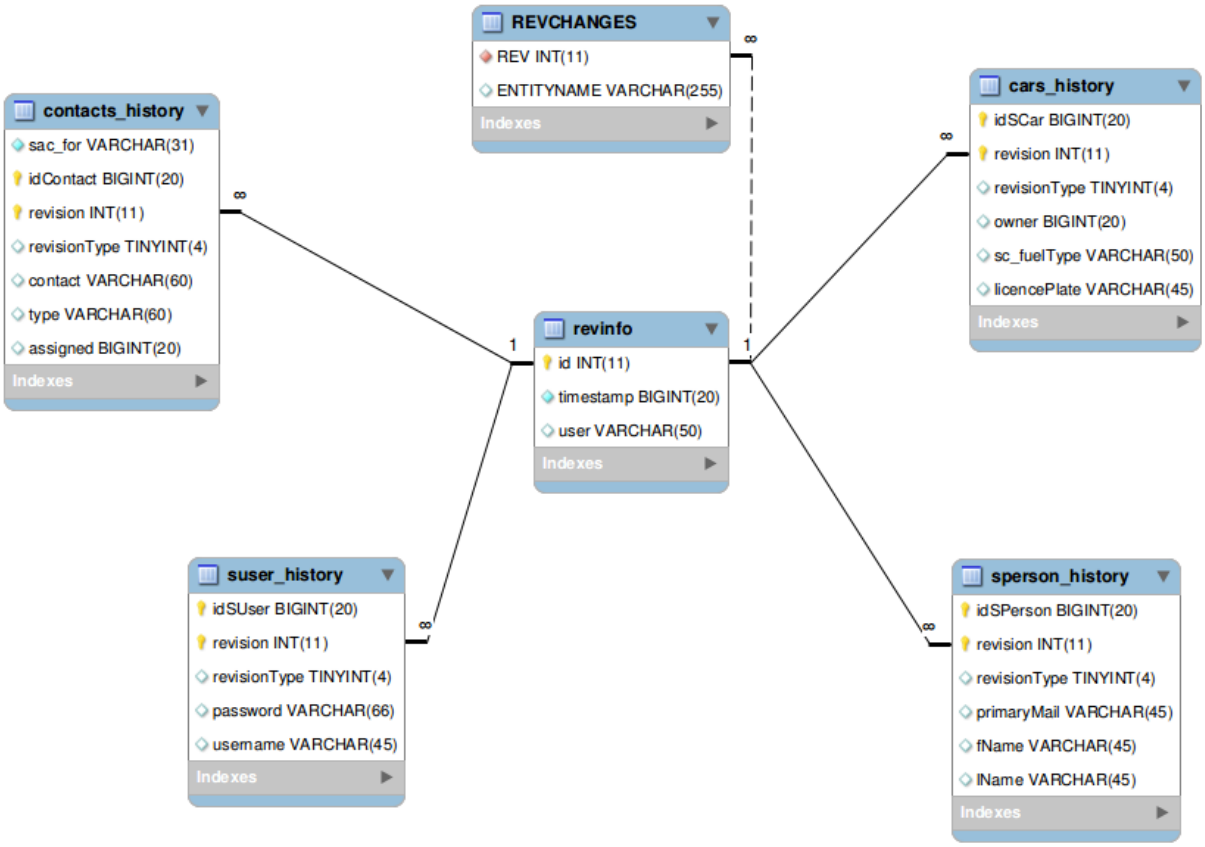
Rysunek 23: *Tabele grupy user*



Rysunek 24: *Tabele grupy person*



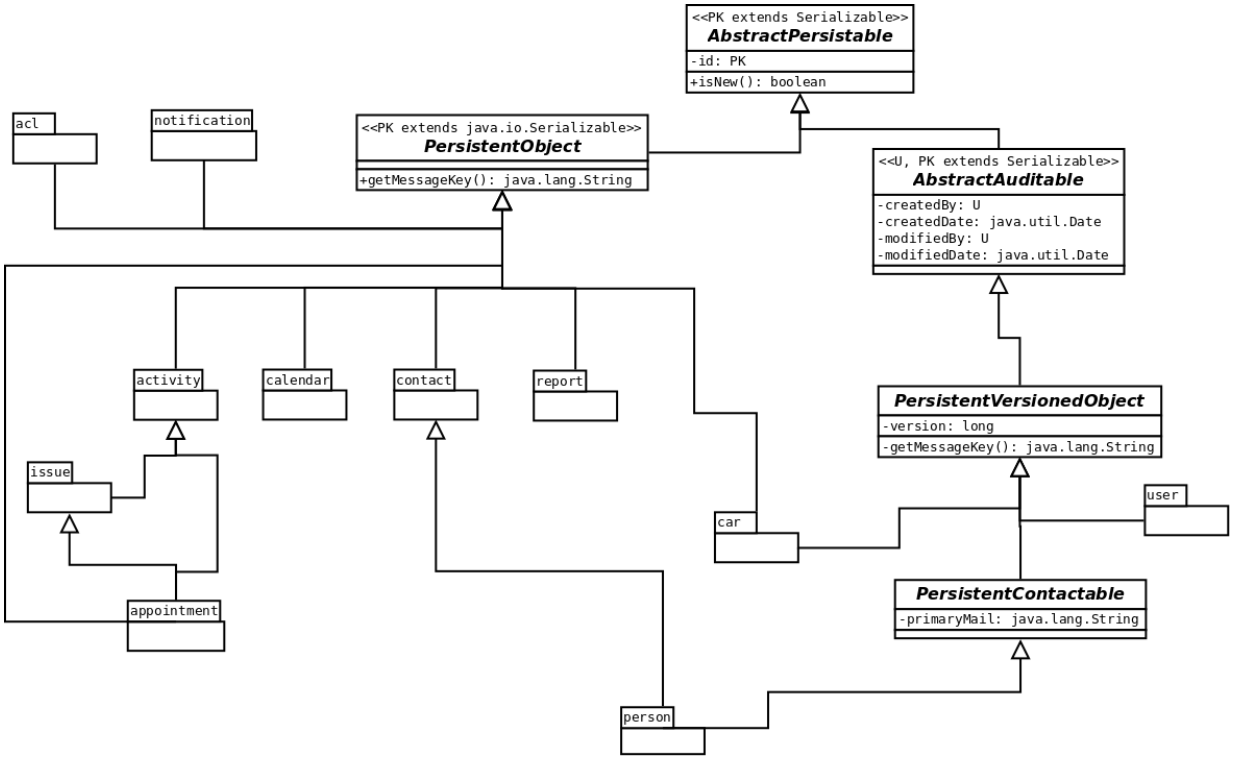
Rysunek 25: *Tabele grupy report*



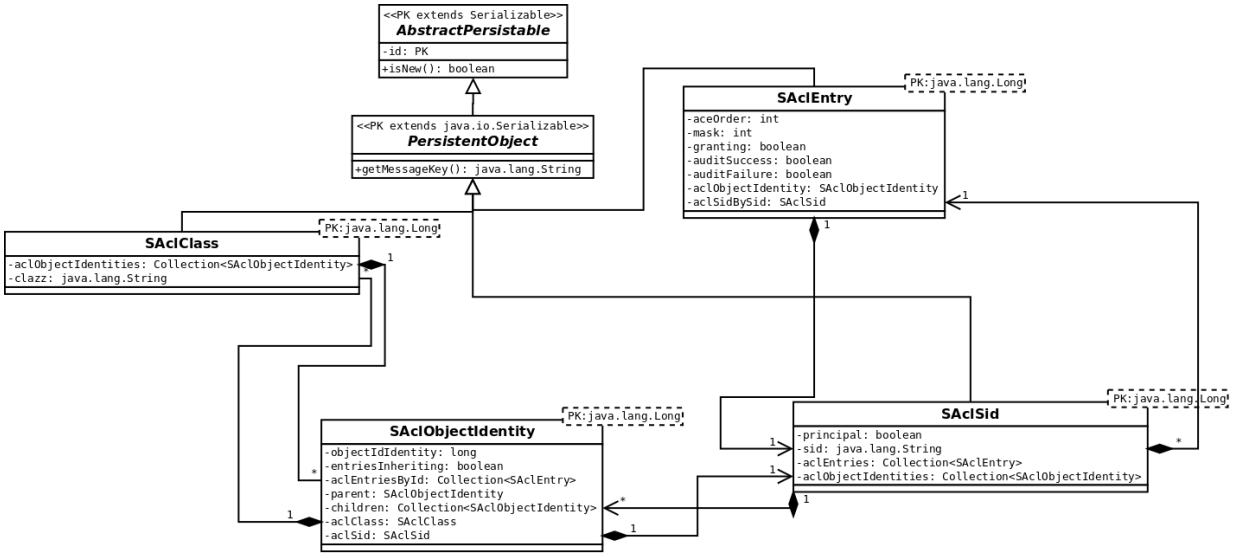
Rysunek 26: *Tabele grupy history*

