Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Framework e-commerce

Przemysław Magiera
Nr indeksu: 229773

Praca inżynierska napisana
pod kierunkiem
Wojciecha Macyny

Politechnika
Wrocławska
Wrocław 2017

Spis treści

Wstęp

Celem niniejszej pracy dyplomowej jest zaprojektowanie i zaimplementowanie frameworku służącego do usprawnienia implementacji systemów e-commerce. Istnieje wiele rozwiązań tego typu, jednak bardzo duża część z nich nie oferuje satysfakcjonujących parametrów wydajnościowych, przez co platformy oparte o takie frameworki są czesto bardzo powolne, do tego rozwijane od wielu lat wykorzystują stare rozwiązania i technologie. Prowadzi to często do niepotrzebnego skalowania pionowego aplikacji, czyli zwiększania mocy obliczeniowej. Proces ten wiąże się z bardzo dużymi kosztami, szczególnie w przypadku platform handlowych typu B2B. Zdecydowanie lepszym wyjściem okazuje się w takim przypadku jeden z dzisiejszych trendów budowania aplikacji, czyli skalowanie poziome, polegające na dzieleniu aplikacji według zastosowania poszczególnych komponentów, umieszczając niezależne jej części na serwerach dziedzinowych (architektura mikroserwisowa). Taka architektura pozwala na skalowanie tylko konkretnych, najbardziej narażonych na wzmożony ruch, elementów infrastruktury, co skutkuje bardzo dużą oszczędnością w stosunku do aplikacji monolitycznych. Budowa mikroserwisowa nie jest jednak cudownym środkiem na każdego rodzaju problemy dzisiejszych aplikacji internetowych, wiąże się z nim bowiem wiele problemów, jak chociażby integracja i synchronizacja między komponentami lub konieczność administracji bardzo złożonego środowiska. Właśnie ze względu na to ostatnie widzimy dziś tak wiele ofert pracy na stanowisko DevOps (development and operations). Wydaje się dlatego, że architektura monolityczna z oddzieleniem warstwy katalogowej (tej najbardziej obciążonej) jest optymalnym rozwiązaniem problemu implementacji sklepów internetowych.
Często dewelopment aplikacji idzie w parze z presją czasu, przez co zapomina się o jakości kodu i rozwiązaniach, które poprawiłyby wydajność i ograniczyły konieczność skalowania. Z zamkniętymi oczami podąża się za schematami i szablonami, aby dostarczyć rozwiązanie jak najszybciej, a nie jak najlepiej. Dlatego właśnie założeniem projektu w ramach pracy jest zaprojektowanie i implementacja frameworku e-commerce spełniającego następujące założenia funkcjonalne:

Praca składa się z czterech rozdziałów. W rozdziale pierwszym omówiono strukturę przedsiębiorstwa . . . , scharakteryzowano grupy użytkowników oraz przedstawiono procedury związane z obiegiem dokumentów. Szczegółowo opisano mechanizmy . . . . Przedstawiono uwarunkowania prawne udostępniania informacji . . . , w ramach . . . . W rozdziale drugim przedstawiono szczegółowy projekt systemy w notacji UML. Wykorzystano diagramy . . . . Zawarto w niej w pseudokod algorytmów generowania oraz omówiono jego właściwo- ści. . .

Słowniczek

indeksacja – proces synchronizacji pomiędzy relacyjną bazą danych, a szybką bazą noSQL, używany w systemie do encji najbardziej narzuconych na duże wykorzystanie. W skrócie:

> *By adding content to an **index**, we make it searchable by Solr.*[**?**]

facet – jest to atrybut danej encji, zazwyczaj wyszukiwalny. Używa się ich do implementacji filtrów używanych podczas wyszukiwania. Z dokumentacji Solra:

> *Searchers are presented with the indexed terms, along with numerical counts of how many matching documents were found were each term. Faceting makes it easy for users to explore search results, narrowing in on exactly the results they are looking for.* [**?**]

reflection – nieskopoziomowe udogodnienie w języku Java, pozwalające na operacje i wyświetlanie właściwości klasy Javowej.

> *Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect"upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.* [7]

Analiza problemu

W tym rozdziale została przedstawiona analiza zagadnienia. Nakreślono problemy i omówiono proponowane przez system ich rozwiązania. Omówiono założenia funkcjonalne i niefunkcjonalne samego systemu jak i jego podsystemów, przedstawiono podobne rozwiązania informatyczne. Zawarty jest również słowniczek, potrzebny do pełnego zrozumienia zagadnienia.

Charakterystyka problemu

Aplikacje webowe, a w szczególności systemy e-commerce, są często wolne. Pisane przez niedoświadczonych deweloperów pod presją czasu nie zawsze wychodzą tak jak powinny. A jak powinny być napisane? W pierwszej kolejności muszą być dobrze przemyślane, a co za tym idzie ich architektura powinna być przygotowana na rozszerzenia i modyfikacje. Powiedzmy, że mamy sklep internetowy, na bazie którego chcielibyśmy stworzyć podobne rozwiązanie. Często jest to niemożliwe ze względu na budowę i obsługę komponentów. Dobrym przykładem na to jest indeksowanie produktów do mechanizmu wyszukującego.

Przykład 2.1 *W systemie istnieje klasa Product z zadeklarowanymi polami biznesowymi, powiedzmy że klient chce nowe pole myCustomField, oczywiście ma być ono wyszukiwalne. Rozszerzamy więc klasę Produkt do MyProdukt i dodajemy pole myCustomField. Mechanizm indeksacji nie ma możliwości wyciągnąć z Produktu pola dotyczącego klasy MyProdukt, ponieważ zaprzeczałoby to zasadom polimorfizmu.*

To tylko konkretny przykład, ale warto zwrócić uwage, że dotyczy on nie tylko produktów, a i również jakichkolwiek encji, którymi chcemy zarządzać w systemie. To pierwszy problem rozwiązań e-commerce, które nie są oparte o elastyczne frameworki.

Kolejną rzeczą idąca za słabą architekturą są tak zwane bottlenecki[1], które blokują szybkie działanie aplikacji. Przykład z życia:

Przykład 2.2 *Nasza platforma jest oparta o framework stworzony parę lat temu, do tego taki, który nie jest open-source, ma architekturę monolityczną. Nasz sklep zyskał na popularności i coraz częściej zdarzają nam się awarię związane z ograniczoną wydajnością, najszybszą i naturalną reakcją jest dokupienie nowego serwera do obsługi większej ilości klientów, jednak wiąże się to z bardzo dużymi kosztami.*

W tym przykładzie warto zwrócić uwagę na zamknięty charakter frameworku. Brak licencji open-source często skutkuje to tym, że projekt jest skazany na zamknięty cykl rozwoju (o ile w ogóle jest dalej rozwijany), ewentualne błędy nie mogą być naprawione *od ręki*. Raz napisana platforma komercyjna nie będzie po 5 latach wcale aktualna. Inaczej sprawy się mają w przypadku open-source'wych frameworków, gdzie możliwość zmiany technologi, a aktualizacje są dostępne praktycznie zawsze. To dzięki zasadzie *inversion of control*, kiedy to programista decyduje się na oddanie kontroli nad

[1] bottleneck - wspólny punkt dla aplikacji, przez muszą przejść użytkowicy korzystając z różnych funkcjonalności

częścią swojej funkcjonalności w ręce używanego przez siebie frameworku, aktualizacja do nowszych wersji używanych technologii jest zwykle bezbolesna i opiera się głównie na zmianie wersji w pliku konfiguracyjnym zależności projektowe.

Następny problem związany ze środowiskiem e-commerce to brak dobrych wyszukiwarek produktów na stronach. Często zdarza się, że wyszukiwarki przeszukują relacyjne bazy danych, zamiast korzystać z płaskich struktur jakie oferują nam rozwiązania typu noSQL. Do tego nie obsługują facetów [2], co sprawia trudności z wyszukaniem produktu i dopasowaniem go pod klienta, a przecież to jest główny biznes. Jasne i wiadome jest, że istnieją sklepy z dobrymi wyszukiwarkami, do tego mogące pochwalić się wysokim miernikiem TPS[3]. Jednakże są to rozwiązania bardzo drogie i niedostępne dla małych przedsiębiorstw, z drugiej strony implementowanie takich rzeczy na własną rękę jest również bardzo drogie, a do tego skomplikowane. W tym momencie z pomocą przychodzą właśnie frameworki, nie wszystkie jednak maja pełna obsługe mechanizmu indeksującego, a szczególnie nie w darmowych wersjach.

Istotną sprawą w opisywanych systemach jest archiwizacja produktu, bądź jej zupełny brak. Problem został opisany na poniższym przykładzie:

Przykład 2.3 *Klient zakupił produkt A o atrybutach (a, b, c, d) w sierpniu. W październiku manager sklepu zmienił atrybuty produktu A na (e, f, g), co wpłynęło również na cenę. W listopadzie klient zdecydował się na reklamację produktu. Obsługa sklepu otrzymała zgłoszenie, ma id produktu, jednak niewiele to pomoże, gdyż atrybuty i cena uległy zmianie.*

Ostatnią, nie mniej jednak ważną rzeczą, która warta jest opisania to sposób, w jaki charakteryzuje się produkty[4] w sklepach internetowych. Produkty należy opisywać w konsekwentny i najbardziej jednoznaczny sposób, jednak nie da się tego osiągnąć bazując tylko na polach ewentualnej klasy Produkt, jest to wybitnie nieelastyczne, każda zmiana wymaga trwałej ingerencji w kod i ponowny release.

Powyższe problemy niektórych systemów e-commerce można sprowadzić do następujących stwierdzeń:

- architektura nieprzygotowana pod nagłe modyfikacje i rozszerzenia
- najczęściej używane punkty aplikacji nie są odseparowane od reszty
- brak dobrych wyszukiwarek i mechanizmów filtrujących
- niekonkterny i mało abstrakcyjny sposób opisu encji biznesowych
- użycie zamkniętych technologi, nie korzystanie z open source, brak zastosowania *inversion of control*
- brak archiwizacji produktów

Proponowane rozwiązania - wymagania funkcjonalne

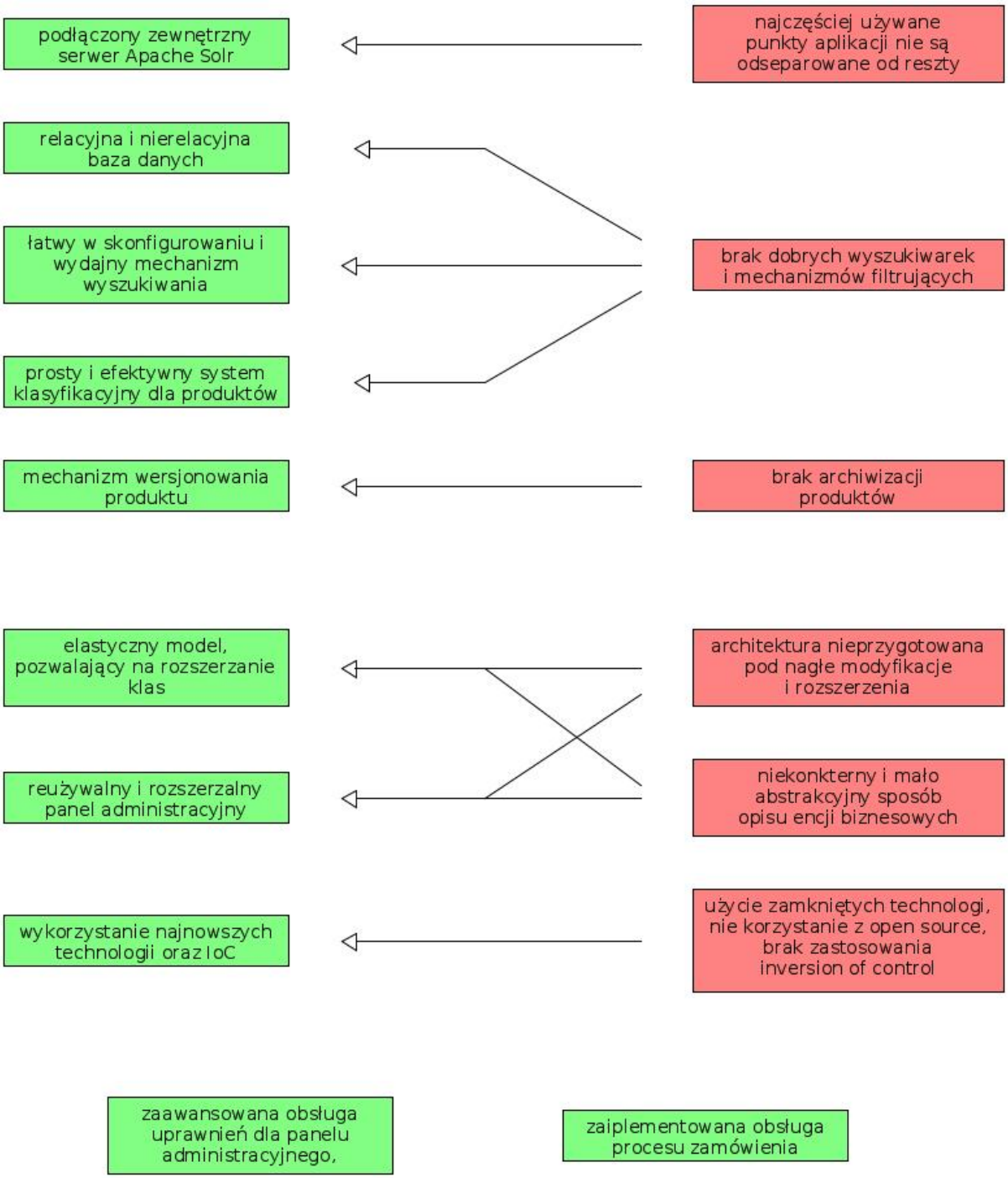
Jasno zdefiniowane problemy same nasuwają pewne rozwiązania, dlatego określono listę następujących wymagań funkcjonalnych:

[2] wyjaśnienie terminu dostępne w sekcji **Słowniczek**
[3] *TPS* – transaction per second, ilość pełnych requestów wraz z odpowiedzią, jaką może obsłużyć serwer na sekundę.
[4] lub cokolwiek innego podlegającego klasyfikacji

- wykorzystanie najnowszych technologii oraz IoC
- podłączony zewnętrzny serwer Apache Solr, służący do bardzo szybkiej obsługi zapytań związanych z katalogiem produktowym,
- prosty i efektywny system klasyfikacyjny dla produktów,
- reużywalny i rozszerzalny panel administracyjny
- zaawansowana obsługą uprawnień dla panelu administracyjnego,
- elastyczny model, pozwalający na rozszerzanie klas bez konieczności ingerowania w strukture systemu
- łatwy w skonfigurowaniu i wydajny mechanizm wyszukiwania,
- relacyjna i nierelacyjna baza danych,
- zaiplementowana obsługa procesu zamówienia,
- inicjalizer projektów, pozwalający szybko stworzyć przykładowe rozwiązanie e-commerce.
- mechanizm wersjonowania produktu

W podrozdziale **Charakterystyka problemu** nakreślono najważniejsze problemy sklepów internetowych i frameworków e-commerce. Rysunek 2.1 ilustruje pokrycie znalezionych problemów wymaganiami funkcjonalnymi opisywanego frameworka.

Z rysunku 2.1 wynika, że *bez pokrycia* zostają dwa wymagania, nie oznacza to jednak, że są to niepotrzebne funkcjonalności. Są jednak powszechnie spotykane w frameworkach e-commerce, można nazwać je standardem. Jednak co mniej oczywiste system uprawnień został oparty na strukturze drzewiastej, uprawnienia są dziedziczone pomiędzy użytkownikami finalnego systemu opartego o framework.



Rysunek 2.1: Połączenie wymagań funkcjonalnych ze znalezionymi problemami.

Projekt systemu

W tym rozdziale przedstawiony zostanie dokładny projekt systemu, jest on podzielony na kilka podrozdziałów. **Grupy użytkowników** oraz **Przypadki użycia** opisują szczegółowo, do czego i przez kogo będzie mogła zostać uzyta implementowana platforma. Sekcje **Diagramy aktywności** i **Diagramy sekwencji** pokazują, w jaki sposób udało się osiągnąć bardziej skomplikowane cele określone za pomocą przypadków użycia, niektóre funkcjonalności w zależności od ich charakterystyki zostaną przedstawione na diagramach aktywności - te dotyczące programisty ze względu na bardziej skomplikowane algorytmy, oraz na diagramach sekwecji - te bardziej nastawione na funkcjonalności biznesowe.

W kolejnych sekcjach znajdą się **Diagramy klas** oraz **Projekt bazy danych**. W pierwszej opisane zostaną klasy, które musiały zostać stworzne do implementacji procesów zdefiniowanych w wcześniejszych podrozdziałach, odpowiednio w drugiej znajdą się schematy bazy danych podzielone ze względu na funkcjonalności.

Do implementacji frameworku oraz funkcjonalności wbudowanych użyto wzorca architektonicznego *model-view-controller*. Polega on na następującym podziale aplikacji: **model**, odpowiedzialny za reprezentacje problemu i logikę, **view** (widok) opisujący interfejs użytkownika oraz **controller**, który przyjmuje polecenia użytkownika i odpowiada na jego żądania za pomocą modelu i widoku. Składa się on z trzech wzorców projektowych:

- kompozyt – sposób tworzenia widoków, możliwość zagnieżdżania ich w sobie
- obserwator – model może zmienić stan, o czym musi zostać powiadomiony widok (aktualizacja informacji)
- strategia – wybranie strategii obsługi zapytania w kontrolerze na podstawie samego żądania

W projekcie użyto również wzorzec projektowy *inversion of control (odwórcenie sterowania)*. Egzemplarze większości encji i serwisów obsługuje Spring Framework, o który oparty jest projekt. Wspomniane *IoC* zostało zrealizowane poprzez wstrzykiwanie zależności.

Grupy użytkowników

We frameworku możemy zdefiniować 3 grupy użytkowników, którzy będą korzystać z jego udogodnień. **Programista** jest to użytkownik frameworka, który implementuje swój sklep z pomocą narzędzi dostarczonych przez opisywany system. **Administrator** to ktoś, zajmujący się backoffi-cową[1] obsługą sklepu - obsługa reklamacji. **Klient** jest to końcowy klient sklepu, który przegląda katalog i dokonuje zakupów.

Warto zazanaczyć w tym momencie, że tematem pracy jest zaimplementowanie frameworka, co wskazuje na to, że funkcjonalności będą skupione głównie na programiście, rola administratora i klienta oraz przypadki użycia z nimi powiązane są jedynie zmienną w implementowanym systemie.

