POLITECHNIKA WARSZAWSKA

WYDZIAŁ ELEKTRYCZNY

Rozprawa doktorska

mgr inż. Wiktor Nowakowski

Semantyka translacyjna dla języka specyfikacji wymagań

Promotor
dr hab. inż. Michał Śmiałek, prof. P.W.

WARSZAWA 2023

Streszczenie

Praca przedstawia wyniki badań nad możliwością zwiększenia efektywności wytwarzania oprogramowania poprzez automatyczną generację kodu na podstawie wymagań funkcjonalnych. W opisanym podejściu wymagania w procesie wytwarzania oprogramowania traktowane są jako artefakty pierwszej kategorii, mające bezpośrednie powiązanie z kodem docelowego systemu. Formułowane są one w postaci precyzyjnych modeli, które z jednej strony zapewniają wysoki poziom komunikatywności, z drugiej – posiadają dobrze zdefiniowaną semantykę translacyjną, umożliwiającą ich bezpośrednie przekształcenie do kodu docelowej aplikacji dla konkretnej platformy technologicznej.

Pierwsza część pracy stanowi zarys problematyki badań. Przedstawione są tu najważniejsze koncepcje dążące do podniesienia poziomu abstrakcji na jakim tworzone są specyfikacje systemów oprogramowania oraz automatyzacji procesu ich przekształceń do docelowego kodu. Przedstawiona jest również motywacja do zwiększenia roli modeli wymagań w procesie wytwórczym a także środowisko ReDSeeDS, które posłużyło do praktycznej realizacji podejścia przedstawionego w pracy.

Zasadniczą część pracy stanowi opis semantyki translacyjnej dla języka specyfikacji wymagań RSL. Zdefiniowana jest ona w postaci zestawu reguł określających sposób mapowania konstrukcji języka źródłowego na konstrukcje języka Java stanowiące kompletną implementację logiki aplikacji oraz interfejsu użytkownika docelowej aplikacji. Definicję semantyki translacyjnej poprzedza opis składni konkretnej oraz abstrakcyjnej języka RSL. Dalsza część, natomiast, opisuje sposób implementacji zdefiniowanych reguł translacji w środowisku ReDSeeDS, z wykorzystaniem języka transformacji MOLA.

W celu dokonania oceny działania i efektywności opracowanego podejścia, wykonano studium przypadku dla przykładowej aplikacji a także przeprowadzono kontrolowany eksperyment z udziałem studentów.

Pracę kończy podsumowanie oraz wnioski z przeprowadzonych badań oraz wskazanie kierunków dalszego rozwoju uzyskanego rozwiązania.

Słowa kluczowe: *wytwarzanie oprogramowania sterowane modelami, inżynieria wymagań, metamodelowanie, semantyka translacyjna, transformacja modeli, generacja kodu, inżynieria języków oprogramowania*

Abstract

This work comprises the results of research investigating possibility of shortening and automating the path from requirements to code, thus making the software development process more efficient. In the presented approach requirements are treated as first-class artefacts in the software development process, playing a direct role in code generation. They are formulated as models which combine informality – facilitating comprehension, with precise translational semantics in order to enable their direct transformation into code for specific technology platform.

The first part of the thesis is an outline of research issues. It discusses existing approaches aiming at increasing the level of abstraction at which software specifications are created and automating the path from these specifications to the resulting code. The motivation to increase the role of requirements models in the software development process is also explained. This part concludes with the description of the ReDSeeDS environment that was used for the implementation of the approach presented in this thesis.

The main part of the thesis integrates various threads of work done by the author in regard to achieving the main objective – the translational semantics for the Requirements Specification Language (RSL). This consists in a set of rules that translate RSL constructs into equivalent Java constructs that fully implement the application logic and user interface of the target application. The definition of translational semantics is preceded by a description of the RSL syntax. This part also describes the way the semantic rules were implemented in the MOLA transformation language within the ReDSeeDS environment.

In order to evaluate the applicability of the presented approach and assess productivity gains compared to traditional software development process a case study as well as a controlled experiment with students were carried out.

The thesis concludes with the final discussion of the presented approach and some propositions of further research directions.