A.1.2. Schemat obiektowy

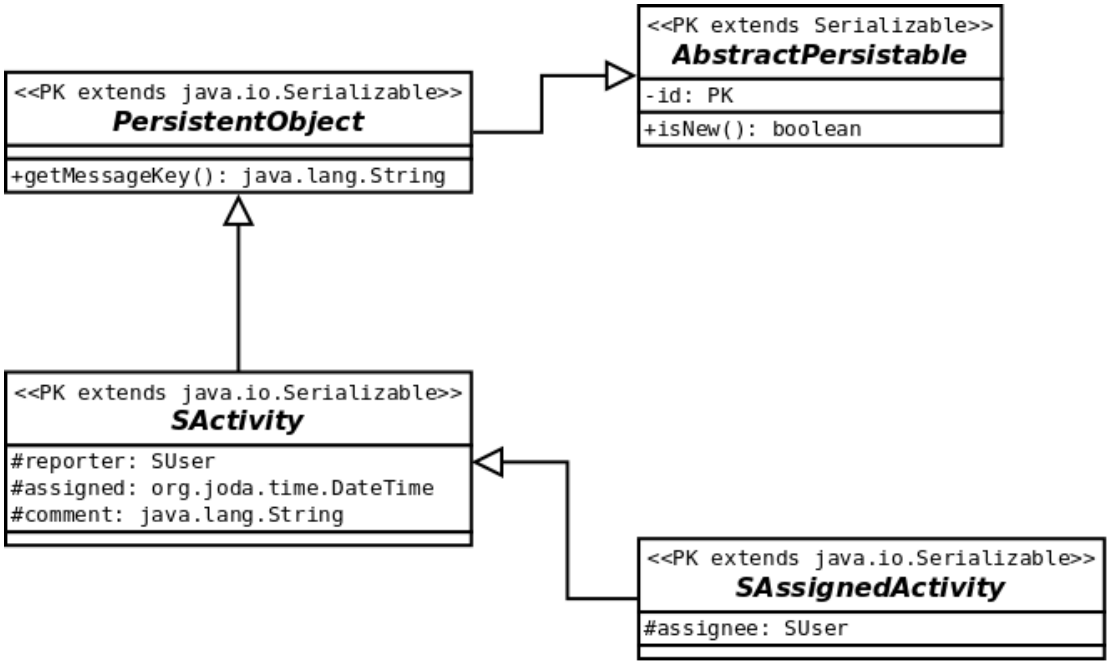
Poniższe rysunki przedstawiają diagramy klas poszczególnych paczek opisujących model danych. Na wszystkich diagramach pominięte zostały metody dostępowe (gettery oraz settery).