[1] backoffice - w systemach e-commerce panel do obsługi i utrzymania sklepu

Oznacza to nic innego, że rola administratora (i rzeczy, które może zrobić), są uzależnione od konfiguracji i dodadków systemu. Głównym celem jest to, aby stworzyć narzędzie dzięki, któremu możliwości administratora i klienta są ograniczone jedynie *fantazją* programisty. Założenie to bardzo dobrze ilustruje następujący przykład:

Przykład 3.1 *Załóżmy, że implementujemy sklep internetowy, korzystając z opisywanego frameworka. Nasz pracodawca życzy sobie aby w sklepie pojawił się również blok z artykułami, którymi będzie można zarządzać w panelu administracyjnym. Normalnie proces implementacji takiej funkcjonalności wiązałby się z przygotowaniem modelu w bazie danych i pełnej jego obsługi, również ze strony front-endowej. Z użyciem frameworka proces można skrócić do zaimplementowania modelu i paru prostych koment aby wygenerować mechanizm do manipulacji stworzonym modelem - w przykładzie blogiem wraz z artykułami.*

Przypadki użycia

Jak zostało zdefiniowane w poprzednim punkcie, w pracy przewidziano 3 grupy użytkowników. W zamyśle framework jest narzędziem dla programisty, jednak w systemie został zaimplementowany szereg rozwiązań gotowych do wykorzystania dla końcowych użytkowników, dlatego diagramy przypadków użycia zostały podzielone na trzy klasy:

- przypadki użycia Programisty
- przypadki użycia Użytkownika Administracyjnego potencjalnego serwisu e-commerce, opartego na opisywanym Frameworku
- przypadki użycia użytkownika końcowego, czyli Klienta

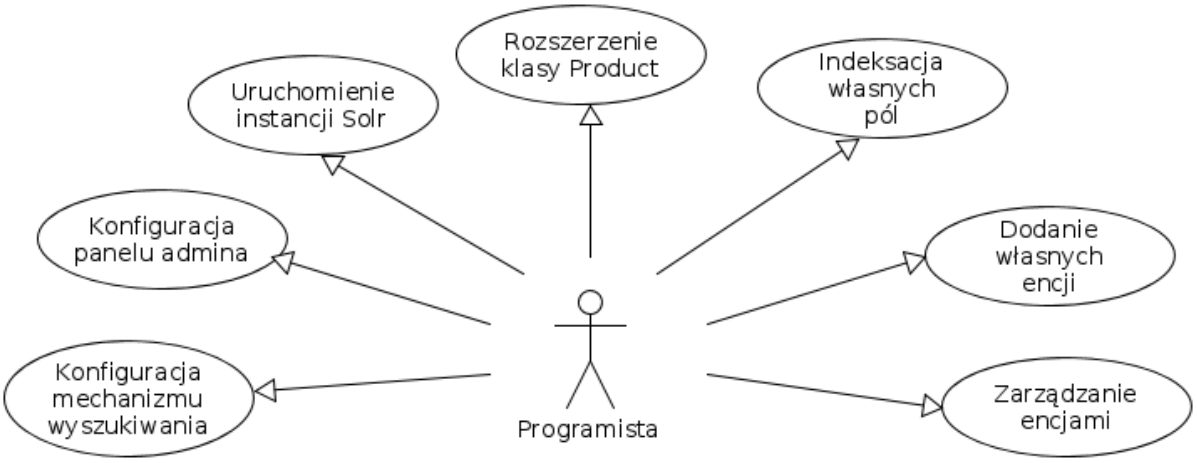
Na rysunku 3.1 zostały przedstawione najważniejsze przypadki użycia frameworku. Programista ma swobodny dostęp do rozszerzania encji, w szczególności klasy Produkt, która ma wyjatkowo strategiczne znaczenie w systemach e-commerce. Dodatkowo ma możliwość uczynienia niestandardowych pól wyszukiwalnymi przez klienta. Sytuacja została zobrazowana na poniższym przykładzie.

Przykład 3.2 *Załóżmy, że mamy niestandardowe pole proste (String) w encji klasyfikowanej przez twórców ewentualnego sklepu związanego z opisywanym frameworkiem jako finalna encja nadająca się do sprzedaży. Niech nazywa się* `MyProduct extends Product` *z polem* `myCustomField`*. Jedyne co w tej sytuacji musimy zrobić aby system mógł wyciągnąć wartość tego pola z encji (o której de facto nie wie) to wpisać do tabeli zawierającej indeksowane cechy produktu nazwę danego pola, system za pomocą refleksji*[2] *wyekstaktuje wartość pola z encji.*

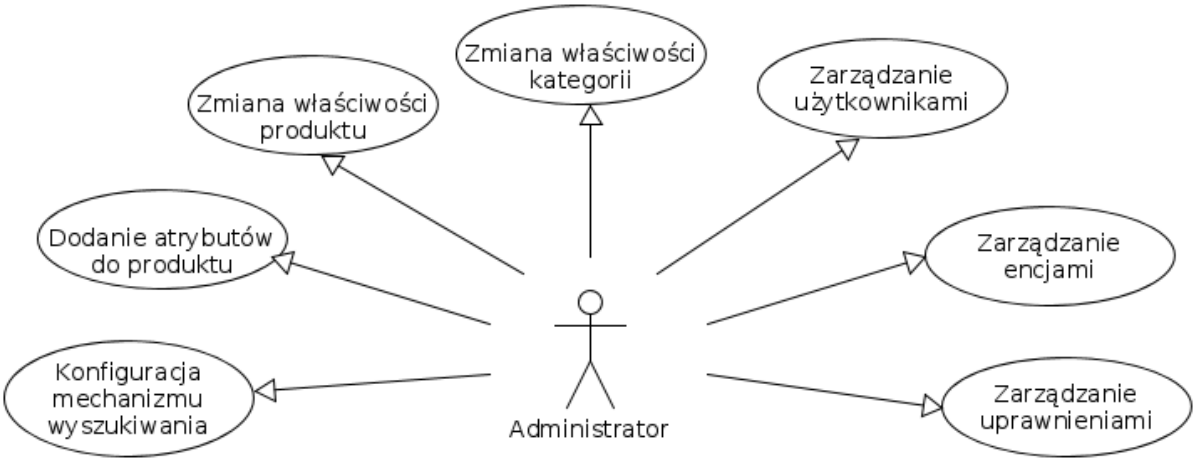
Dodane przez Programistę encje są obsługiwane przez framework, dodatkowo po dodaniu specjalnej adnotacji[3] *nad nią, może być zarządzana w uniwersalnym panelu administracyjnym. Osoba zajmująca się implementacją sklepu opartego o opisywaną platformę może uruchomić dowolną (skończoną) ilość instancji Apache Solr, czyli bazy danych noSQL, służącej do obsługiwania zapytań związanych z katalogiem produktowym (skalowalność pionowa tylko tej części aplikacji, która tego potrzebuje). W odniesieniu do przypadku użycia Nadpisanie mechanizmu wyszukiwania z rysunku 3.1 serwisy są oparte na interfejsach, zapewniając Programiście możliwość nadpisania jego logiki zgodnie z zasadami polimorfizmu.*

[2] *refleksja* (eng. reflection) – udogodnienie w języku Java, pozwalające na wyświetlenie i manipulacje właściwościami klasy. Więcej w sekcji **słowniczek**.

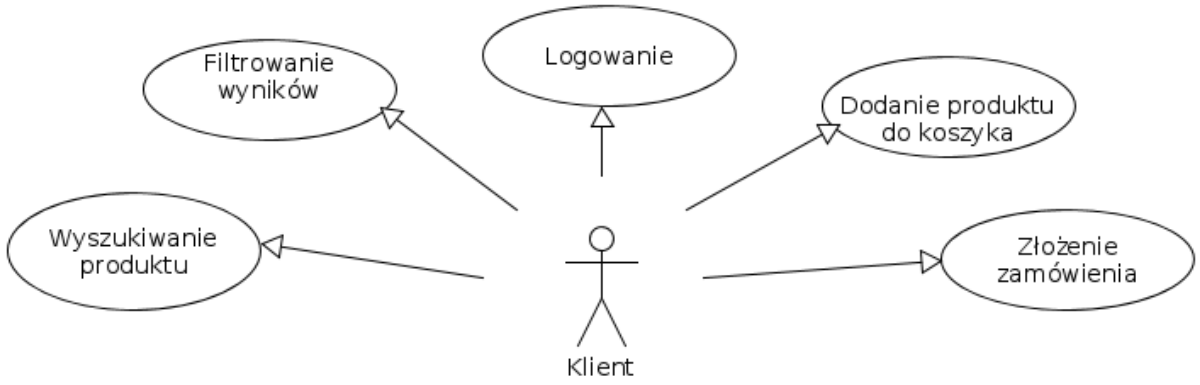
[3] adnotacja – używane w języku Java od wersji 1.7, najczęściej służą do określania dodatkowych właściwości pól bądź klas



Rysunek 3.1: Diagram przypadków użycia związany z Programistą.



Rysunek 3.2: Diagram przypadków użycia związany z Administratorem ewentualniego systemu.



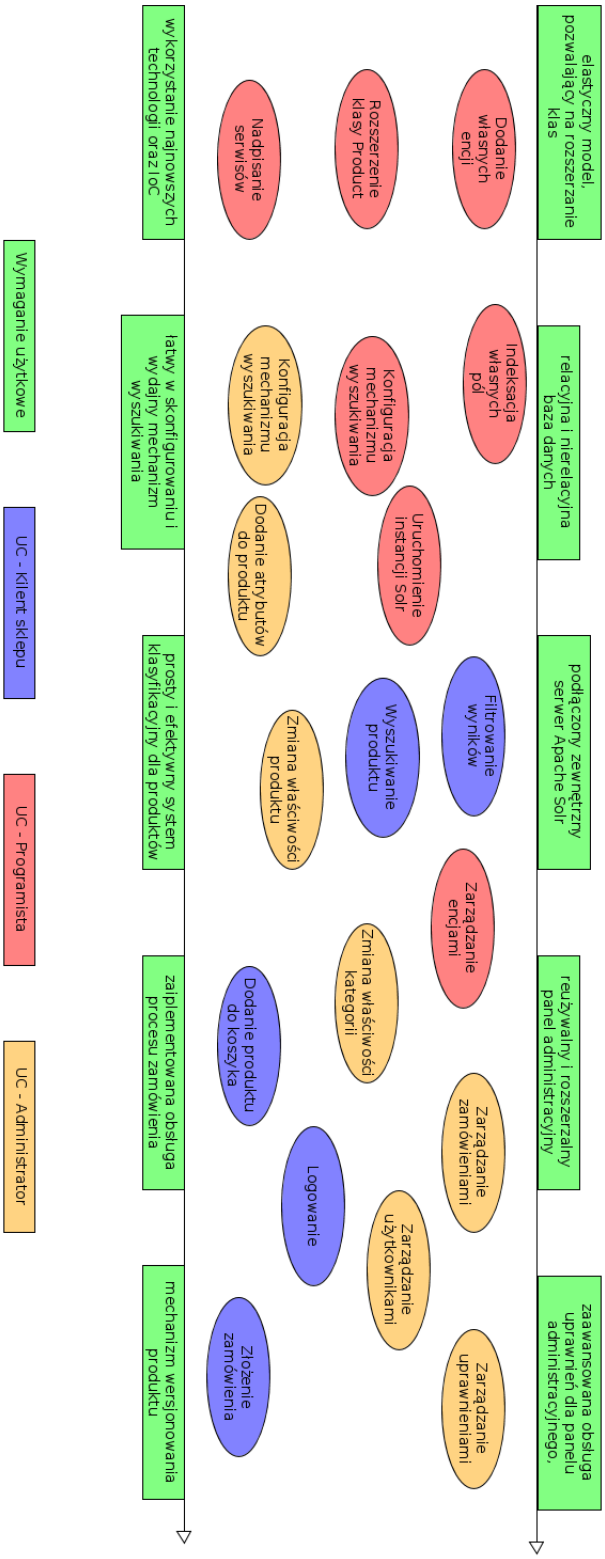
Rysunek 3.3: Diagram przypadków użycia związany z Klientem końcowym ewentualniego systemu.

Rysunek 3.2 przedstawia przypadki użycia z punktu widzenia Administratora potencjalnego systemu. Z punktu widzenia platformy jest to również klient, gdyż framework zakłada, że nie ma on wiedzy technicznej i nie potrafi programować. Podobnie jak programista, może konfigurować mechanizm wyszukiwania, jednak bardziej wysokopoziomowo, np. deklaracja używanych facetów. Panel administracyjny zakłada zarządanie najważniejszymi encjami: produkt, kategoria, użytkownik, zamówienie, uprawnienie i parę innych, zdefiniowanych dokładniej w podrozdziale **Diagramy bazy danych**.

Diagram na rysunku 3.3 dotyczy przypadków użycia elementów frameworku przez końcowego użytkownika. Są to klasyczne funkcjonalności tradycyjnego sklepu internetowego. *Wyszukanie produktu* zostało zaprojektowane, tak aby możliwy był również do zaimplementowania mechanizm podpowiedzi i podświetlania. Apache Solr udostępnia taką funkcjonalność. *Reklamacja* dotyczy opisanego w rozdziale **Analiza problemu** kłopotu z archwizacją produktu, został on rozwiązany prostym mechanizmem wersjonowania.

Diagramy typu *use-case* ściśle wiążą się z wymaganiami funkcjonalnymi systemu. Jest wiadome, że można je również sklasyfikować pod względem aktorów występujących w systemie, dlatego też powiązania przypadków użycia z wymaganiami funkcjonalnymi zostały umieszczone na diagramie z rysunku 3.4. Na diagram należy patrzeć poziomo, po zapoznaniu się z legendą. Wymagania są w nieprzypadkowej kolejności, są ustawione od lewej do prawej. Im bardziej na prawo, tym wymaganie jest bardziej biznesowe, im bardziej na lewo – dotyczy rdzeniowych elementów platformy. Warto zauważyć zależność, że im dalej patrzymy na diagram, tym więcej niebieskich i żółtych *use case'ów* – tych zarezerowowanych dla Administracji i Klientów rozwiązania e-commerce. Natomiast im bardziej na lewo tym więcej czerwonego, czyli przypadków przemyślanych dla Programisty.

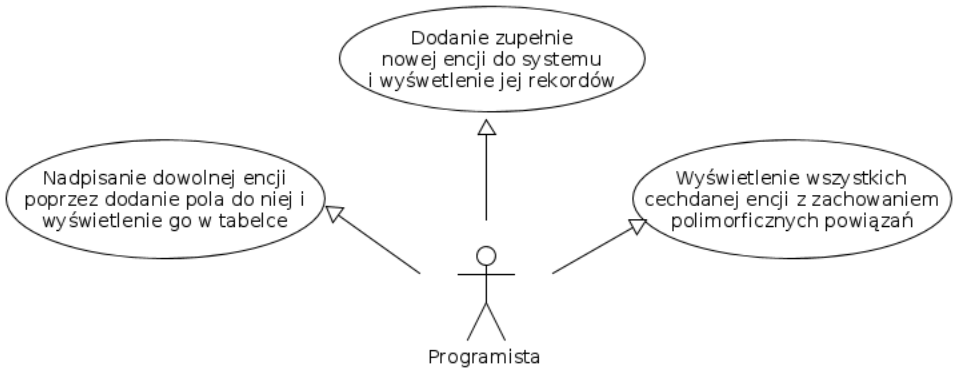
Jak zostało wspomniane wcześniej w sekcji **Grupy użytkowników i założenia**, framework jest narzędziem głównie dla programisty, to on zdecyduje co ma się znajdować w finalnym systemie, dlatego w niniejszym podrozdziale zostaną rozwinięte przypadki użycia dla programisty związane z obsługą i oprogramowywaniem dynamicznych elementów platformy. Jest to odpowiedź na najtrudniejsze z wymagań, czyli **elastyczny model, pozwalający na rozszerzenie klas** oraz **reużywalny i rozszerzalny panel administracyjny**.



Rysunek 3.4: Diagram przypadków użycia związany z wymaganiami funkcjonalnymi

Dynamiczna tabela encyjna

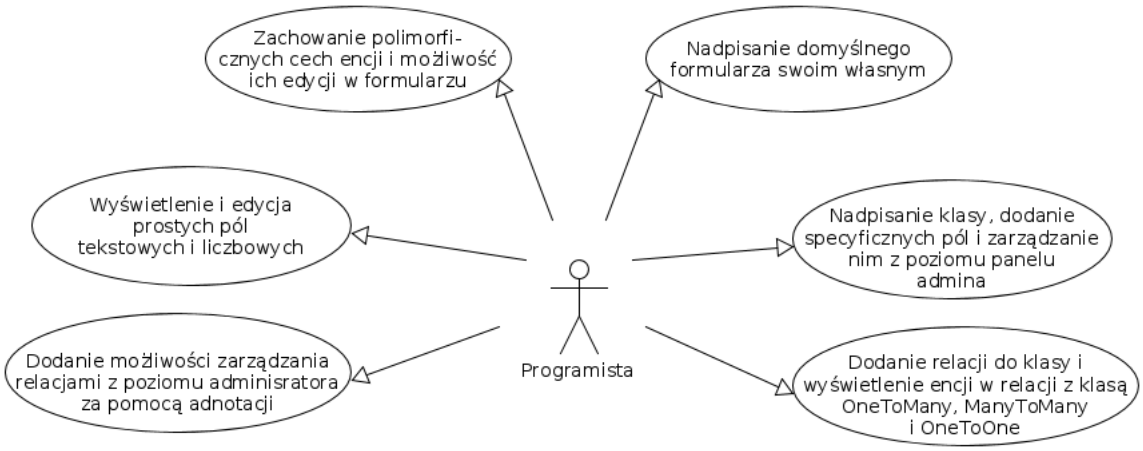
Dynamiczna tabela encyjna jest autorskim rozwiązaniem, służącym do wylistowywania dowolnych encji związanych z systemem w panelu administracyjnym, wiążą się z nim przpadki użycia z rysunku 3.5



Rysunek 3.5: Diagram przypadków użycia związany z dynamiczą tabelą encyjną

Dynamiczny formularz encyjny

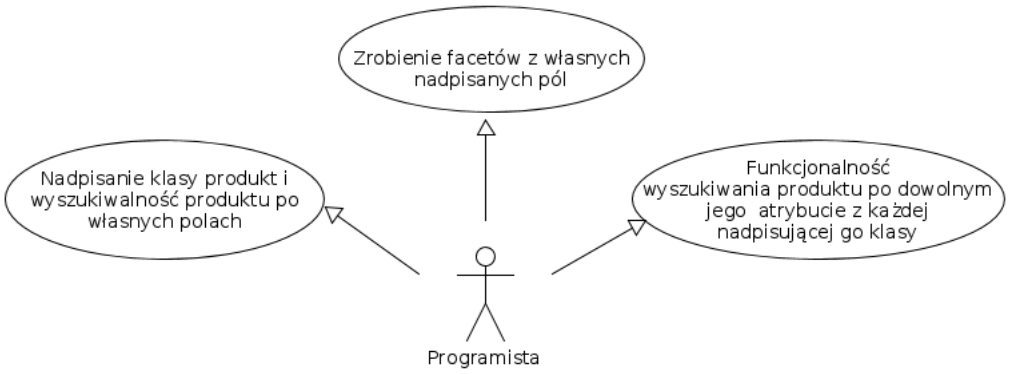
Dynamiczny formularz encyjny to kolejne autorskie rozwiązanie, służące do dodawania edycji i wyświetlania szczegółów encji związanych z systemem w panelu administracyjnym, wiążą się z nim przpadki użycia z rysunku 3.6



Rysunek 3.6: Diagram przypadków użycia związany z dynamicznym formularzem encyjnym

Manipulacja produktem

Produkt w implementowanym systemie jest to encja bardzo dynamiczna, łatwo konfigurowalna. Wiążą się z nim przypadki użycia znajdujące się na rysunku 3.7



Rysunek 3.7: Diagram przypadków użycia związany z dynamicznym formularzem encyjnym

Diagramy aktywności

W tej sekcji zostały przedstawione diagramy aktywności dla najbardziej skomplikowanych logicznie elementów systemu. Dynamiczna tabela i formularz opisany w poprzednim punkcie wymagają skomplikowanych operacji aby mogły pozostać ogólne i elastyczne na tyle ile być muszą. Znaczy to, że powinny obsługiwać zmiany w modelu wywołane przez osoby trzecie - **programistów**. W niniejszym podrozdziale przedstawiono działanie algorytów stojących za dynamicznymi elementami platformy.

Wyszukiwanie cech produktu

Ta sekcja odnosi się do podrozdziału **Manipulacja produktem**. Wytłumaczono tutaj jak osiągnięto założoną elastyczność przy konfigurowaniu encji reprezentującej produkt w sklepie.

Każdy Produkt w systemie jest indeksowany, oznacza to że z relacyjnej bazy danych, wraz ze swoimi wszystkimi atrybutami trafia do nierelacyjnych dokumentów na osobnym serwerze, aby odciążyć aplikacje w razie dużego ruchu. Zebranie wszystkich atrybutów produktów to skomplikowane zadanie, gdyż jego cechy mogą być ukryte w następujących miejscach:

- atrybuty dziedziczone po atrybutach klasyfikacyjnych kategorii, w której się znajduje
- atrybuty dziedziczone po wszystkich przodkach swojej kategorii
- własne pola i pola wszystkich klas, które nadpisały Produkt

Zadanie to wymaga zejścia do poziomu refleksji w Javie, jednak ten temat został poruszony później. Wyszukiwaniem wszystkich atrybutów obsługuje algorytm, którego diagram aktywności został umieszczony na rysunku 3.8

Pierwszym krokiem jest znalezienie wszystkich produktów, co nie jest również oczywistym zadaniem, gdyż nie jest wiadome jaką klasę ma finalny produkt, mógł zostać nadpisany przez programistę, który w swojej klasie zdefiniował pewne pola, które również muszą zostać uwzględnione przy indeksacji produktów. Po znalezieniu *klasy sufitowej* (czyli najwyższej w abstrakcji) mamy pewność, że wszystkie niestandardowe pola znajdą się w obiekcie pobranym przez nas z bazy danych. Z bazy danych muszą zostać również pobrane pola zadeklarowane jako te, które są wyszukiwalne w sklepie

(oczywiste jest, że powinno się mieć wybór, które pola z produktu trafią do sklepu, a które nie). Mając te dwie rzeczy, jest możliwe wyciągnąć z obiektu, którego klasa nie jest znana wszystkie interesujące nas pola. Wszystkie wyciągnięte wartości następnie trafiają do dokumentu. Nie jest to jeszcze koniec, gdyż zostają nam jeszcze do wyciągnięcia cechy produktu z systemu klasyfikacyjnego, zostało to zrealizowane algorytmem przejścia po drzewie oraz zapytaniem bazodanowym. Wszystkie cechy produktu są dodawane do listy dokumentów (jeden dokument - jeden produkt) i są wysyłane na serwer Apache Solr. Przykład skąd mogą pochodzić cechy produktu został umieszczone na rysunku 3.9.

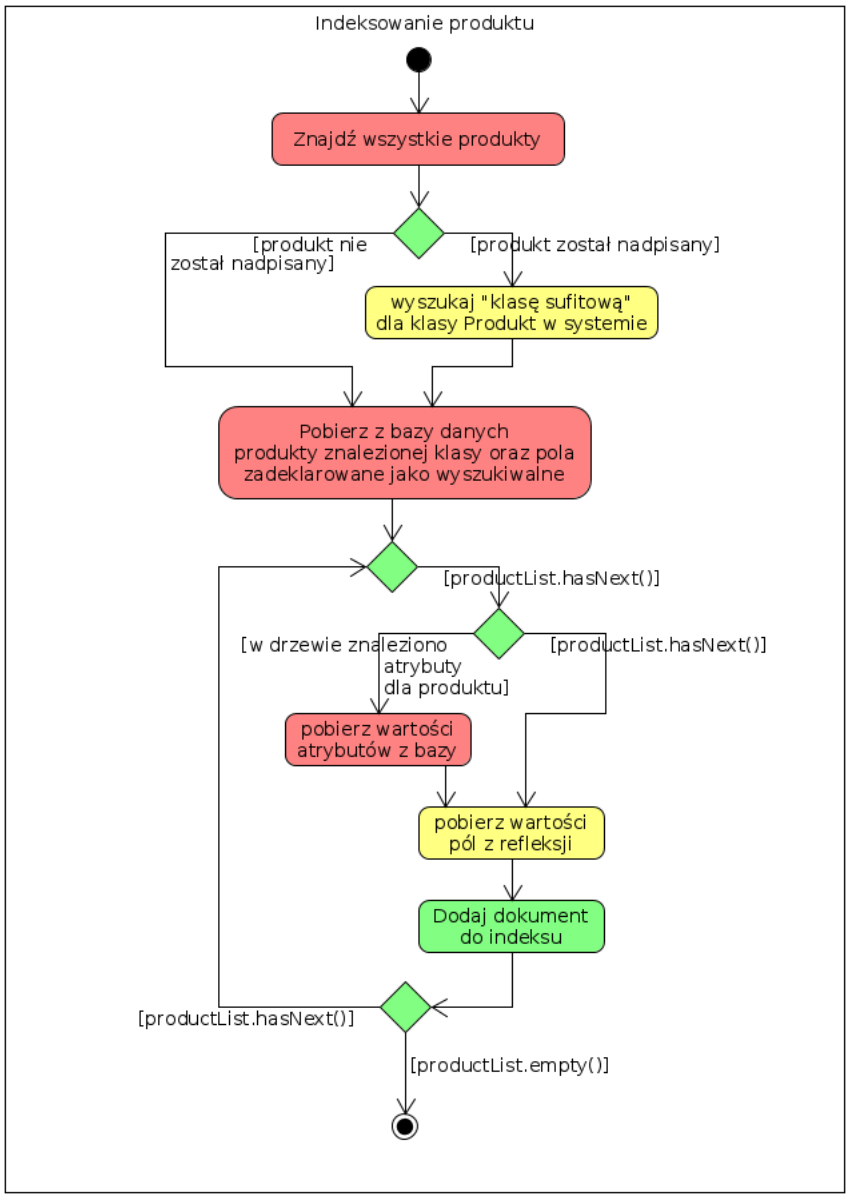
Konstrukcja zapytania dynamicznego Apache Solr

Na diagramach use case zostały opisane możliwości konfiguracji mechanizmu wyszukującego. Zdefiniowane wymagania zostały zrealizowane za pomocą konstrukcji dynamicznego zapytania. W prostych słowach chodzi tu o kwerendę zwracającą produkty przy wyszukiwaniu. Wykres na rysunku 3.10 przedstawia algorytm tworzący te zapytania. Pierwszy etap widoczny na rysunku nasuwa podejrzenie, że w bazie danych musi istnieć tabela z wylistowanymi polami, które będą wyszukiwalne w sklepie, jest to jak najbardziej trafne przypuszczenie. Dodatkowo w tabeli zostały umieszczone dodatkowe ustawienia dla pól wyszukiwania, aby mechanizm mógł być w pełni konfigurowalny.

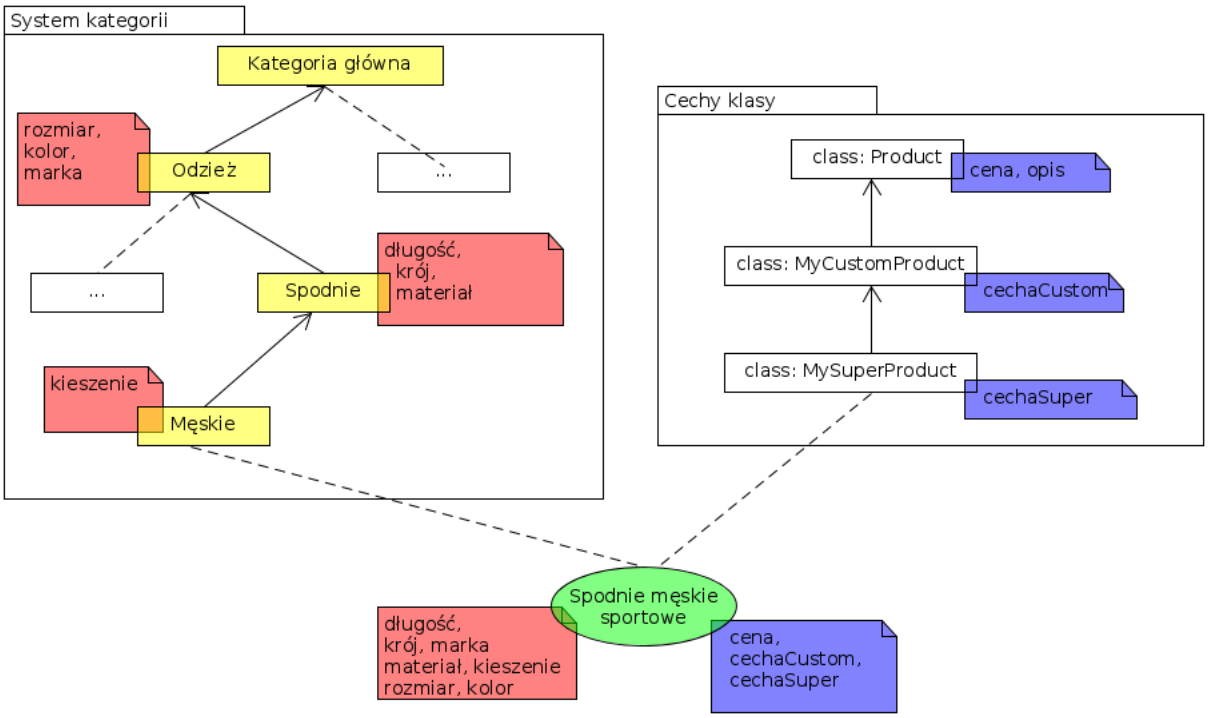
Konstrukcja dynamicznej tabeli encyjnej

Dynamiczna tabela encyjna opisana w poprzednim podrozdziale, jest wbrew pozorom zadaniem analogicznym do wyciągania cech z produktu. Ułatwieniem jest to, że w tabeli nie uwzględnia się atrybutów z systemu klasyfikacyjnego, gdyż dotyczy on tylko produktów, niestety utrudnieniem jest to, że nie jest wiadome jakie klasy dokładnie powinny być wyświetlone w tabelach. Jedyne co jest dane to tabela konfiguracyjna z kodami klas, które mają zostać wyświetlone w panelu administracyjnym. Proces został opisany na diagramie z rysunku 3.11

Najpierw szukane są encje z rodzaju podanego we wspomnianej tablei, następnie refleksją wyciąga się jej pola zaadnotowane przez programistę jako te, które chce wyświetlić w tabeli jako nagłówki (adnotacja `@AdminVisible(tableVisible=true)`). Ze znalezionej listy obiektów pobiera się wartości pól, które znajdują się w nagłówkach.



Rysunek 3.8: Diagram aktywności opisujący algorytm znajodwania wszystkich atrybutów produktu



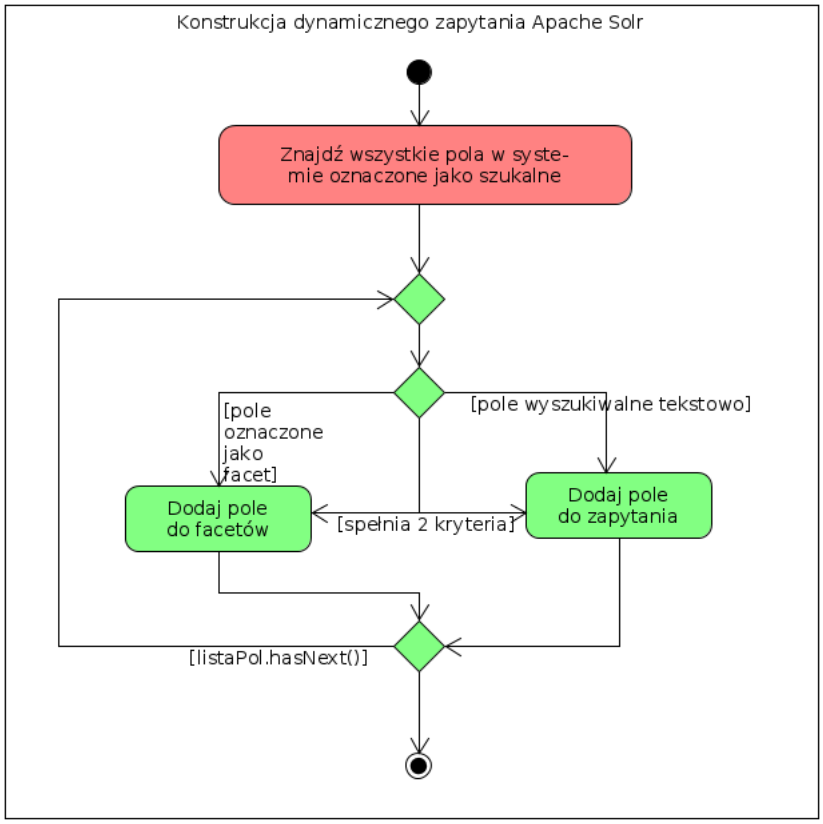
Rysunek 3.9: Diagram przykładowy skąd mogą pochodzić cechy produktu

Konstrukcja dynamicznego formularza encyjnego

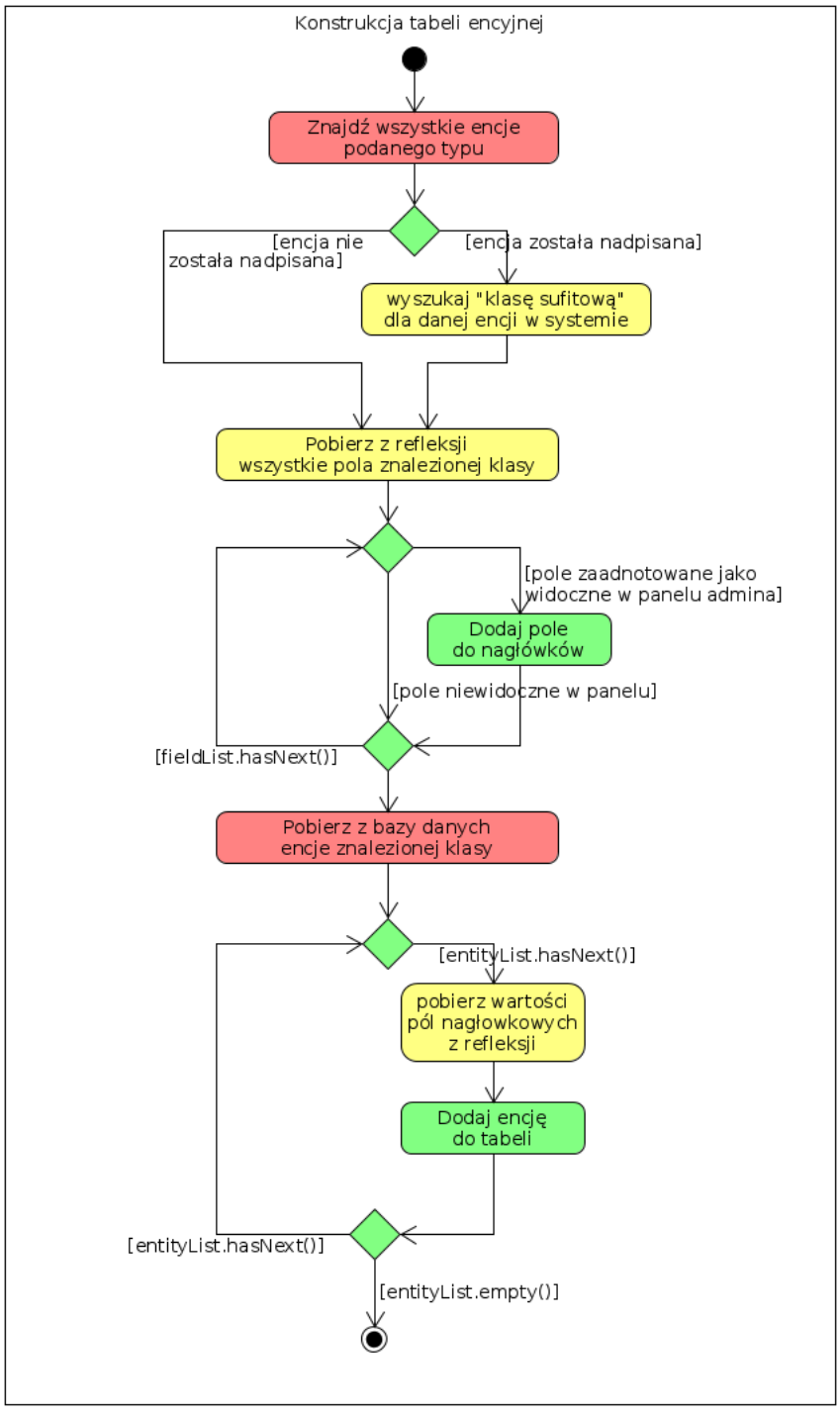
Odbywa się to na bardzo podobnej zasadzie, jak konstrukcja tabeli. Jednak poza polami prostymi zaadnotowanymi jako widzialne w panelu administracyjnym, problem stanowią relacje, które trzeba wyświetlić. Aby zachować rozszerzalność platformie zastosowane zostały tu mechanizmy z dwóch poprzednich przykładów. Po znalezieniu listy pól danej klasy, następuje tu jeszcze sklasyfikowanie ich względem tego, czy są to relacje czy nie. Wiadome jest, że relacji nie można przedstawić w postaci pola tekstowego, dlatego potrzebujemy wtedy ręcznie doczytać kolekcję w relacji z encją, której detale są wyświetlane. Wszystkie kolekcje są w systemie kolekcjami leniwymi (ze względu na performance), dlatego przy manipulacji encją jej duże elementy takie jak kolekcje są doczytywane dopiero w moemencie ich użycia(ponieważ wymagają joinów, a wiadome jest że join to iloczyn kartezjański, oczywiście odpowiednio napisany nie będzie dokładnie tym samym jednak to i tak duże obciążenie). W zwykłej sytuacji framework Hibernate doczytałby sam tę kolekcję, jednak przy tak dużej ogólności rozwiązania nie jest to możliwe, gdyż na poziomie dynamicznego formularza mówimy o `AbstractEntity`[4], o której właściwie w ogóle nie ma informacji, więc Hibernate nie jest *świadomy* manipulacji i nie doczyta jej kolekcji nawet będąc w sesji. To przysporzyło wiele kłopotów, jednak problem został rozwiązany ręcznym doczytaniem kolekcji za pomocą Criteria API[5]. Schemat algorytmu rozwiązującego problem został umieszczony na rysunku 3.12.