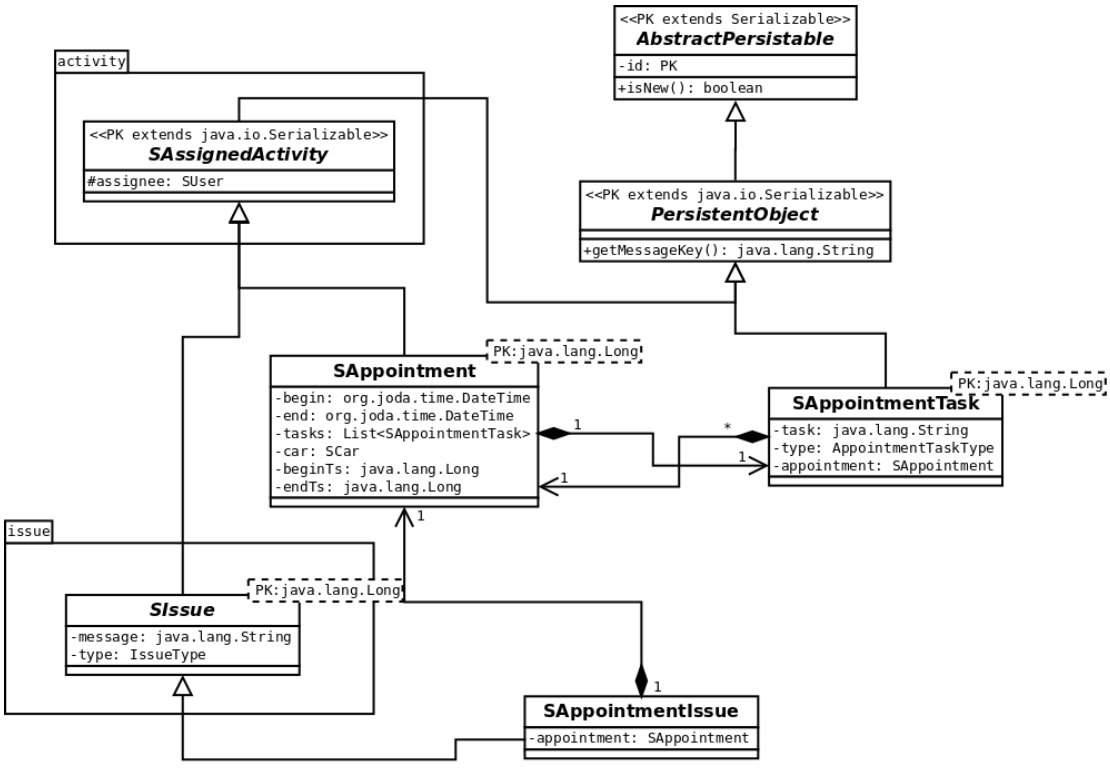
Rysunek 27: *Obiektowy model danych*



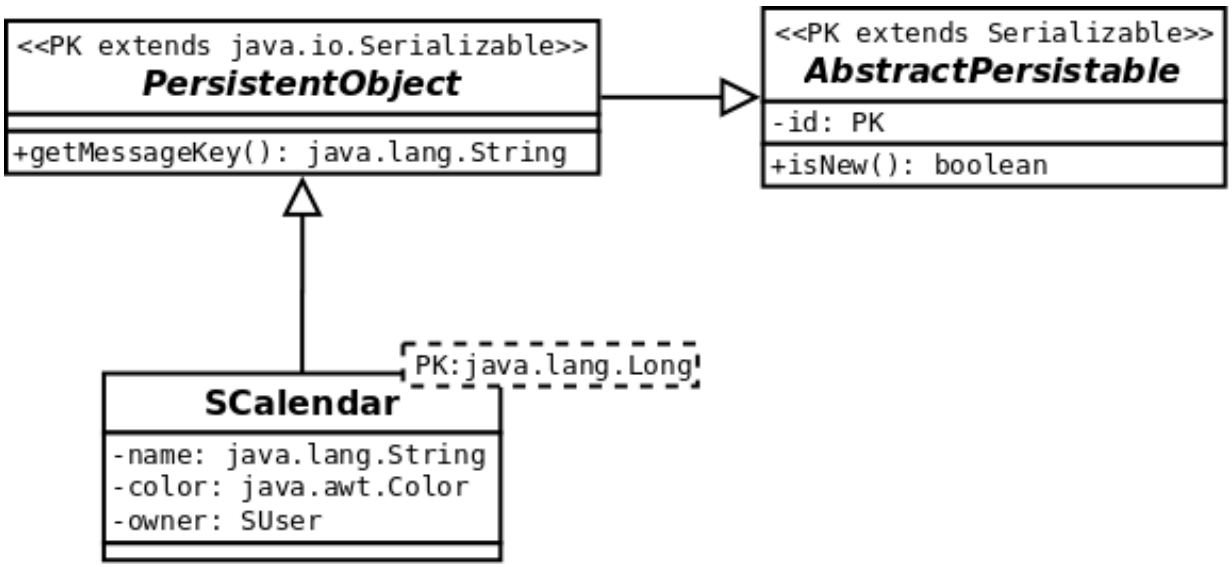
Rysunek 28: *Obiektowy model danych - Paczka* ***acl***