[4] AbstractEntity - klasa w systemie reprezentująca najbardziej podstawową encję, jej cechy to id oraz kod

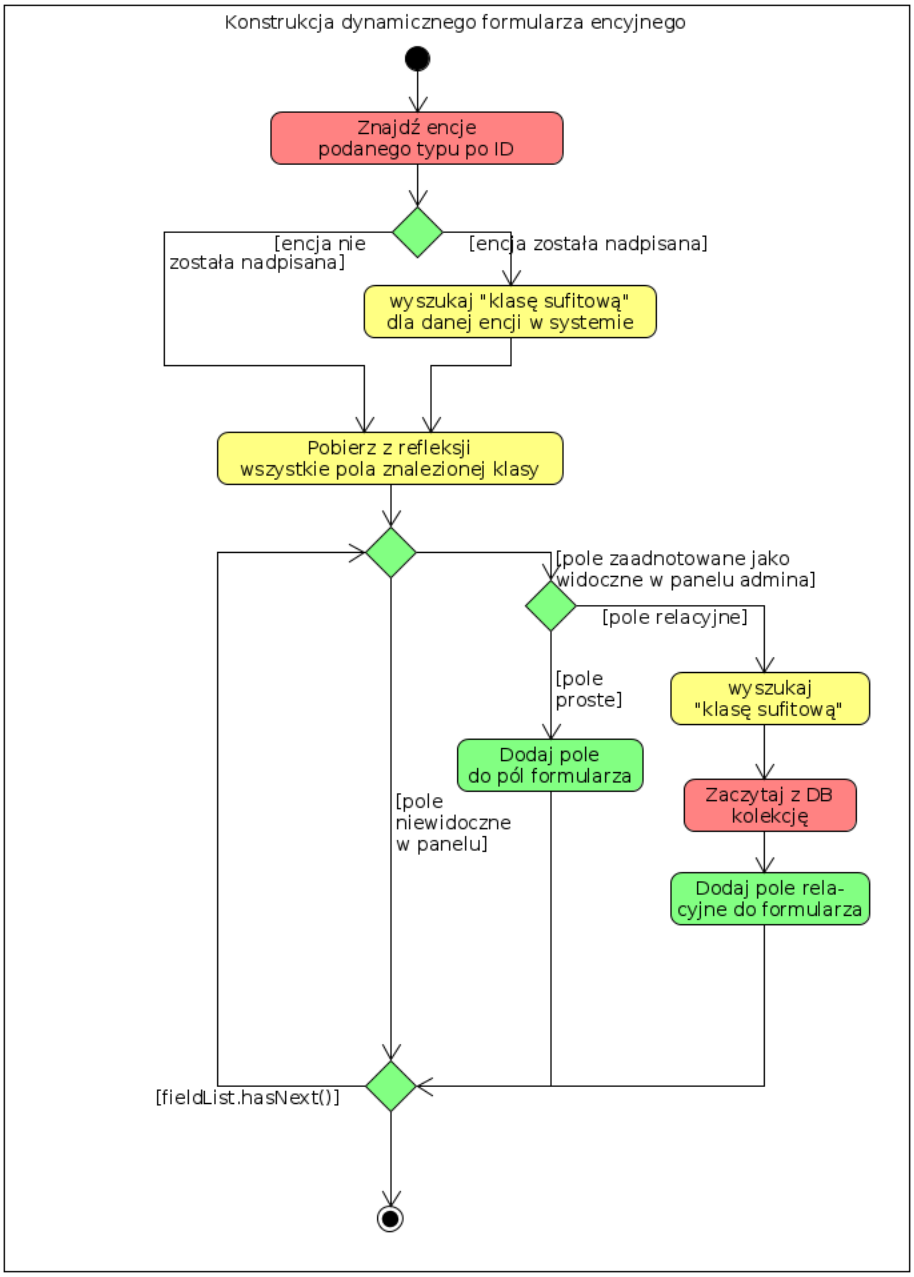
[5] Criteria API - API do komunikacji z bazą danych, zgodne se standardem JPA, używane do bardziej zaawansowanych i niestandardowych operacji



Rysunek 3.10: Diagram przedstawiający algorytm kostrukcji dynamicznego zapytania Apache Solr



Rysunek 3.11: Diagram aktywności opisujący algorytm wyszukujący cechy encji uwzględnianych w tabeli.



Rysunek 3.12: Diagram aktywności opisujący algorytm wyszukujący cechy encji uwzględnianych w tabeli.

Podsumowanie diagramów aktywności

Diagramy aktywności i zaprezentowane rozwiązania w nich pokrywają pewne przypadki użycia, w tej sekcji zostanie podkreślone jak przedstawione algorytmy spełniają zdefiniowane use case'y.

Patrząc na sekcję **Wyszukiwanie cech produktu** i **Konstrukcja zapytania dynamicznego Apache Solr** zostały zrealizowane następujące przypadki użycia:

- Programisty:
  - Rozszerzenie klasy produkt
  - Konfiguracja mechanizmu wyszukiwania
  - Indeksacja własnych pól
- Administratora:
  - Konfiguracja mechanizmu wyszukiwania
  - Dodanie atrybutów do produktu
  - Zmiana właściwości produktu
- Klienta sklepu
  - Wyszukiwanie produktu
  - Firtrowanie wyników

Sekcja **Konstrukcja dynamiczniej tabeli encyjnej** i **Konstrukcja dynamicznego formularza encyjnego** pokryły następujące przypadki użycia:

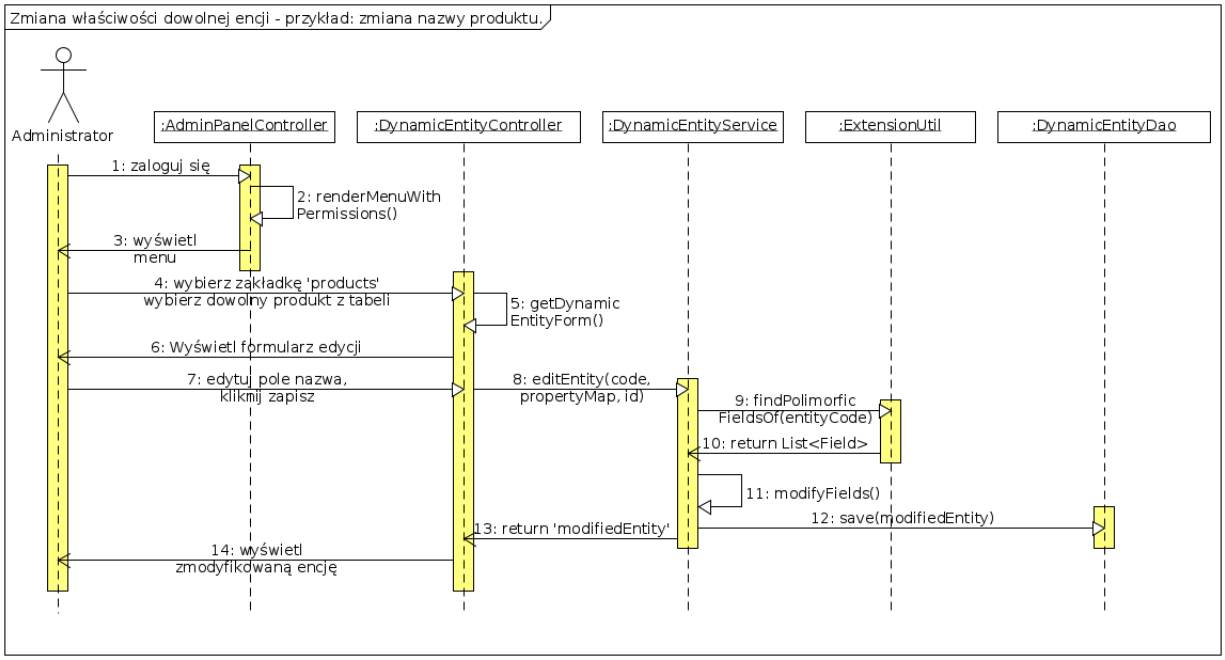
- Programisty:
  - Konfiguracja panelu administracyjnego
  - Dodanie własnych encji
  - Zarządzanie encjami
- Administratora:
  - Zmiana właściwości kategorii
  - Zarządzanie użytkownikami
  - Zarządzanie zamówieniami
  - Zarządzanie uprawnieniami

W podrozdziale opisującym przypadki użycia opisano również przypadki w sekcjach o dynamicznym formularzu, tabeli i manipulacji produktem, oczywiste jest, że wspomniane użycia bezpośrednio wiążą się z opisanymi komponentami.

Realizacja przypadków użycia, których nie mogły pokryć algorytmy opisane w niniejszym podrozdziale ze względu na swoje biznesowe pochodzenie, zostaną opisane na diagramach sekwencji.

Diagramy sekwencji

W tej sekcji zostały przedstawione diagramy sekwencji dla poszczególnych elementów systemu, mają one razem z diagramami aktywności zrealizować wszystkie przypadki użycia. Diakramy sekwencji dobrze oddają sens funkcjonalności biznesowych i właśnie ich dotyczy ta sekcja.

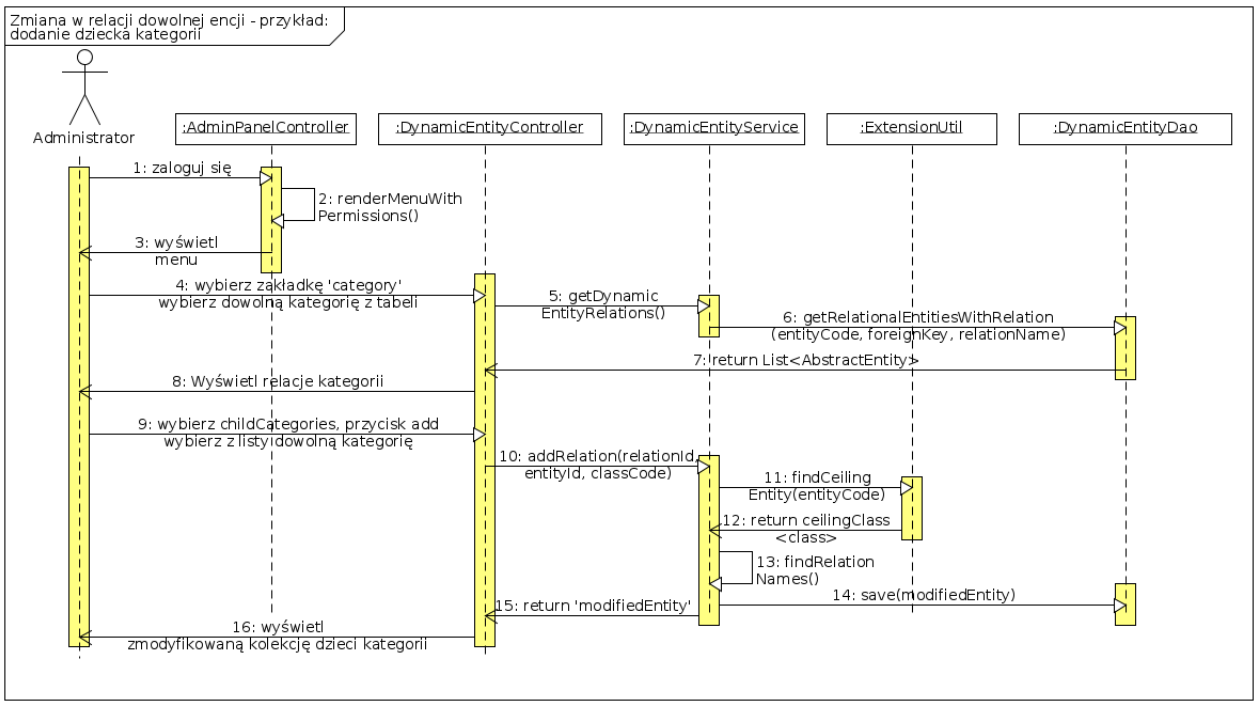


Rysunek 3.13: Diagram sekwencji opisujący opiisujący zmianę właściwości dowolnej encji na przykładzie produktu.

Zmiana właściwości dowolnej encji

W każdym systemie bazodanowym, konieczna jest manipulacja encjami. We wcześniejszych rozdziałach zostało wprowadzone pojęcie dynamicznych tabel i formularzy edycyjnych. Z punktu widzenia bazy danych, nie ma dużego znaczenia jaka encja jest edytowana, zarówno Produkt jak i Kategoria to jedynie zbiór krotek w tabeli relacyjnej, dlatego właśnie zdecydowano się na generyczny mechanizm modyfikacji encji. W prostych słowach, w systemie nie ma osobnych formularzy edycyjnych dla poszczególnych encji, są one generowane na podstawie kodu Javowego.

Funkcjonalność zmiany właściwości dowolnej encji zostanie omówiona na przykładzie zmiany nazwy produktu, ale jak już wyżej wspomniałem, ten sam mechanizm działa w dowolnej encji, która może być modyfikowana przez framework. Po zalogowaniu się i wyświetleniu menu, administrator wybiera dowolny produkt, po czym zmienia dane pole w formularzu edycji i następnie zapisuje produkt. Diagram 3.13 przedstawia cały proces. Najważniejsze etapy to 5 oraz od 8 do 13, to one odpowiedzialne są za elastyczność systemu - każda encja jest modyfikowana w ten sam sposób - generycznie, przez refleksję. W tym miejscu widać główny cel i zamysł pracy: system jest siadomy tego jaki obiekt edytuje, dlatego może mieć dostęp do jego właściwości (9. findPolimorficFieldsOf) - metoda ta zwraca wszystkie możliwe pola dla danego obiektu, właśnie dlatego, gdy programista nadpisze jakąś klasę, która jest używana w systemie, nic się nie stanie gdyż framework będzie w stanie wyciągnąć z obiektu każde nowe pole. Mechanizm ten zastosowany jest również we wszystkich kluczowych funkcjonalnościach, co skutkuje bardzo elastycznym modelem.



Rysunek 3.14: Diagram sekwencji opisujący opiisujący zmianę właściwości dowolnej encji na przykładzie produktu.

Modyfikacja dowolnej relacji encji

Dynamiczny formularz encyjny omówiony na diagramach aktywności zawiera również pola relacyjne, rysunek 3.14 przedstawia diakram sekwencji zmiany w relacji dowolnej encji na przykładzie dodania dziecka do kategorii. Okazuje się być dużym problemem dla frameworka, gdyż modyfikowany jest obiekt pewnej klasy i tylko na tej podstawie potrzeba określić jakie ma relacje i co więcej, wyciągnąć je z bazy danych. Problem został rozwiązany poprzez wprowadzenie do systemu dodatkowych parametrów adnotacji `@AdminVisible` takich jak nazwa klasy `className` i mapowanie relacji `mappedBy` - punkt 5 na rysunku 3.14, następnie dla wszystkich encji relacyjnych zostaje zbudowana dynamiczna tabela - tylko że wewnątrz formularza edycyjnego (pkt. 6 do 8). Następnie po kliknięciu *add* przy tabelce z relacją, pobierana jest lista możliwych do przypisania encji i w przypadku kliknięcia na dany rekord, zostaje on wprowadzony w relację z edytowaną encją. Kolejne punkty odpowiadają za elastyczność rozwiązania, czyli ponownie (jaj w poprzednich funkcjonalnościach), dla każdej znalezionej encji należy znaleźć klasę sufitową i to na jej podstawie wyciągać pola i tworzyć powiązania. Całość jest przedstawiona w formie pól tekstowych, checkboxów i tabelek z relacjami.

Dodanie dowolnej encji na przykładzie atrybutu klasyfikacyjnego

Ta ważna część frameworku zostanie przedstawiona na podstawie dodawania facetu - czyli atrybutu, po którym można filtrować produkt. Tak jak w powyższych sekcjach, nie ma dużego znaczenia jaką encję dodajemy - generyczny formularz i mechanizm uzupełniania pól w obiekcie danej klasy jest taki sam dla wszystkich encji w systemie. Dzięki diagramowi sekwecji z rysunku 3.15 można

zobaczyć jak dodać atrybut do produktu, który będzie wyszukiwalny, a zarazem zobaczyć jak w systemie persystowane są nowe obiekty. Tym razem refleksja została użyta do przepisania wartości z formularza do obiektu (punkty 8 do 12) oraz do znalezienia najwyższej w hierarchii klasy modyfikowanej encji (pkt 5).

Jak zostało już wspomniane, każdy atrybut może mieć wartości, które powinien przyjmować, np. *rodzaj podszewki: skórzana/gumowa/materiałowa*. Dlatego do atrybutu klasyfikacyjnego możliwe jest zdefiniowane jego przyjmowanych wartości, dzieje się to w dokładnie ten sam sposób, jak na rysunku 3.15, jedynie encja z `CategoryFeature` (atrybut klasyfikacyjny) zmienia się na `CategoryFeatureValue`. Po dodaniu możliwych wartości dla stworzonego atrybutu, należy je do niego przypisać w sposób opisany w sekcji **Modyfikacja dowolnej relacji encji** na diagramie z rysunku 3.14 oraz dodać atrybut z możliwymi wartościami do relacji kategorii (w formularzu edycji dla kategorii rys. 3.13 i 3.14). Atrybut zostanie uwzględniony w wynikach wyszukiwania opisanych w podrozdziale **Wyszukiwanie prduktu**. Aby rozjaśnić rozważania, został przytoczony przykład 3.3.