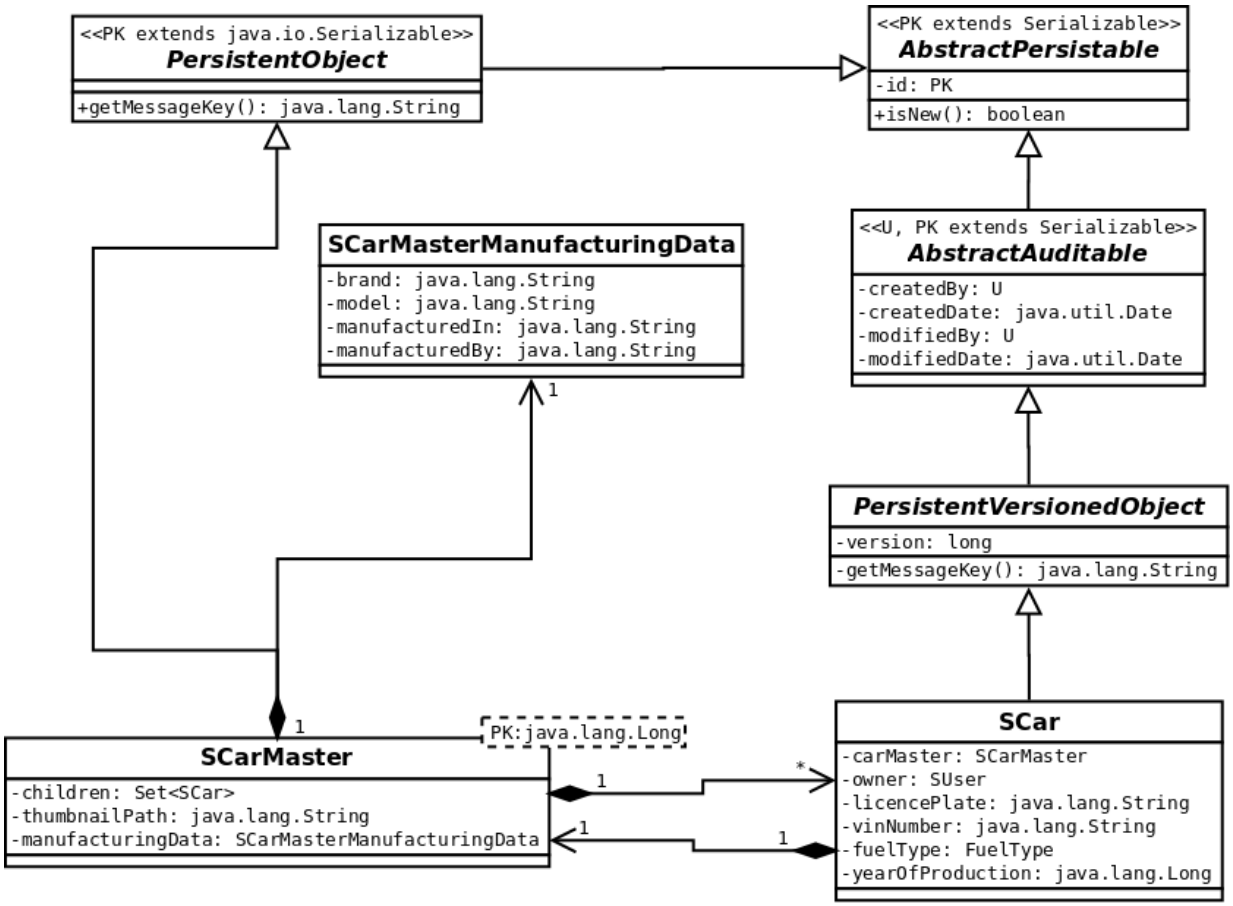
Rysunek 29: *Obiektowy model danych - Paczka* ***activity***