Przykład 3.3 *Załóżmy, że w mamy kategorię obuwie, chcielibyśmy aby każdy produkt w wynikach wyszukiwania w sklepie był możliwy do odfiltrowania na podstawie rodzaju podszewki. W tym celu dodajemy atrybut klasyfikacyjny Podszewka z zaznaczeniem, że ma być to facet (filtr). Dodajemy 3 wartości: skórzana/gumowa/materiałowa. Przypisujemy możliwe wartości do atrybutu, sam atrybut do kategorii obuwie, od tej pory w formularzu edycyjnym produktu, który znajdzie się w kategorii obuwie, znajdziemy pole rodzaj podszewki. Co więcej gdy więcej niż jeden znaleziony przy wyszukiwaniu w sklepie produkt będzie miał zdefiniowany dla siebie ten atrybut, to pojawi się on (atrybut) jako przycisk do odfiltrowania.*

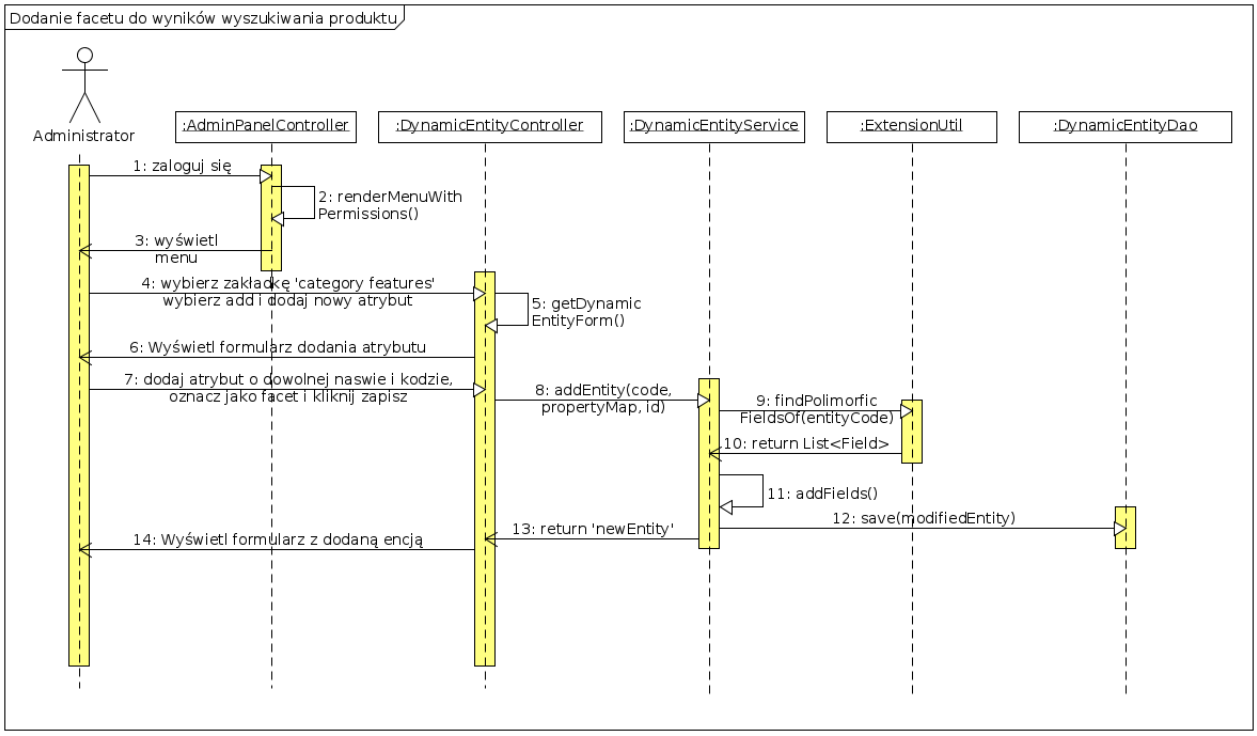
Wyszukiwanie produktu

Proces wyszukiwania produktu został ujęty na diagramie z rysunku 3.16. Aby przedstawić to jak najbardziej obrazowo, w procesie uczestniczą trzy komponenty: frameworkowy kontroler wyszukiwania, serwer Solr, który przechowuje produkty w płaskiej strukturze i baza danych relacyjna, która przechowuje pola i włąsciwości produktu podlegające mechnizmowi wyszukiwania. Na początku z bazy danych wciągane są właściwości i wszystkie zmienne, więc to czy dane pole ma być wyszukiwalne tekstowo, czy ma być filtrem (facetem). Następnie budowana jest dynamiczna kwerenda i wysyłana na serwer Apache Solr, który zwraca wyniki przetwarzane w punktach 10 do 12. Budowane są wtedy reprezentacje filtrów jak i samych wyników wyszukiwania, czyli produktów.

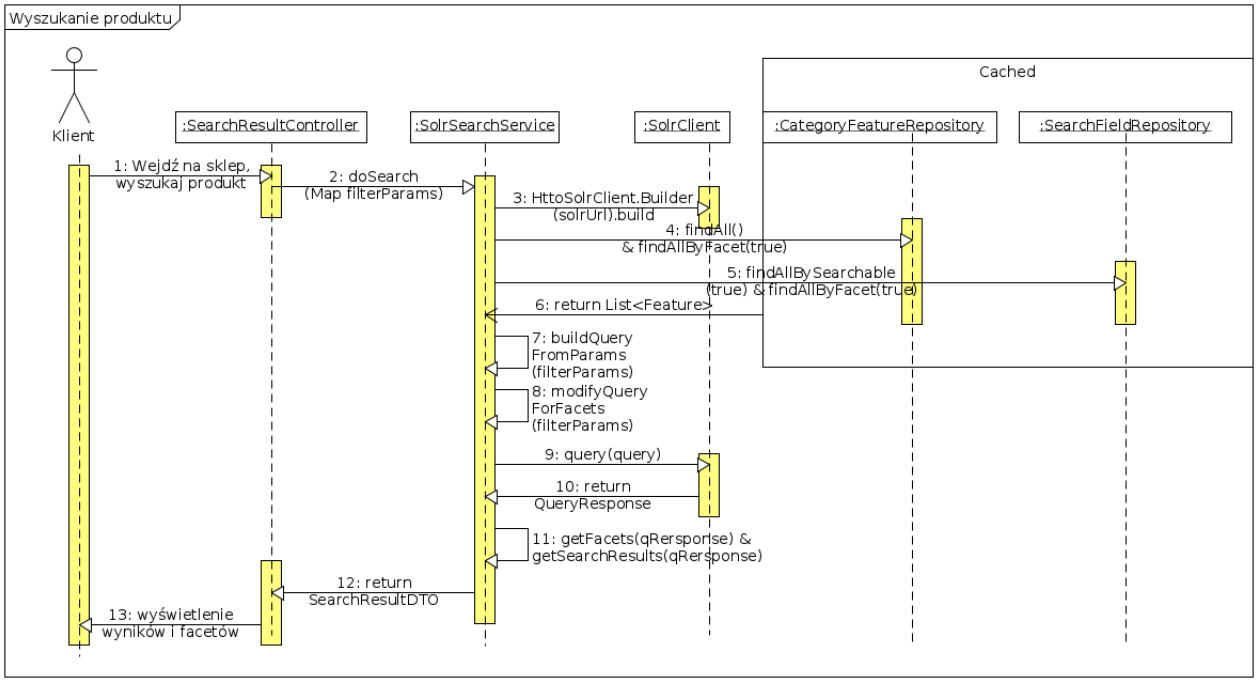
Wart zauważenia jest region wykresu w ramce *cached*, otóż jednym z postulatów frameworku było nie korzystanie z bazy danych relacyjnej przy wyszukiwaniu w katalogu produktowym (najbardziej obciążony fragment systemu), dlatego właściwości potrzebne do stworzenia kwerendy są pobierane raz, a później trzymane w pamięci cache (jest to mechanizm domyślnie dostarczany przez Hibernate, nie wymagał on implementacji).

Proces zakupowy

Proces zakupowy jest standardowym mechanizmem konicznym do przeprowadzenia transakcji, nie wymaga on szczególnej optymalizacji, ze względu na to, że nie jest to obciążona część platformy. Klient końcowy, po wejściu na sklep może wyszukać produkt oraz zdefiniować kryteria wyszukiwania. Po znalezieniu szukanych produktów może umieścić je w koszyku, a następnie zamówić elementy dodane do koszyka. Szczególnym rozwiązaniem dla frameworka jest tworzenie kopii produktu w momencie składania zamówienia, aby klient w razie reklamacji miał się do czego odwołać.



Rysunek 3.15: Diagram sekwencji opisujący dodanie dowolnej encji na przykładzie atrybutu klasyfikacyjnego.



Rysunek 3.16: Diagram sekwencji opisujący proces wyszukiwania produktu.

Rysunek 3.17: Diagram sekwencji opisujący proces zakupowy.

Ta funkcjonalność, jest umotywowana tym, że cechy produktu w systemie są bardzo dynamiczne i mogą być dowolnie zmieniane, więc produkt sprzedany w danym miesiącu może być niemożliwy do odnalezienia po pewnym czasie. Całość została umieszczona na diagramie z rysunku 3.17.

Diagramy klas

Na podstawie wcześniejszych rozważań możliwe jest zdefiniowanie następujących części frameworku:

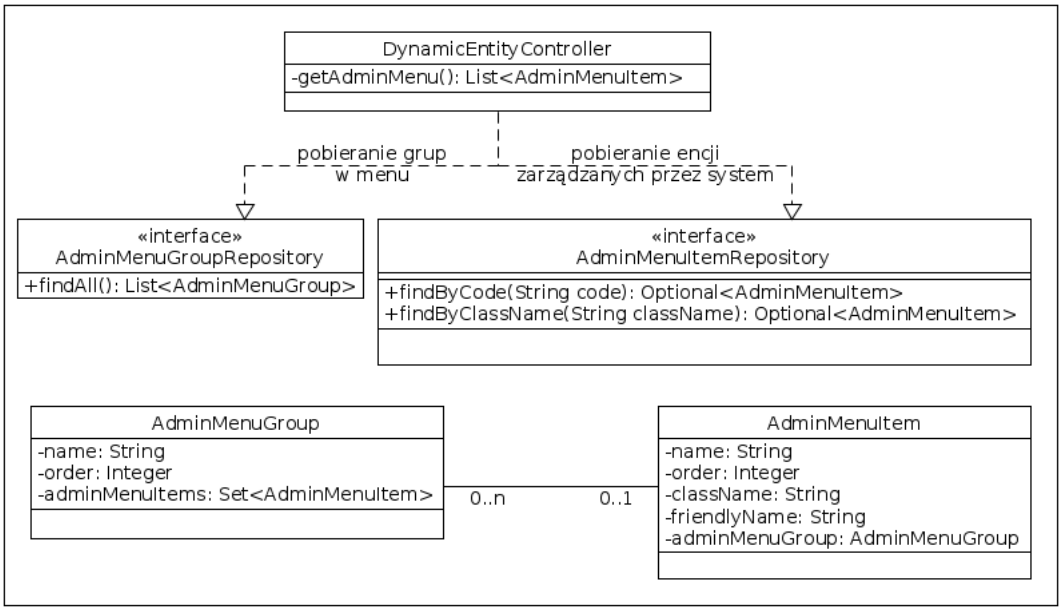
- części przewidziane dla programisty i administratora sklepu
  - konfigurowalne menu w panelu administracyjnym
  - dynamiczny formularz encyjny
  - dynamiczna tabela encyjna
- części przewidziane dla administratora sklepu
  - system klasyfikacyjny - warstwa modelowa
  - system klasyfikacyjny - warstwa serwisowa
  - system klasyfikacyjny - warstwa kontrolerów
  - struktura uprawnień w panelu administracyjnym
  - mechanizm indeksacji i wyszukiwania produktów
- część przewidziana dla użytkownika końcowego - klienta sklepu
  - mechanizm wyszukiwania produktów
  - proces zamówienia i archiwizacji produktu

Diagramy klas zostaną podzielone ze względu na pochodzenie funkcjonalności zdefiniowane wyżej.

Konfigurowalne menu

Na diagramie 3.18 został przedstawiony diagram klas dla menu panelu administracyjnego. Aby zapewnić możliwość obsługi dowolnej encji w panelu administracyjnym, które zostały zaimplementowane na potrzeby sklepu, np. Order (zamówienie), zostało stworzone menu, które jest oparte na klasie AdminMenuItem. Klasa ta bespośrednio odpowiada dowolnej encji. Oznacza to, że jeśli encja powinna być obsługiwana z panelu administracyjnego, to musi w systemie (jego bazie danych) istnieć jej odpowiednik w postaci AdminMenuItem z nazwą jej klasy (`className`). W systemie została przewidziana możliwość grupowania poszczególnych zakładek za pomocą klasy `AdminMenuGroup`. Budowanie menu odbywa się w klasie `DynamicEntityController`, która korzysta z klas połączonych z bazą danych, tak aby móc zaczytywać kolejne encje (`AdminMenuGroupRepository i AdminMenuItemRepository`). Dla lepszego zrozumienia został podany przykład **??**

Przykład 3.4 *W systemie istnieje encja Order, chcielibyśmy mieć możliwość zarządzania tą encją z poziomu panelu administracyjnego. W tym celu na poziomie bazy danych powinna być zapisany egzemplarz AdminMenuItem z nazwą klasy (className): com.example.model.Order (nazwa pakietu została podana tylko dla przykładu). System w tym momencie będzie już rozpoznawał encję Order, wyświetli ją w menu oraz będzie w stanie stworzyć dla niej tabele encyjną ze wszystkimi rekordami oraz formularz edycji.*



Rysunek 3.18: Diagram klas dla menu w panelu administracyjnym

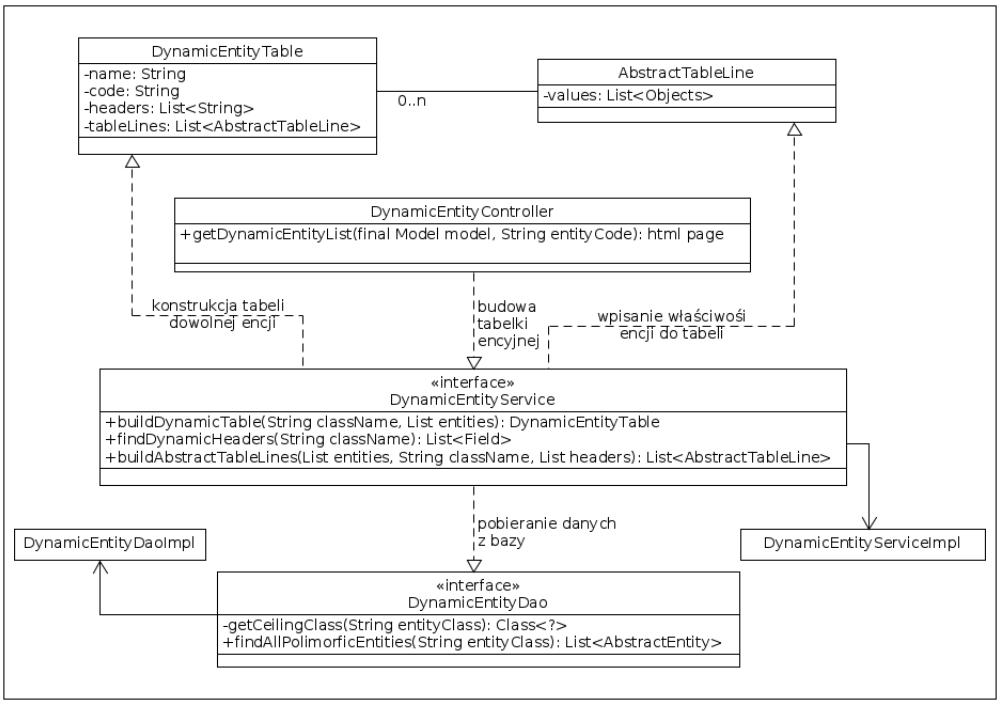
Dynamiczna tabela encyjna

Na rysunku 3.19 został umieszczony diagram klas dla dynamicznej tabeli encyjnej, o której była mowa w poprzedniej podsekcji. Tabela ta służy do wyświetlania wszystkich egzemplarzy danej encji w systemie, np. wszystkie produkty lub kategorie. Aby zamodelować tabelę zostały zaimplementowane dwie klasy: `DynamicEntityTable i AbstractTableLine`. Odpowiednio odpowiadają one całej strukturze tabeli oraz pojedynczemu wierszowi. Oczywiste jest, że tabela musi posiadać wiele wierszy. `DynamicEntityService` używa tych klas do konstrukcji tabelki. Serwis musi mieć możliwość pobierania danych z bazy, jednak jest to utrudnione zadanie ze względu na to, że nie jest z góry określone, że serwis ten zawsze konstruuje tabelkę klasy, np. `Product`. W tym celu została zaimplementowana klasa `DynamicEntityDao`. Jest to obiekt typu DAO[6], którego zadaniem jest wyciągać z bazy danych encje zdefiniowane w systemie, niezależnie od tego jakiej klasy są i czy nie zostały nadpisane. Stąd metoda `getCeilingClass(className)`, która odpowiada za znalezienie *najmłodszej* klasy w abstrakcji. Aby rozjaśnić ten mechanizm został podany przykład 3.5

Przykład 3.5 *W systemie istnieje wiele encji, np. Customer, Order, Product. W przypadku generowania dla nich tabeli, należy pamiętać o tym, że każda z nich może zostać nadpisana przez programistów budujących rozwiązania oparte o implementowany framework. Powiedzmy, że dodano klasę* `MyProduct extends Product`*. Abyśmy otrzymali dostęp do pól z tej nowej, nieznanej systemowi klasy potrzebujemy właśnie metody, która znajdzie za pomocą refleksji, klasę* `MyProduct`*, dzięki której system automatycznie, bez wkładu programisty będzie mógł uwzględnić w tabeli encyjnej pola, które zostały dopisane do produktu. To jest właśnie zadanie metody* `getCeilingClass`*.*

Kontroler `DynamicEntityControler` korzysta z serwisu, aby umieścić tabelę z encjami na stronie panelu administracyjnego pod konkretnym mapowaniem, np. */admin/entity/product/table*.

[6]DAO - Data Access Object - obiekt dostępu do bazy danych



Rysunek 3.19: Diagram klas dla tabeli encyjnej w panelu administracyjnym

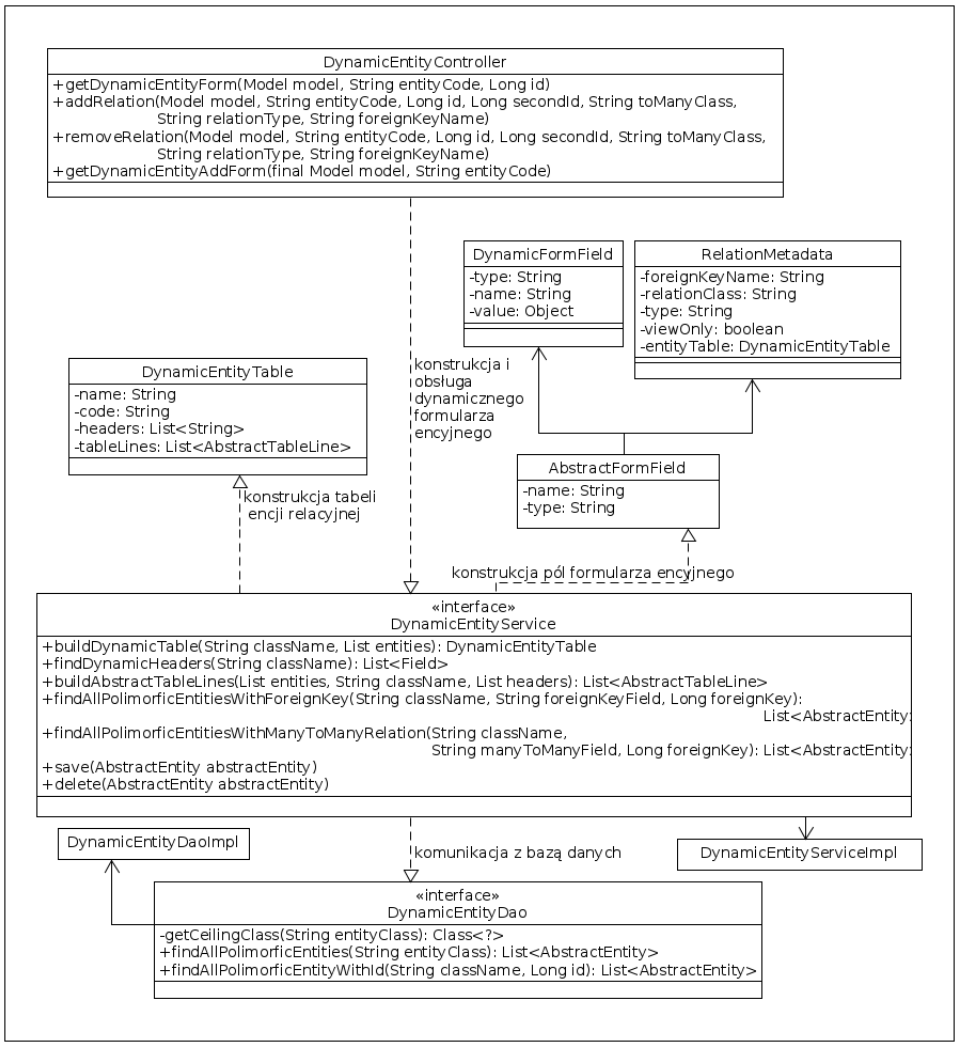
Dynamiczny formularz encyjny

Na rysunku 3.20 został umieszczony diagram klas dla dynamicznego formularza edycji encji. Zdefiniowano dla niego dwa rodzaje pól: `DynamicFormField i RelationMetadata`. Odpowiadają one odpowiednio polom prostym oraz relacyjnym. Każde pole relacyjne ma tabelkę z encjami (`DynamicEntityTable` - modelowo jest to ta sama tabelka co w poprzedniej podsekcji) będącymi w relacji z encją dla której generowany jest formularz, np. kolekcja cen w każdym produkcie. Dynamiczny serwis encyjny podobnie jak w przypadku tabeli, konstruuje formularz korzystając z opisanych wyżej modelowych klas. Posiada zestaw metod które:

- wyciągają wartości pól z encji za pomocą refleksji (tak eby wpisać je do formularza edycji)
- znajdują za pomocą refleksji relacje encji, dla której budowany jest formularz, z uwzględnieniem problemu nadpisania klasy opisanego w przykładzie 3.5
- obsługują CRUD edytowanej encji oraz jej relacji

`DynamicEntityController` w przypadku tej funkcjonalności obsługuje zapytania z serwera związane z wyświetleniem formularza encji, przetwarzaniem edycji oraz obsługą relacji encji (CRUD[7]). Klasa `DynamicEntityDao` spełnia takie samo zadanie jak w przypadku tabeli, wyciąga z bazy danych egzemplarze encji z uwzględnieniem możliwej abstrakcji.