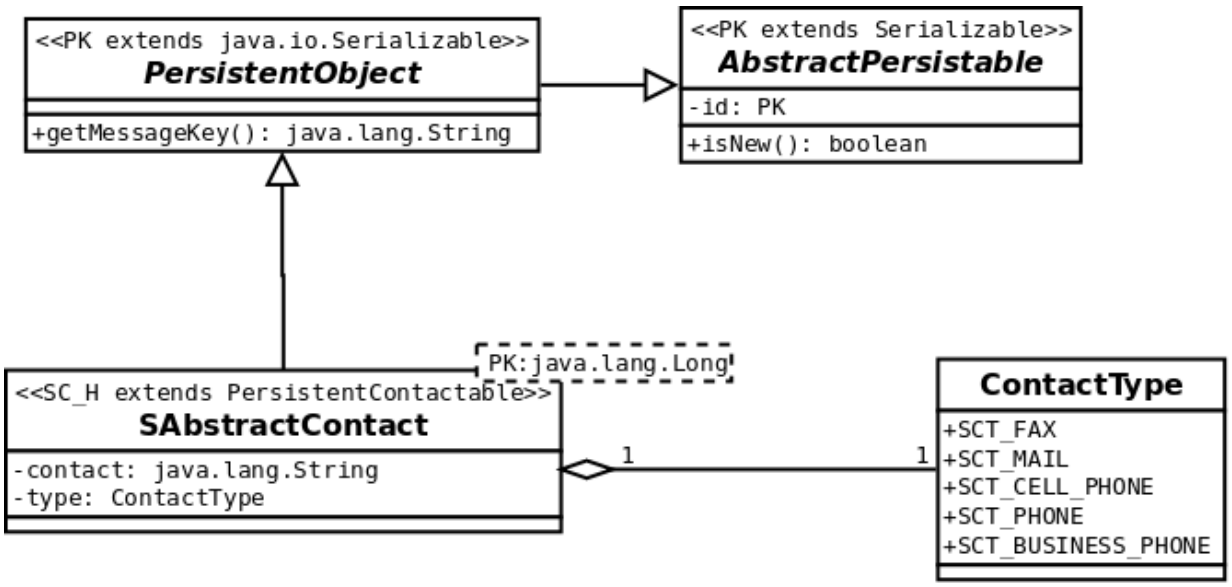
Rysunek 30: *Obiektowy model danych - Paczka* ***appointment***



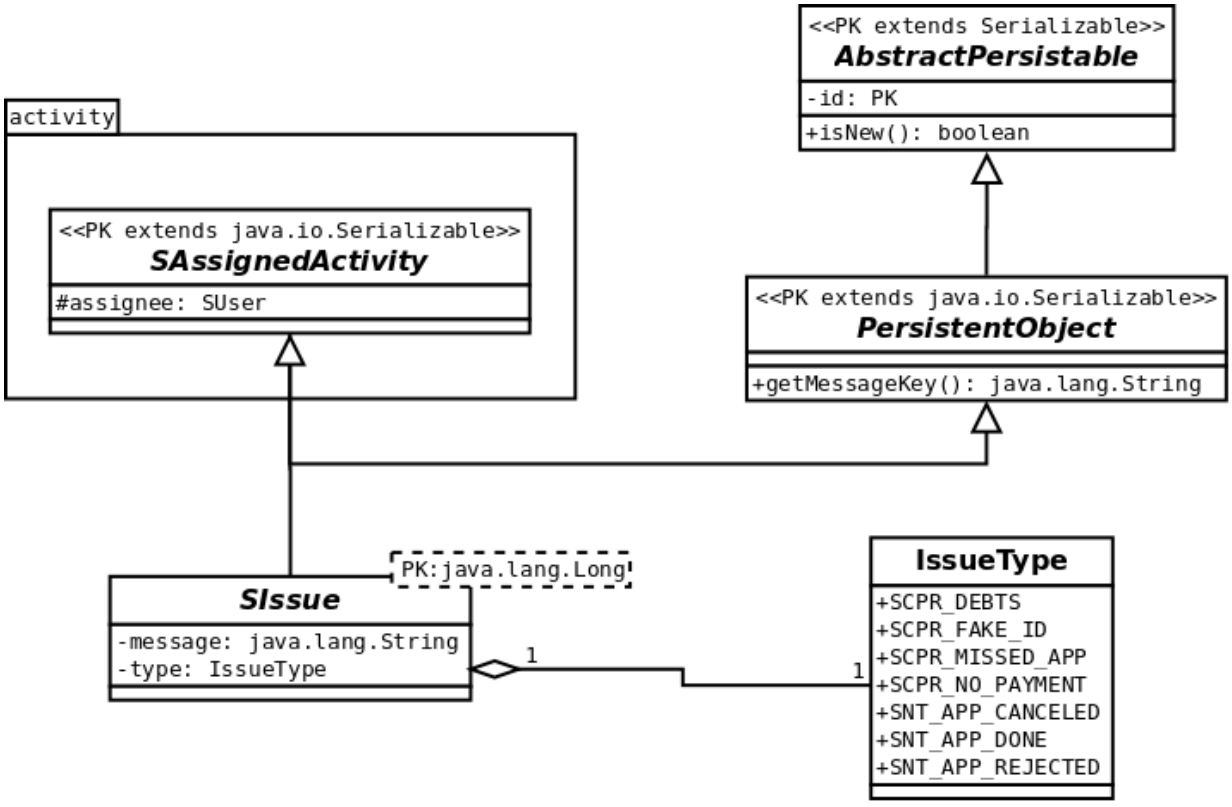
Rysunek 31: *Obiektowy model danych - Paczka* ***calendar***