[7]CRUD - CreateReadUpdateDelete - zestaw podstawowych operacji na encji, w przypadku tej funkcjonalności odnosi się do obsługi pól relacyjnych w encji, np. dodawania, usuwania, czytania i edytowania zdjęć do produktu



Rysunek 3.20: Diagram klas dla tabeli encyjnej w panelu administracyjnym

Model systemu klasyfikacyjnego

System klasyfikacyjny jest powiązany z drzewem kategorii. Pozwala na zdefiniowanie dziedzicznych cech dla każdej z nich, które pojawią się w końcowym produkcie przypisanym do kategorii. Poniżej znajduje się opis słowny poszczególnych klas ujętych na diagramie 3.21.
Cechy kategorii (`com.tinecommerce.core.catalog.model::Category`):

- dowolną liczbę dzieci i rodziców
- listę asocjacjji do zdefiniowanych jej cech
- przypisane do niej produkty

Cechy produktu (`com.tinecommerce.core.catalog.model::Product`):

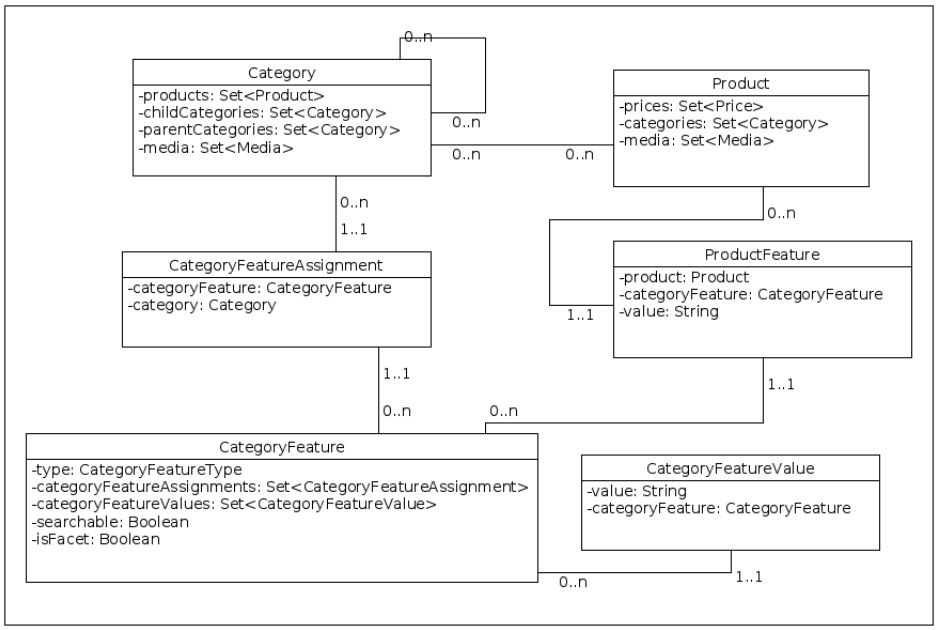
- lista cen
- lista przypisanych kategorii
- lista cech (ProductFeature)

`Product Feature` jest to klasa pośrednicząca pomiędzy strukturą drzewiastą systemu klasyfikacyjnego, a samym produktem, dlatego jej cechy to: produkt, cecha ketegorii oraz wartość cechy kategorii. Warto zauważyć, że z punktu widzenia modelu jest możliwe aby każdy produkt posiadał dowolną cechę wynikającą z klasyfikacji, ponieważ Product i CategoryFeature są *de facto* w relacji many to many. Jednak warstwa serwisowa (opisana w dalszej sekcji) jest odpowiedzialna za ograniczenie tych cech tylko do takich, które wynikają ze ścieżki w drzewie kategorii, która prowadzi od korzenia do samego produktu. Aby jaśniej nakreślić udział modelu w systemie klasyfikacyjnym został podany przykład 3.6

Przykład 3.6 *Kategoria (Category)* ***sport*** *posiada atrybut klasyfikacyjny (Category Feature)* ***waga sprzętu****, ich powiązanie definiuje się encją Category Feature Assignment. Atrybut klasyfikacyjny może mieć zdefiniowane wiele możliwych dla siebie wartości (Category Feature Value), np.* ***5KG, 2KG, 1KG****. Jeżeli jakiś produkt będzie należał do kategorii* ***sport****, to w jego fromularzu edycji pojawi się możliwe do zdefiniowania pole (Product Feature)* ***waga sprzętu****, której będziemy mogli nadać wartość* ***5KG, 2KG, 1KG****.*

Warstwa serwisowa systemu klasyfikacyjnego

Na rysunku 3.22 został umieszczony diagram klas warstwy serwisowej systemu klasyfikacyjnego. W opisie modelu zostało wspomniane, że warstwa serwisowa zajmuje się ograniczeniem atrybutów klasyfikacyjnych i przypisywaniem ich produktom. Ponadto zapewnia walidacje drzewa kategorii (aby nie powstawały cykle).
Serwis kategorii (CategoryService) jest interfejsem, który waliduje strukture drzewa kategorii, dodaje kategorie do drzewa (tworzy powiazania) oraz umożliwia wydobycie wszystkich przodków i potomków węzła. Serwis w przypadku znalezienia cyklu w drzewie kategorii rzuca wyjątek informujący o cyklicznym połączeniu między kategoriami (CircularEntityConnectionException) Interfejs posiada podstawową implementację w postaci klasy CategoryServiceImpl.
Serwis atrybutów klasyfikacyjnych (CategoryFeatureService) został zaimplementowany po to aby umożliwić pobranie wszystkich możliwych atrybutów klasyfikacyjnych dla dowolnego Produktu. To właśnie ten serwis jest odpowiedzialny za ograniczenie tych cech tylko do takich, które wynikają



Rysunek 3.21: Diagram klas systemu klasyfikacyjnego - model

ze ścieżki w drzewie kategorii, która prowadzi od korzenia do samego produktu. Sytuacja ta została opisana na diagramach aktywności w sekcji **Wyszukiwanie cech produktu**.

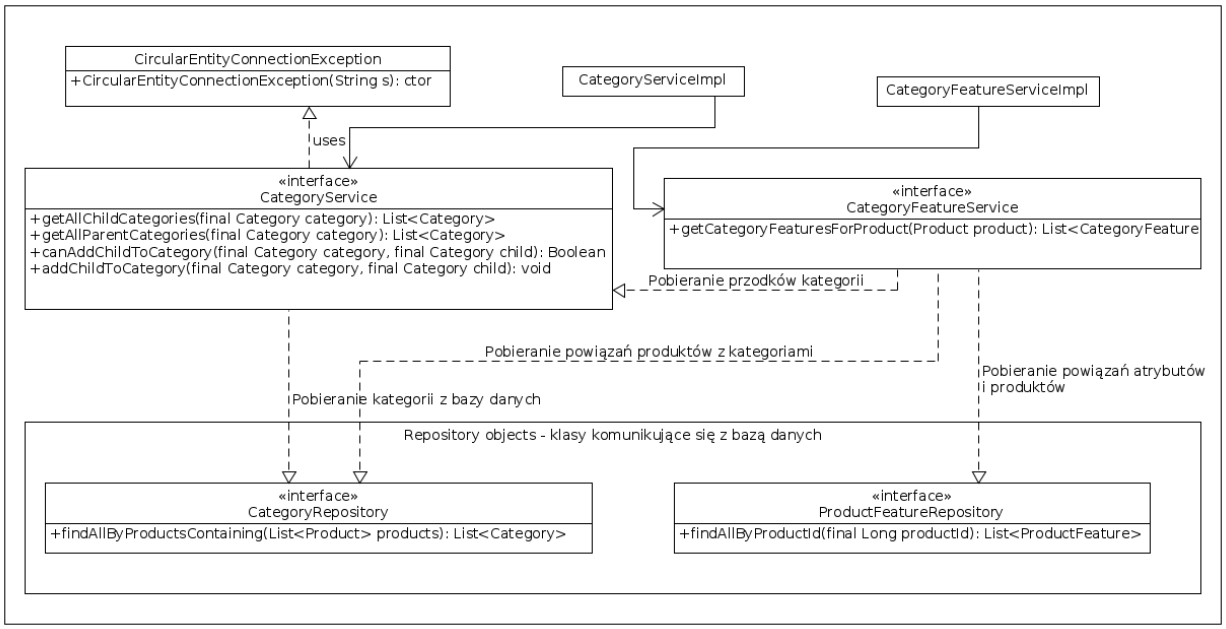
Serwisy mają również połączenie z bazą danych za pomocą obiektów *repository*. W tych klasach są konkretne metody pobierające encje z bazy danych. Każda metoda odpowiada kwerendzie bazodanowej, która została wygenerowana przez framework Hibernate na podstawie nazwy metody.

Warstwa kontrolerów systemu klasyfikacyjnego

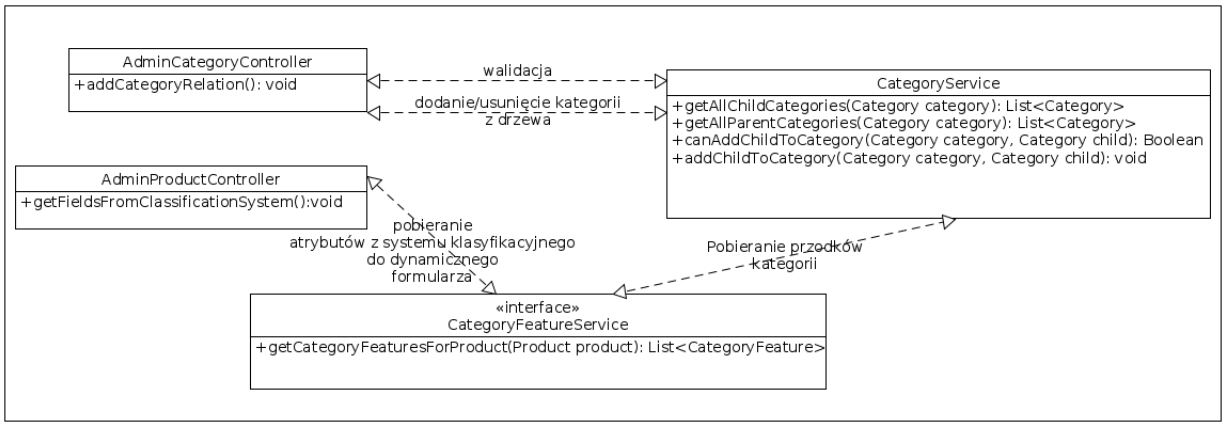
Rysunek 3.23 przedstawia diagram klas warstwy kontrolerów systemu klasyfikacyjnego. Panel administracyjny i edycja encji w systemie jest oparta na refleksji (czyli wyciąganiu wartości pól z klas), jednak w przypadku atrybutów klasyfikacyjnych nie mamy do czynienia z fizycznymi polami, dlatego potrzebujemy mechanizmu, który obsłuży przypisywanie ich wartości, dodawanie lub usuwanie. Kontrolery związane z systemem klasyfikacyjnym są rozszerzeniem ogólnego kontrolera **DynamicEntityController** (omówionego w poprzednich sekcjach), który zapewnia właśnie domyślne rozwiązanie oparte na refleksji. Więc kontrolery te powstały ze względu na konieczność obsługi nietypowych pól w formularzu edycji produktu jak i kategorii. W przypadku produktu są to pola, które nie należą bezpośrednio do klasy Product, jednak mogą mieć zdefiniowaną wartość - czyli wartości atrybutów klasyfikacyjnych. Natomiast w przypadku kategorii wiąże się to z koniecznością walidacji dodawania kolejnych powiązań w drzewie kategorii.

Struktura uprawnień w panelu administracyjnym

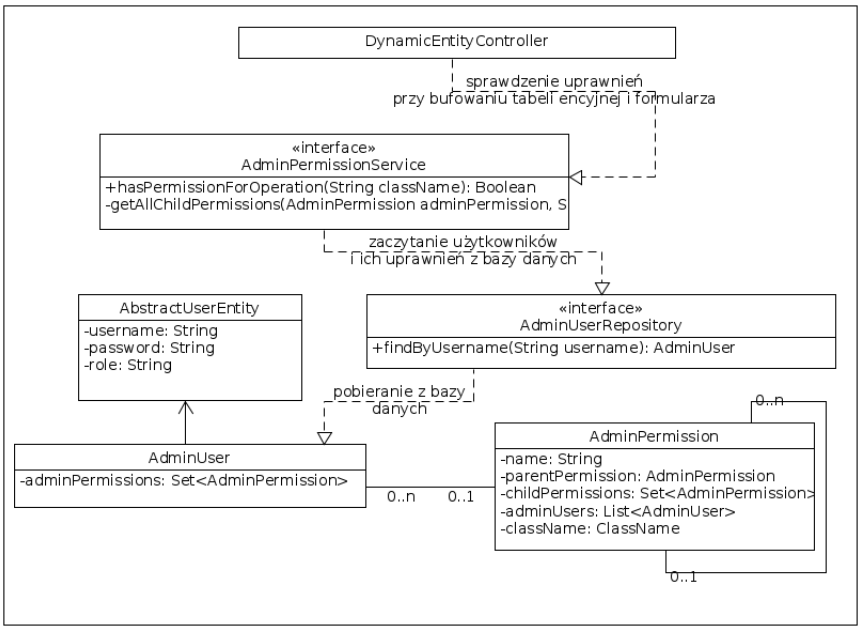
Panel administracyjny wyposażony jest w drzewiasty system uprawnień, który zapewnia kontrole dostępu do encji dla różnych użytkowników. Diagram klas użytych w tym rozwiązaniu został przedstawiony na rysunku 3.24. `DynamicEntityControler` w przypadku tej funkcjonalności sprawdza w



Rysunek 3.22: Diagram klas systemu klasyfikacyjnego - serwisy



Rysunek 3.23: Diagram klas systemu klasyfikacyjnego - kontrolery



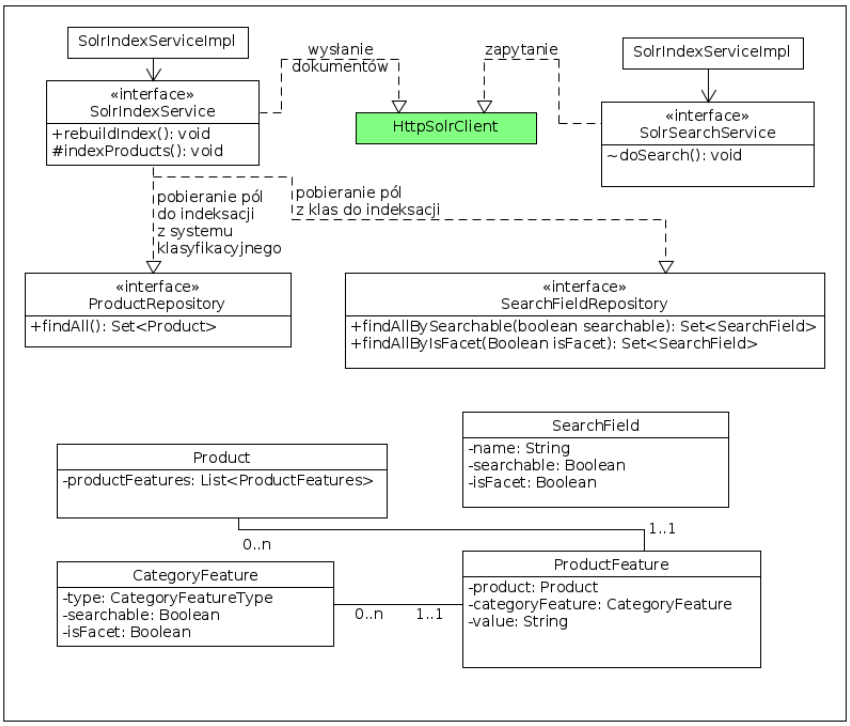
Rysunek 3.24: Diagram klas systemu uprawnień

swoich metodach CRUD dotyczących encji, czy zalogowany użytkownik ma prawo do wyświetlenia i edycji encji, w którą ma zamiar wejść. Dzieje się to za sprawą metody `hasPermissionForOperation (className)` w klasie `AdminPermissionService`, która wędrując po drzewie uprawnień sprawdza czy dany użytkownik ma powiązanie z uprawnieniem potrzebnym do wyświetlenia danej encji. Obiekty z bazy danych są pobierane za pomocą klasy `AdminUserRepository`. System skonfigurowany jest domyślnie na to aby jeden egzemplarz klasy `AdminPermission` odpowiadał jednej klasie encyjnej (np. cenie - Price). Uprawnienia mają również listę dzieci, dlatego można je układać w drzewiaste struktury. Działanie struktury objaśni przykład 3.7

Przykład 3.7 *Gdy użytkownik administracyjny* `AdminUser` *otrzymuje dane uprawnienie A, jest w stanie wyświetlić i edytować encję, której dotyczy to uprawnienie A jak i również wszystkie inne encje, których dotyczą uprawnienia, dla których A jest rodzicem.*