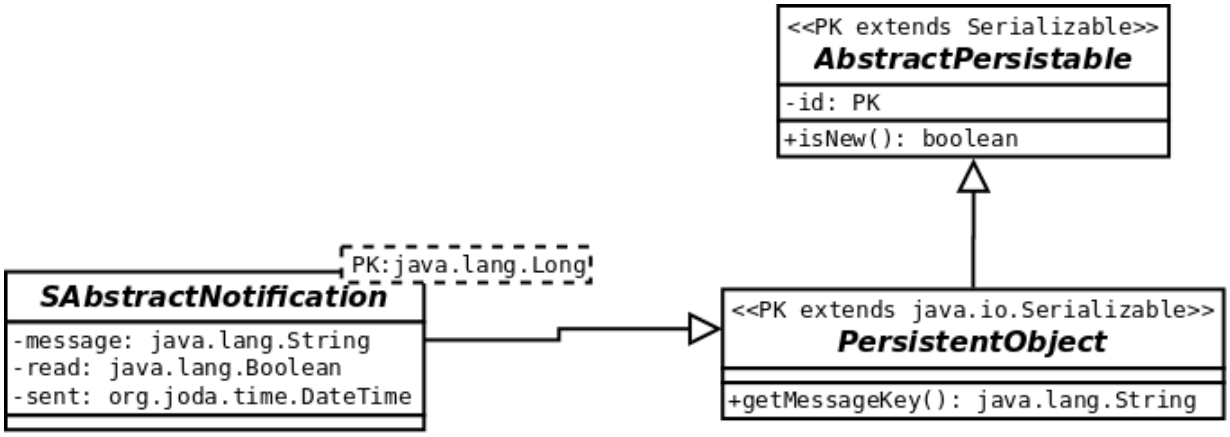
Rysunek 32: *Obiektowy model danych - Paczka* ***car***



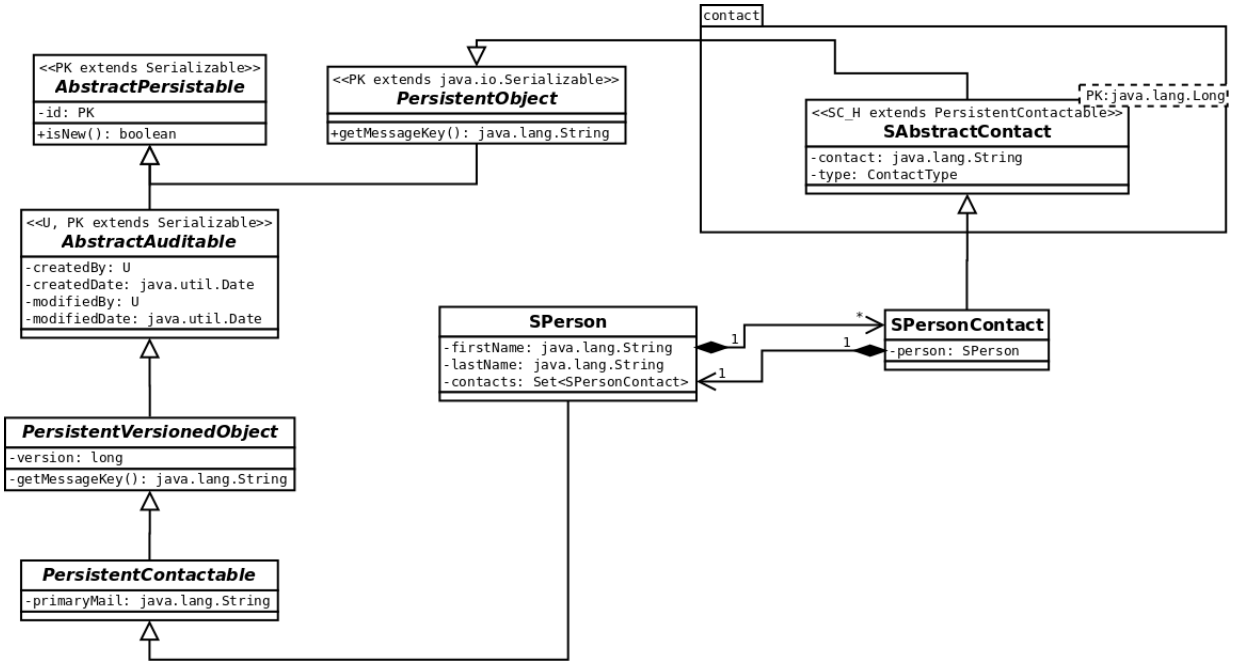
Rysunek 33: *Obiektowy model danych - Paczka* ***contact***