Indeksacja i wyszukiwanie produktów

W systemie zostały zaimplementowane dwa główne serwisy obsługujące zapytania związane z wyszukiwaniem i widzialnością produktów na sklepie. Pierwszy z nich to `SolrIndexService`, którego zadaniem jest wydobyć atrybuty produktu z systemu klasyfikacyjnego oraz z pól jego klasy jak zostało to przedstawione na rysunku 3.9. Potrzebuje on w tym celu dwóch obiektów dostępów do bazy dancyh. `ProductRepository` wyciągnie produkty (`Product`) wraz z kolekcjami ich dynamicznych cech z systemu klasyfikacyjnego (`ProductFeature/CategoryFeature`). Natomiast `SearchFieldRepository` wyciągnie nazwy pól, które zosatły zdefiniowane w systemie jako potrzebne do indeksacji (np. nazwa, opis lub cena). Każdy egzemplarz klasy `SearchField` odpowiada więc pewnej cesze produktu, która ma być wyszukiwalna. Cechy te zostaną wyciągnięte z klasy `Product` i jego ewentualnych nadklas (np. opisywany wielokrotnie hipotetyczny *MyProduct*). Wszystkie atrybutu produktu są pakowane w tak zwane *dokumenty* (jeden dokument odpowiada

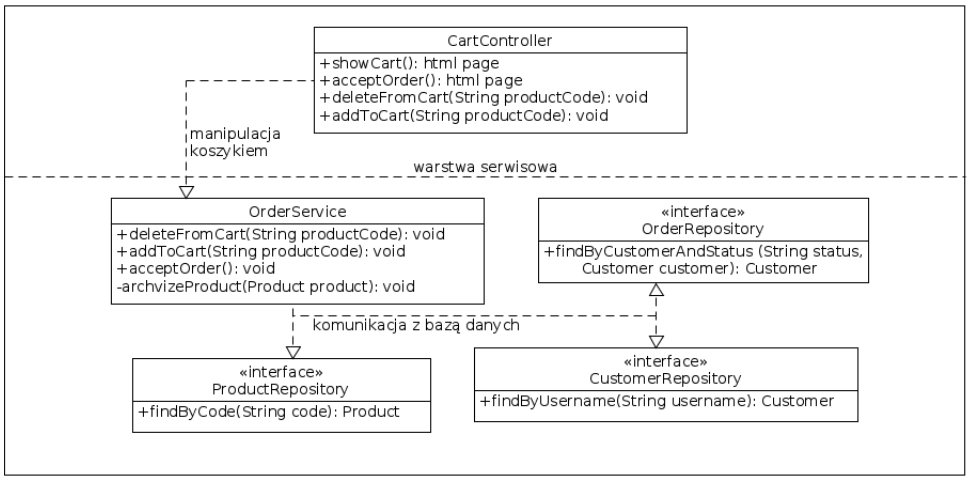


Rysunek 3.25: Diagram klas mechanizmu indeksująco-wyszukującego

jednemu produktowi) i wysyłane na serwer Apache Solr. Zaindeksowany Solr jest wtedy gotowy do przyjmowania zapytań dotyczących wyszukiwania. **SolrSearchService** jest odpowiedzialny za konstruowanie kwerend Solrowych i interpretacje wyników wyszukiwania.

Proces zamówienia

Jeden z najbardziej charakterystycznych procesów we frameworku e-commerce to procces zamówienia, diagram klas użytych do implementacji tego procesu został umieszczony na rysunku 3.26. `CartController` odpowiada za obsłuje zapytań związanych z operacjami na koszyku takimi jak dodawanie, usuwanie, złożenie zamówienia, wyświetlenie koszyka. Serwis zamówień `OrderService` obsługuje logikę tych zapytań wraz z funkcjonalnością archiwizacji cech produktu (metoda `archvizeProduct(Product product)`). Serwis używa singletonów Repository, aby znajdować w bazie danych zamówienia rozpoczęte przez zalogowanych użytkowników, produkty i szczegóły kupujących.



Rysunek 3.26: Diagram klas procesu zamówienia

Projekt bazy danych

W tej sekcji został przedstawiony projekt bazy danych. Dla zachowania ciągłości i przejrzystości rozważań diagramy zostały podzielone ze względu na funkcjonalności, podobnie jak na diagramach klas:

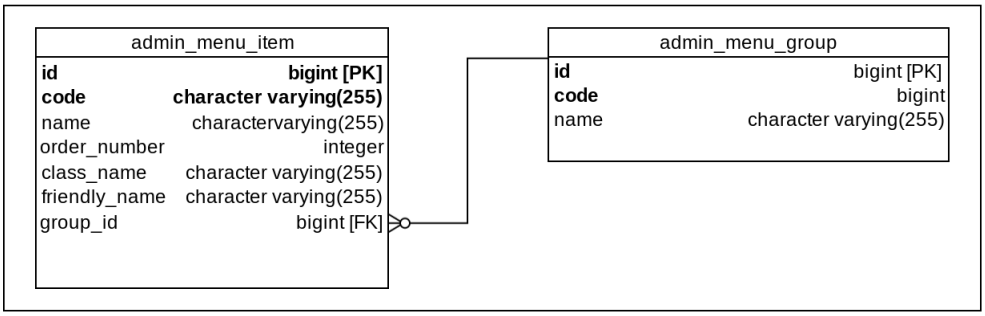
- menu w panelu administracyjnym
- system klasyfikacyjny
- struktura uprawnień
- proces zamówienia i archiwizacji produktu

Następnie w podsekcji **Mechanizm wyszukiwania** została opisana nierelacyjna baza Apache Solr i jej korelacje z relacyjną bazą danych. Na samym końcu została omówiona platforma DBMS i normalizacja relacyjnej części bazy danych.

Warto zaznaczyć w tym miejscu, że większość tabel ma kod (`code`), który jest unikalnym kluczem, po którym (zaraz obok id) można wyciągać encje z tabeli. Takie rozwiązanie zwiększa bezpieczeństwo aplikacji webowej, gdyż na stronie sklepu dostępnej dla osób trzecich nie są ujawniane klucze główne, tylko kody, które jednoznacznie określają encję w tabeli. Pogrubione kolumny w tabelach oznaczają unikalność wartości.

Menu w panelu administracyjnym

Na rysunku 3.27 został umieszczony diagram fragmentu bazy danych odpowiadający za menu w panelu. Zostało ono oparte na dwóch tabelach relacyjnych `admin_menu_item` (zakładka w menu) oraz `admin_menu_group` (grupa zakładek). Do pierwszej z nich składa się z id, unikalnego kodu, nazwy klasy, jaką ma reprezentować dana zakładka, kolejności wyświetlania, nazwy i nazwy wyświetlanej (`friendly_name`) oraz `group_id`, czyli klucz obcy grupy, w której się znajduje. Te dwie tabele łączy relacja one to many, w której rodzicem jest `admin_menu_group`. Nie jest konieczne aby grupa musiała zawierać jakiekolwiek elementy, podobnie nie jest obligatoryjne aby element musiał być przypisany do konkretnej grupy.



Rysunek 3.27: Diagram fragmentu bazy danych odpowiadającego za menu

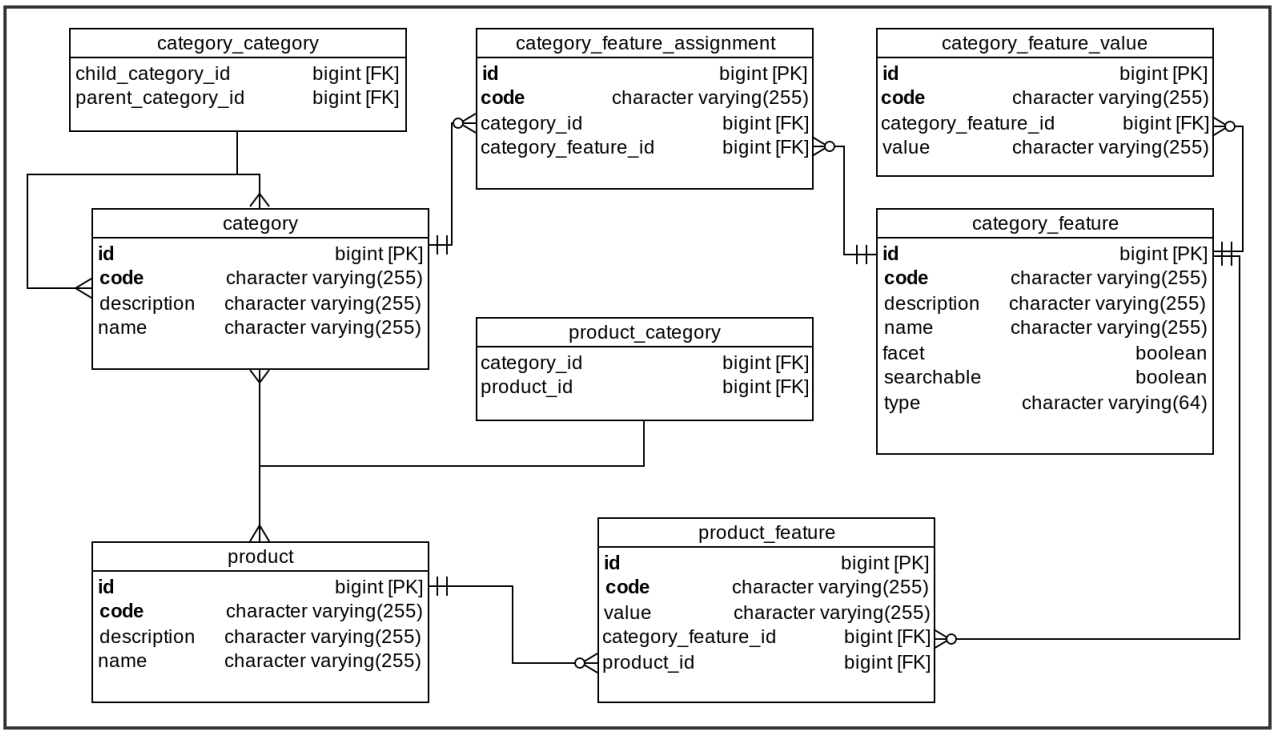
System klasyfikacyjny

Diagram na rysunku 3.28 przedstawia część bazy danych odpowiadającą za system klasyfikacyjny. Ze względu na obszerność diagramu, tabele zostaną omówione jedna po drugiej.

- `category` – główna tabela zawierająca kategorię produktów dostępnych w systemie, kategorie mogą przyjmować strukturę drzewa, w którym każdy węzeł ma dowolną ilość dzieci i rodziców, dlatego tabela jest w relacji many to many ze samą sobą.
- `product` – tabela przechowująca produkty dostępne w systemie, powiązana relacją many to many z kategoriami, tym sposobem każdy produkt może być w dowolnie wielu kategoriach oraz każda kategoria może być przypisana do wielu produktów.
- `category_feature` – w tej tabeli przechowuje się atrybuty klasyfikacyjne, które mogą zostać przypisane do kategorii, oprócz standardowych kolumn takich jak nazwa lub opis, są również flagi, które informują system o tym czy dany atrybut ma być wyszukiwalny pełnotekstowo na sklepie (`searchable`) lub czy ma być filtrem ograniczającym wyniki wyszukiwania (`is_facet`). Tabela ta jest w relacji many to many z tabelą `category`, co umożliwia kategorii mieć wiele atrybutów i atrybutowi znaleźć się w wielu kategoriach.
- `category_feature_assignment` – jest to tabela asocjacyjna pomiędzy kategorią, a atrybutem klasyfikacyjnym. Posiada ona własne id i kod, gdyż jest również systemową encją.
- `category_feature_value` – każdy atrybut klasyfikacyjny może mieć predefiniowane wartości, które składowane są w tej tabeli. `category_feature_value` jest powiązane relacją one to many z `category_feature` z kluczem obcym `category_feature_id`.
- `product_feature` – jest to tabela, która pozwala na powiązanie produktu atrybutami klasyfikacyjnymi jest w relacji one to many z tabelą `product i category_feature`, ma w tym celu dwa klucze obce `category_feature_id` i `product_id`. Pozwala również przypisać wartość atrybutowi w produkcie. Dla uproszczenia można powiedzieć, że to kolekcja cech produktu.

Mechanizm uprawnień

Rysunek 3.29 przedstawia fragment bazy danych odpowiedzialny za obsługę uprawnień w panelu administracyjnym. Tabela `admin_user` zawiera rekordy z użytkownikami administracyjnymi, którzy

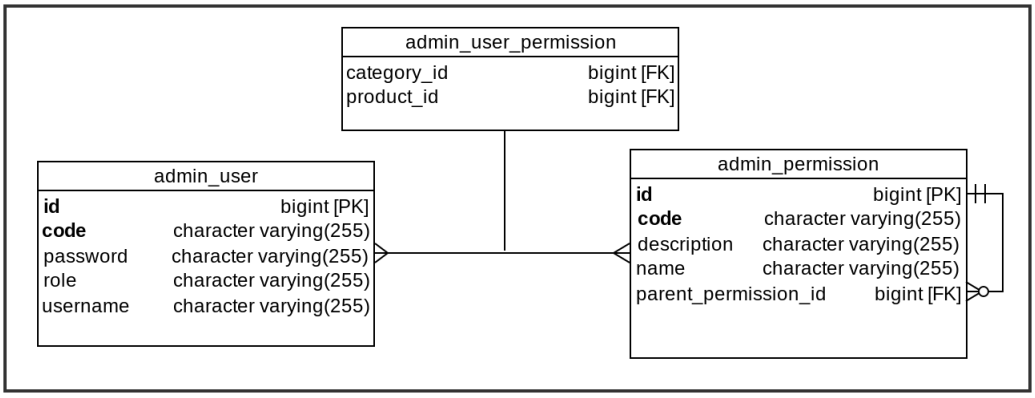


Rysunek 3.28: Diagram fragmentu bazy danych odpowiadającego za menu

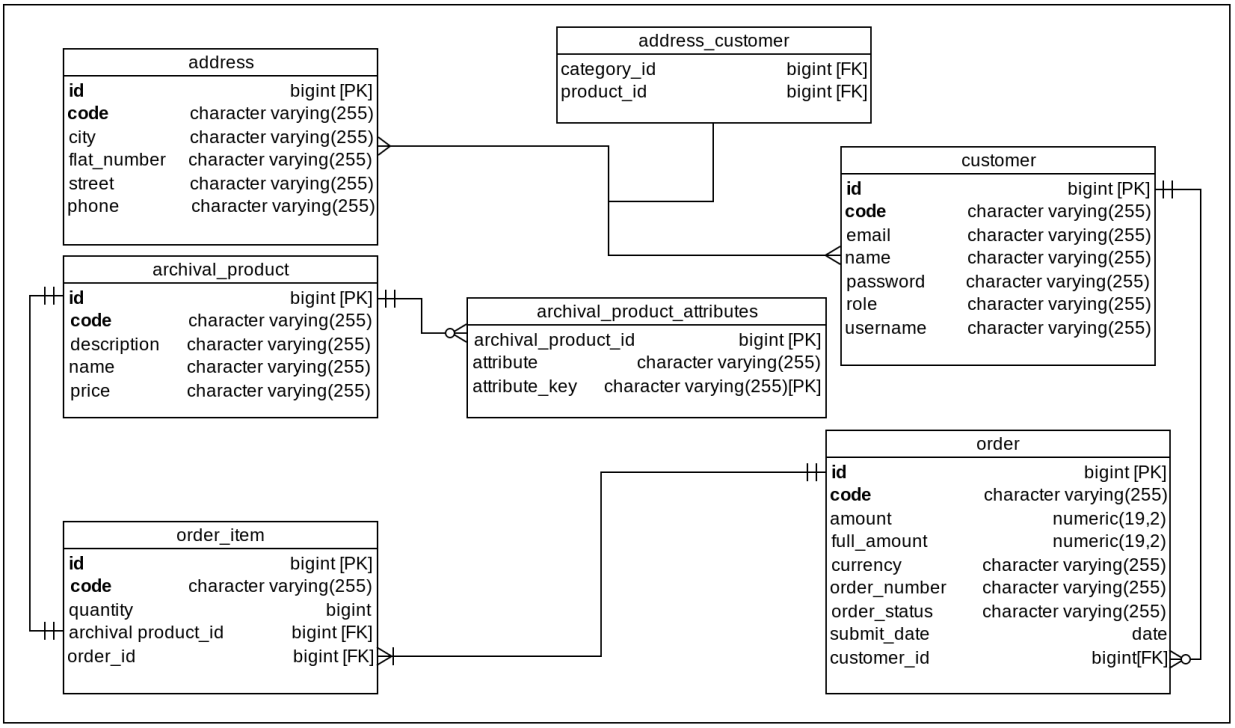
mogą zalogować się do systemu. Tabela ta jest w relacji many to many z `admin_permission`, co oznacza, że użytkownik administracyjny może mieć wiele uprawnień, jak i jedno uprawnienie może być przypisane do wielu użytkowników. Dodatkowo `admin_permission` posiada kolumnę `class_name` z nazwą klasy w systemie, której dotyczy uprawnienie. Aby stworzyć możliwość dziedziczenia uprawnień i układania ich w grupy, dodano klucz obcy `parent_permission_id` tworząc relację one to many do samego siebie. W ten sposób można tworzyć z uprawnień drzewiaste struktury – oczywiście przy założeniu niewystępowania cykli.

Proces zakupowy

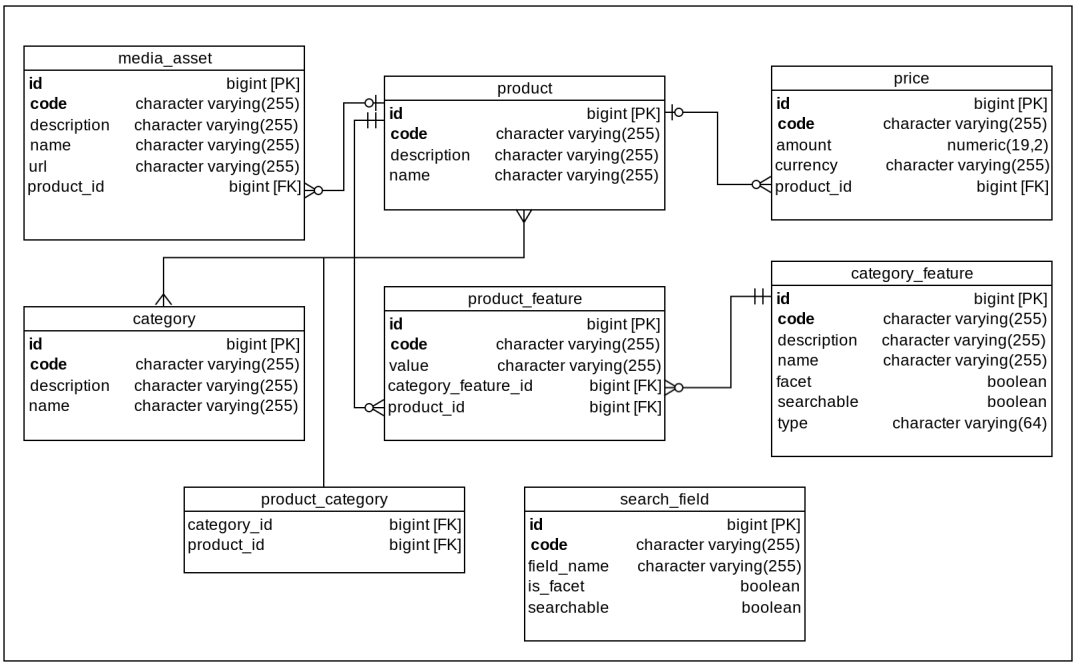
Diagram na rysunku 3.30 przedstawia tabele bazodanowe, które biorą udział w porocesie składania zamówienia. Oczywistymi elementami są tabele `customer i address`, powiązane relacją many to many. Zawierają one niezbędne szczegóły, które potrzebne są przy składaniu zamówień. Tabela `order` natomiast odpowiada systemowemu koszykowi, ma klucz obcy `customer_id`, co oznacza, że jest w relacji one to many z klientem. Daje to klientowi możliwość składania wielu zamówień. Jest ona również w relacji one to many (tym razem jako rodzic) z tabelą `order_item`. Ta reprezentuje jeden produkt dodany do koszyka w dowolnej ilości (za sprawą kolumny `quantity`). `order_item` posiada również relację one to one do tabeli `archival_product`, które pełni w sysetmie rolę archiwum produktowego, tak aby przy hipotetycznym usunięciu produktu ze sklepu zamówienie, które zostało już złożone nie zmieniło się. W celu utrzymania największej elastyczności w opisie archiwalnego produktu, stworzono dla niego w relacji one to many tabelę `archival_product_attributes`, do której jest możliwość zapisania prawie wszsytkich cech oryginalnego produktu. Te dwie tabele to przykład na to, jak może zostać zapisana w bazie danych mapa klucz-wartość.



Rysunek 3.29: Diagram fragmentu bazy danych odpowiadającego za uprawnienia w panelu administracyjnym



Rysunek 3.30: Diagram fragmentu bazy danych odpowiadającego za uprawnienia w panelu administracyjnym



Rysunek 3.31: Diagram fragmentu bazy danych skąd zbierane są atrybuty produktowe

Połączenie bazy relacyjnej z płaską strukturą noSQL Apache Solr

W tym podrozdziale został opisany mechanizm wyszukiwania od strony bazodanowej. Na diagramie 3.31 znajduje się diagram fragmentu relacyjnej bazy danych, w których znajdują się atrybuty produktowe, biorące udział w procesie wyszukiwania. Tabele na tym diagramie zostały opisane już w poprzednich podrozdziałach, jednak celem tego diagramu jest pokazanie, z jak wielu tabel pochodzą atrybuty produktowe i jak skomplikowane są połączenia między nimi. Niemożliwe jest aby system był w stanie za każdym wyszukaniem przez klienta wyciągać z relacyjnej bazy taką ilość danych. Byłoby to bardzo obciążające dla serwera. Dlatego podczas opisanego w poprzednich rozdziałach procesu indeksacji, który odbywa się w systemie raz na 15 minut, dane te są zbierane i produkty wraz ze wszystkimi ich atrybutami są wysyłane do nierelacyjnej bazy danych Apache Solr. Każdy produkt odpowiada jednemu dokumentowi w strukturze Solra. Takie rozwiązanie jest istotnie szybsze dlatego, że:

- unikamy klasycznych połączeń bazodanowych, które są zwykle transakcyjne, co jest kosztowne czasowo
- unikamy kwerend z JOIN'ami, które często sprowadzają się do iloczynu kartezjańskiego, który również jest bardzo kosztowny
- Solr korzysta z silnika Lucene, który indeksuje każde słowo w każdym dokumencie, dlatego kwerenda jednosłowna z jednym dokumentem daje złożoność O(1). W miarę wzrostu danych złożoność wyszukiwania jest liniowa, gdyż przy każdym wyszukaniu skanowany jest każdy dokument. W przypadku relacyjnej bazy danych przy wzroście ilości rekordów, drastycznie spada wydajność ze względu na konieczność użycia JOIN'ów.

System utrzymania bazy danych

Jako system utrzymania bazy danych został wybrany PostgreSQL. Głównym powodem jest to, że uchodzi on za lepiej spisujący się do dużych baz danych (które mogą pojawić się w platfromie e-commerce). Bywa używany w projektach zawierających hurtownie danych, i które są nastawione na bardzo szybki zapis i odczyt. Dodatkowo Hibernate[8] jest łatwo konfigurowalny dla baz opartych o PostgreSQL. Ważną zaletą tego systemu jest także spełnienie właściwości transakcji według ACID (Atomicity, Consistency, Isolation, Durability). Spełnienie tych właściwości gwarantuje brak zagubionych lub błędnych danych w razie awarii systemu.

Normalizacja bazy danych

Tabele zawarte w systemie są w trzeciej postaci normalnej. Wynika to z tego, że do tworzenia schematu bazy danych zostało wykorzystane mapowanie relacyjno-obiektowe, które same w sobie nie zapewnia 3NF, jednak sposób implementowania encji w standardzie JPA narzuca ją domyślnie. Każda tabela ma swój klucz główny oraz jest odzwierciedleniem pól w danej klasie - spełnienie 1NF, każda kolumna powinna być atomowa, przechowywać tylko jedną informację. Dodatkowo jedna encja w systemie oznacza jedną tabelę w bazie danych, co jest zgodne z 2NF, czyli wszystkie niekluczowe kolumny w tabeli muszą dotyczyć konkretnej klasy. 3NF jest również spełniona, ponieważ w żadnej z tabel nie ma niekluczowych informacji, które zależałyby od innych niekluczowych informacji. Wprowadzająca zamieszanie może być w tym przypadku tabela `order` z kolumną `full_amount`, która zależy od ilości i ceny produktów, jednak te informacje znajdują się w innych tabelach, przez co 3NF jest zachowana.

[8] Hibernate – framework w Javie do komunnikacji z bazą danych

Implementacja systemu

Opis technologii

W projektcie użyto wielu technologi oraz frameworków. Wszystkie oparte są o język Java, dokumentację można znaleźć w [2]. Interfejs użytkownika zaprojektowano przy użyciu szablonów opartych o projekt Start Bootstrap, którego opis można znaleźć na stronie [5].

Spring Framework/Spring Boot

Główną technologią jest framework Spring. Ułatwia pisanie aplikacji w Javie, w szczególności webowych, ze względu na dużą elastyczność i wzorzec projektowy Model-View-Controller, singleton oraz spełnia zasadę *Inversion of Control*. Aplikacja jest oparta o platformę Spring Boot. Jest to nakładka na framework Spring, który przy obecnych standardach stał się skomplikowany w konfiguracji. Spring Boot zapewnia autokonfigurację wielu komponentów systemu, łatwe dodawanie modułów Springowych (np. security lub web). Co najważniejsze posiada wbudowany kontener na aplikację internetową, przez co nie ma potrzeby konfigurowania osobnego kontenera (np. Tomcat) i umieszczania tam aplikacji. Znacznie przyśpiesza to dewelopment. Więcej informacji najduje się w dokumentacji [4] oraz w książce [9].

Hibernate/JPA 2.1

Java Persistence API jest interfejsem służącym do komunikacji aplikacji z bazą danych. Jego implementacją jest framework Hibernate. Korzystanie z tych standardów ułatwia komunikację z bazą danych i zapewnia wiele przydatnych funkcjonalności. Między innymi zapewnia również obsługę transakcji. W połączeniu ze Spring Data to potężne narzędzie, a jednocześnie jest proste w użytku. Więcej informacji o frameworku Hibernate i JPA w książce [6].

Spring Data

Spring Data ułatwia konfigurację połączeń między bazą danych, a serwerem. Do tego zapewnia automatyczne generowanie prostych kwerend bazodanowych, korzystając z zasady *query by convention*[1]. Jest to sposób pisania nazw metod w interfejsach, które framework jest w stanie zaimplementować. Całość powoduje odejście tradycyjnej warstwy DataAccessObject. Framework pozwala na samodzielne implementowanie trudniejszych kwerend, w spersonalizowany sposób.

Apache Solr

Oparta na silniku Lucene i uruchomiona na osobnym serwerze wyszukiwarka zapewnia odseparowanie najbardziej obciążonej części sklepu. Dzięki Solrowi w implementowanym frameworku

[1] query by convention - sposób generowania kwerend bazodanowych na podstawie nazw metod w interfejsach

jest obecna bardzo szybka wyszukiwarka z możliwością wyszukiwania pełnotekstowego, filtrowania, sortowania i zawężania wyników względem różnych pól produktu. Książka [8] i dokumentacja [1] opisują działanie i możliwości Solra. Do integracji serwera Apache Solr z implementowanym frameworkiem użyto biblioteki SolrJ [3].

Omówienie kodów źródłowych

Projekt składa się z 3986 linii kodu w języku Java, dlatego aby zachować zwięzłość i zrozumiałość w ramach omówienia kodów źródłowych zostanie przedstawiona tylko jedna funkcjonalność odnosząca się do konkretnych przypadków użycia zdefiniowanych w sekcji **Przypadki użycia** rozdziału **Projekt systemu**. Większość funkcjonalności jest napisana w analogiczny sposób ze względu na wzorzec architektoniczny model-view-controller, dlatego dokładna analiza jednej funkcjonalności wystarczy do zrozumienia metodologi wytwarzania kodu w projekcie.

Na rysunku 3.1 znajduje się use-case: *Zarządzanie encjami*. Związany jest przypadek z diagramu 3.6 *Wyświetlenie i manipulacja relacjami encji*. Te dwa przypadki wiążą się z konstrukcją dynamicznego formularza edycji dla danej encji. Dla lepszego zrozumienia warto przypomnieć diagramy związane z implementacją tej funkcjonalności. Na rysunku 3.12 przedstawiono diagram aktywności, zaś rysunek 3.20 przedstawia klasy użyte do implementacji. Dla przykładu działania kodu źródłowego zostanie przeanalizowana budowa formularza dla dowolnej encji.

Kod źródłowy 4.1: Przetworzenie zapytania żądania edycji encji `DynamicEntityController.java`.

```
@GetMapping("/entities/{entityCode}/{id}/edit")
public String getDynamicEntityForm(final Model model,
    @PathVariable(name = "entityCode") String entityCode,
    @PathVariable(name = "id") Long id) {
    model.addAttribute(ATTR_MENU_ITEMS , adminMenuGroupRepository.findAll());

    String className = resolveEntityCode(entityCode);
    if(permissionService.hasPermissionForOperation(className)) {
        DynamicForm dynamicForm = new DynamicForm();
        dynamicForm.setEntityClass(entityCode);
        List<RelationMetadata> relationalEntities = new ArrayList<>();

        dynamicEntityService.constructDynamicEntityForm(className, id,
                                dynamicForm, relationalEntities);

        model.addAttribute("relationalEntities", relationalEntities);
        model.addAttribute("dynamicForm", dynamicForm);
        model.addAttribute("entityName", entityCode);
        model.addAttribute("id", id);
    } else {
        model.addAttribute("authError", true);
    }
    return "dynamicEntityForm";
}
```

Kod źródłowy 4.1 przedstawia kontroler odpowiadający za przetworzenie żądania konstrukcji formularza edycji. Na przykład w przypadku produktu o `id = -105` mapowanie (url) będzie wyglądało następująco: `/entities/product/-105/edit`, gdzie `product` to kod encji `entityCode`. Na początku metody następuje dodanie do modelu menu, które jest widoczne na każdej stronie administracyjnej. Następnie w metodzie `resolveEntityCode`, kod encji produkt zostaje przetłumaczony

na pełną nazwę klasy z której pochodzi. W tym celu system sprawdza czy w bazie danych encja o takim kodzie należy do tych zdefiniowanych jako możliwe do obsłużenia w panelu administracyjnym. Kod źródłowy tej metody znajduje się na listingu 4.2.

Kod źródłowy 4.2: Zamiana kodu encji na nazwę klasy: `DynamicEntityController.java`.

```
private String resolveEntityCode(String entityCode){
    return adminMenuItemRepository.findByCode(entityCode)
            .map(AdminMenuItem::getClassName)
            .orElse("");
}
```

Mając nazwę klasy sprawdzane jest czy zalogowany użytkownik posiada prawa do edycji i wyświetlania danej klasy. Dzieje się to w metodzie `hasPermissionForOperation`. Zalogowany użytkownik pobierany jest z bazy danych, po czym jego wszystkie uprawnienia zostają zebrane w metodach `getAllChildrenPermissions` za pomocą rekurencyjnego przejścia po drzewie uprawnień, które należą do użytkownika administracyjnego. W tym przypadku każde uprawnienie, które jest mu przypisane odgrywa role korzenia (przechodzimy po wielu drzewach). Każde uprawnienie będące potomkiem korzenia, jest brane pod uwagę.

Kod źródłowy 4.3: Sprawdzenie uprawnień i algorytm przeszukiwania drzewa: `PermissionService.java`.

```
public boolean hasPermissionForOperation(String checkedClassName){
    AdminUser adminUser = adminUserRepository
                        .findByUsername(getLoggedInUsername()).get();
    return getAllChildPermissions(adminUser.getAdminPermissions())
            .stream()
            .map(AdminPermission::getClassName)
            .anyMatch(checkedClassName::equals);
}

private Set<AdminPermission> getAllChildPermissions(
Set<AdminPermission> adminPermissions) {
    final Set<AdminPermission> result = new HashSet<>();
    adminPermissions.forEach(adminPermission ->
        getAllChildPermissions(adminPermission, result));
    return result;
}

private void getAllChildPermissions(AdminPermission adminPermission,
                                    Set<AdminPermission> result) {
    adminPermission.getChildPermissions().forEach(child ->
        this.getAllChildPermissions(child, result));
    result.add(adminPermission);
}
```

Jeżeli algorytm przeszukiwania drzewa znajdzie nazwę klasy w finalnej kolekcji uprawnień administratora, kontroler z listingu 4.1 rozpoczyna budowę formularza. Formularz konstruowany jest w metodzie `constructDynamicEntityForm` przedstawionej na listingu 4.4. Korzystając z metody `getAllFieldsFromClass` wyciąga z klasy (i wszystkich możliwych nadklas) wszystkie pola, które są objęte interfejsem `@AdminVisible`, następnie dla każdego pola metoda `handleFieldInClass` obiera strategię jak skonstruować fragment formularza dla konkretnego pola.

Kod źródłowy 4.4: Konstrukcja formularza: `DynamicEntityService.java`.

```
public void constructDynamicEntityForm(String className, Long id,
            DynamicForm dynamicForm,
            List<RelationMetadata> relationalEntities) {
    List<Field> fields = getAllFieldsFromClass(className);
    AbstractEntity entity = dynamicEntityDao
                            .findAllPolimorficEntityWithId(className, id);
    for (Field field : fields) {
        for (Class<?> cls : ExtensionUtil.getSubclassesOf(className)) {
            handleFieldInClass(cls, field, entity, dynamicForm,
            relationalEntities);
        }
    }
}

private List<Field> getAllFieldsFromClass(String className) {
    return ExtensionUtil.getPolymorphicFielsdOf(className)
            .stream()
            .filter(field -> field.isAnnotationPresent(AdminVisible.class))
            .sorted(Comparator.comparingInt(field ->
                    field.getAnnotation(AdminVisible.class).order()))
            .collect(Collectors.toList());
}
```

Kod źródłowy 4.5 z metodą `handleClassField` przedstawia pracę mechanizmu determinującego strategię konstrukcji pola w formularzu. System rozpoznaje 3 typy pól: kolekcje jeden-do-wielu, kolekcje wiele-do-wielu oraz pola proste (string, bool, int). Dla każdego z nich zaimplementowano osobną metodę, która dodaje do formularza edycji aktualnie obsługiwane pole.

Kod źródłowy 4.5: Konstrukcja formularza: `ClassFieldHandler.java`.

```
private void handleFieldInClass(Class cls, Field field,
            AbstractEntity entity, DynamicForm dynamicForm,
            List<RelationMetadata> relationalEntities){
    Field classField = cls.getDeclaredField(field.getName());
    classField.setAccessible(true);

    if (Collection.class.isAssignableFrom(classField.getType()) &&
    classField.getAnnotation(OneToMany.class) != null) {
        handleOneToManyField(field,entity,relationalEntities);
    } else if (Collection.class.isAssignableFrom(classField.getType())
            && classField.getAnnotation(ManyToMany.class) != null) {
        handleManyToManyField(field,entity,relationalEntities);
    } else {
        handlePlainField(field,entity,classField,dynamicForm);
    }
}
```

Po zakończeniu budowy formularza kontroler z listingu 4.1 skonstruowany formularz oraz dodatkowe informacje do modelu (`model.addAttribute`). Następnie zwraca nazwę widoku (pliku HTML) z szablonem formularza encyjnego. W przypadku braku znalezienia odpowiedniego uprawnienia, użytkownikowi wyświetlany jest widok z brakiem uprawnień.

Instalacja i wdrożenie

W tym rozdziale należy omówić zawartość pakietu instalacyjnego oraz założenia co do środowiska, w którym realizowany system będzie instalowany. Należy przedstawić procedurę instalacji i wdrożenia systemu. Czynności instalacyjne powinny być szczegółowo rozpisane na kroki. Procedura wdrożenia powinna obejmować konfigurację platformy sprzętowej, OS (np. konfiguracje niezbędnych sterowników) oraz konfigurację wdrażanego systemu, m.in. tworzenia niezbędnych kont użytkowników. Procedura instalacji powinna prowadzić od stanu, w którym nie są zainstalowane żadne składniki systemu, do stanu w którym system jest gotowy do pracy i oczekuje na akcje typowego użytkownika.

Podsumowanie

W podsumowanie należy określić stan zakończonych prac projektowych i implementacyjnych. Zaznaczyć, które z zakładanych funkcjonalności systemu udało się zrealizować. Omówić aspekty pielęgnacji systemu w środowisku wdrożeniowym. Wskazać dalsze możliwe kierunki rozwoju systemu, np. dodawanie nowych komponentów realizujących nowe funkcje.

W podsumowaniu należy podkreślić nowatorskie rozwiązania zastosowane w projekcie i implementacji (niebanalne algorytmy, nowe technologie, itp.).

Bibliografia

[1] Apache solr technology. Web pages: https://lucene.apache.org/solr/guide/.

[2] Java technology. Web pages: http://www.oracle.com/technetwork/java/.

[3] Solrj guide. Web pages: https://lucene.apache.org/solr/guide/7˙5/using-solrj.html.

[4] Spring boot documentation. Web pages: https://docs.spring.io/.

[5] Start bootstrap project. Web pages: https://startbootstrap.com/.

[6] G. G. G. K. Christian Bauer. *Java Persistence with Hibernate, Second Edition*. Manning, 2015.

[7] G. McCluskey. Using java reflection. Web pages: https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html.

[8] T. G. T. Potter. *Solr in Action*. Manning, 2014.

[9] C. Walls. *Spring Boot in Action*. Manning, 2016.

Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.