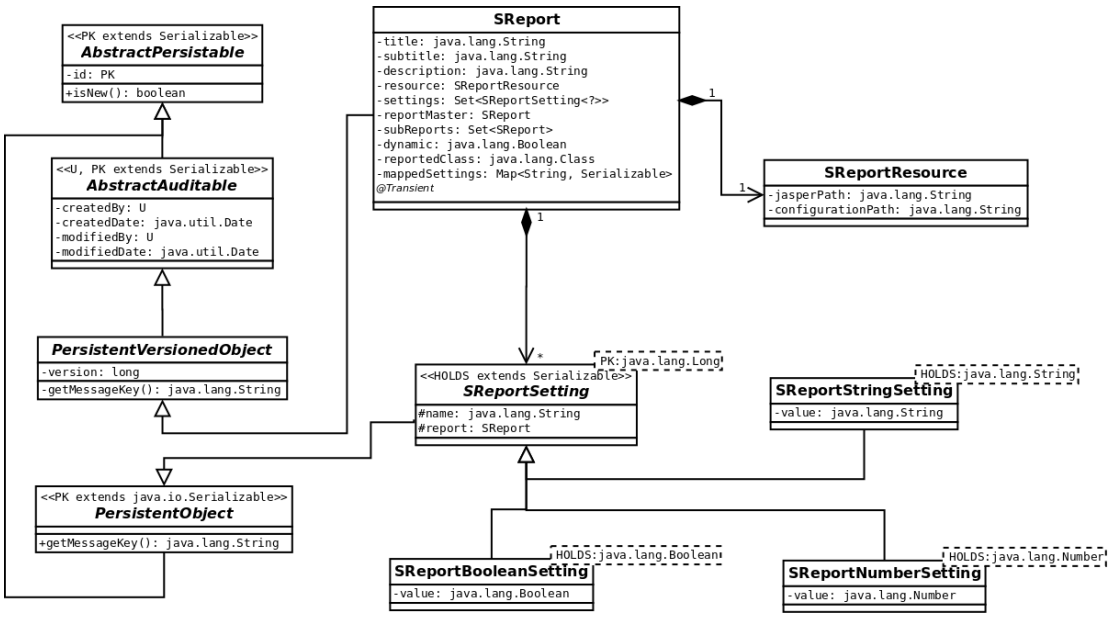
Rysunek 34: *Obiektowy model danych - Paczka* ***issue***



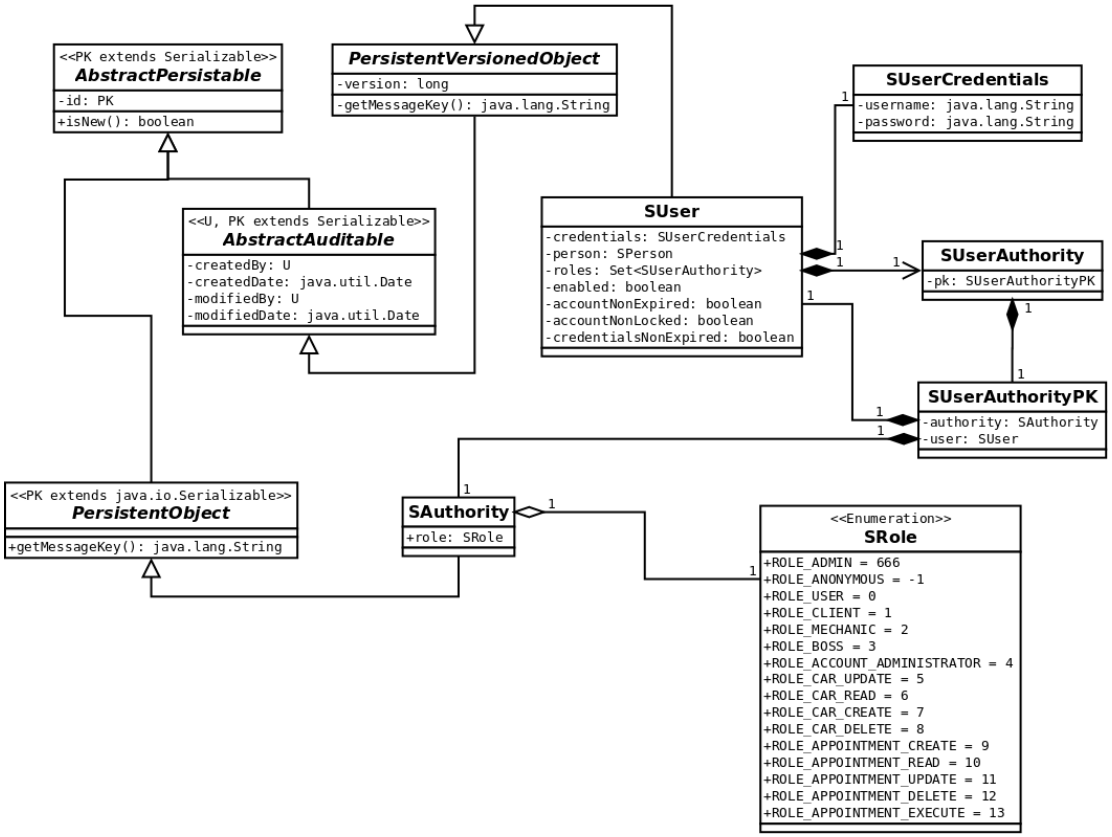
Rysunek 35: *Obiektowy model danych - Paczka* ***notifications***



Rysunek 36: *Obiektowy model danych - Paczka* ***person***

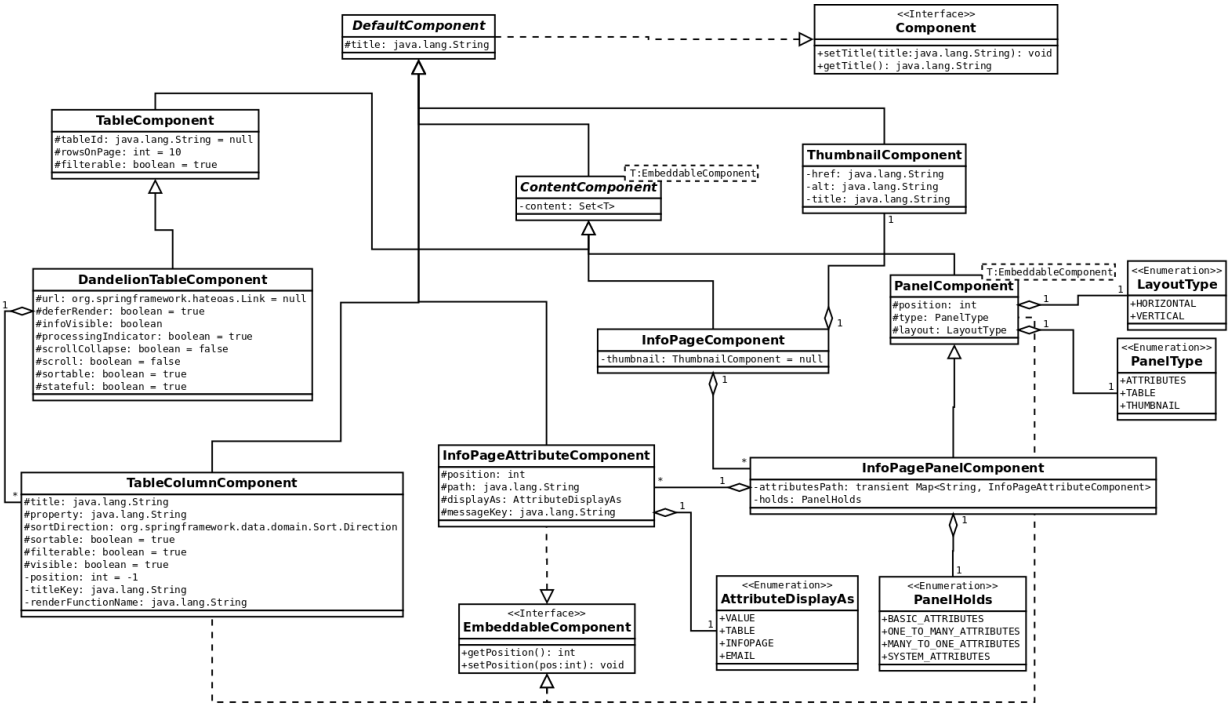


Rysunek 37: *Obiektowy model danych - Paczka* ***report***



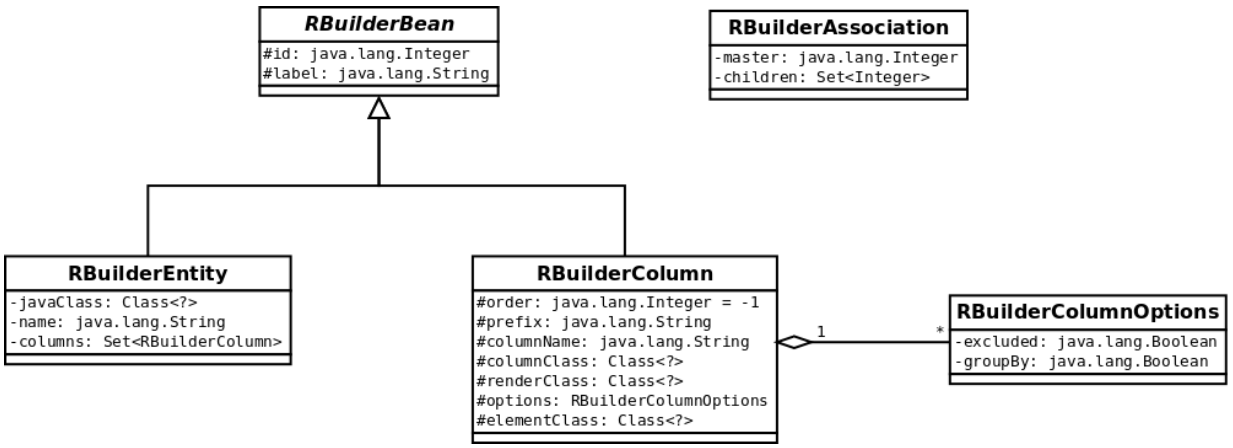
Rysunek 38: *Obiektowy model danych - Paczka* ***user***

A.2. Model danych komponentów InfoPage oraz TableComponent



Rysunek 39: *Diagram modelu danych komponentów **InfoPage** oraz **TableBuilder***

A.3. Model danych komponentu RBuilder



Rysunek 40: *Diagram UML modelu danych **RBuilder***

B. 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.

B.1. Liczba linii kodu aplikacji

Tabela 9: Liczba linii kodu według języka programowania

Język	**LOC**	**LOC (SC)**	**LOC (C)**	**LOC (EL)**
Java	29555	16628	9136	3791
JS	1044	?[1]	?	?
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

B.1.1. Java

Tabela 9 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**

[1] Nie znalezione narzędzia do analizy kodu danego języka

B.2. Strukturalne metryki kodu

Tabela 10: 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

B.3. 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 11: 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 współczynników 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.

B.4. 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 12: 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.