Keywords: *Model-Driven Software Development, Requirements Engineering, metamodeling, translational semantics, model transformations, code generation, Software Language Engineering*

Spis treści

1 Wprowadzenie

Jednym z pierwszych, a zarazem kluczowych, etapów procesu wytwarzania oprogramowania jest formułowanie wymagań. Niezależnie od stosowanej metodyki, właściwe przeprowadzenie tego etapu często decyduje o sukcesie bądź porażce projektu. Sukces rozumiany jest przy tym jako dostarczenie oprogramowania spełniającego faktyczne potrzeby zamawiającego oraz działającego zgodnie z oczekiwaniami użytkowników końcowych. Artefaktami, które mają bezpośredni wpływ na sposób funkcjonowania oraz cechy systemu są: projekt systemu oraz implementujący go kod. O ile zależność między tymi artefaktami jest zasadniczo jednoznaczna i bezpośrednia, o tyle ich zależność od wymagań pozostawia lukę semantyczną. W powszechnej praktyce, wymagania formułowane są w postaci niesformalizowanej – zazwyczaj w języku naturalnym, jako mniej lub bardziej ustrukturyzowany tekst. Ich przekształcenie w kod systemu jest procesem manualnym, pracochłonnym i obarczonym ryzykiem niewłaściwej interpretacji wymagań. W efekcie, pomimo ich znaczenia dla sukcesu projektu, wymagania mają jedynie pośredni wpływ na jego finalny efekt.

Rozwój inżynierii wymagań jako dyscypliny, w dużej części koncentruje się na psychologicznych aspektach pozyskiwania wymagań, technikach ich analizy czy notacjach pozwalających zwiększyć komunikatywność specyfikacji wymagań [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]. Dostrzegalny jest również dynamiczny rozwój narzędzi pozwalających w efektywny sposób zarządzać wymaganiami – ich treścią, zależnościami, wersjami, itp. Próbą zniwelowania luki semantycznej między wymaganiami a finalnym produktem są zwinne metody wytwarzania oprogramowania [11] [12] [13]. Kładą one duży nacisk na częstą i bezpośrednią komunikację pomiędzy przedstawicielem zamawiającego a zespołem wytwórczym, co ma przełożyć się na lepsze zrozumienie wymagań oraz możliwość wczesnej identyfikacji i rozstrzygnięcia potencjalnych niejednoznaczności.

Nadal, jednak, przekształcenie wymagań w docelowy kod, pozostaje procesem manualnym i pracochłonnym. Inżynieria wymagań, w małym stopniu koncentruje się na tworzeniu rozwiązań, które prowadziłyby do zmiany paradygmatu [14] [15], np. rozwoju notacji, metod i narzędzi pozwalających formułować wymagania w sposób umożliwiający ich automatyczne przekształcanie w kod.

Szansę na zmianę paradygmatu przyniosła koncepcja architektury sterowanej modelami (ang. Model-Driven Architecture – MDA) [16] [17] [18] [19] [20] [21]. Postuluje ona tworzenie specyfikacji oprogramowania na poziomie abstrakcji niezależnym od technologii wytwarzania

oprogramowania a następnie automatyczne przekształcanie takiej specyfikacji w artefakty (modele) zawierające więcej szczegółów technologicznych i – w rezultacie – kodu. Idea ta była sukcesywnie rozwijana w kierunku szerszego wykorzystania modeli i ich transformacji w procesie wytwarzania oprogramowania. Nurt ten określany jest nazwami: wytwarzanie oprogramowania sterowane modelami (ang. Model-Driven Software Development - MDSD) [22] [23], inżynieria oprogramowania sterowana modelami (ang. Model-Driven Software Engineering - MDSE) [24] czy też w szerszym kontekście – inżynieria sterowana modelami (ang. Model-Driven Engineering – MDE) [25] [26]. Jednym z obszarów tego nurtu są języki specyficzne dla dziedziny (ang. Domain-Specific Languages – DSL) [27] [28] [29] [30], które pozwalają na jednoetapową generację zaawansowanego kodu w oparciu o modele na poziomie opisu dziedziny problemu.

Rozwój tych obszarów przyniósł wiele praktycznych rozwiązań i narzędzi. Rozwiązania te, w przeważającej części, skupiają się jednak na modelach na poziomie projektowym, które mają mniej lub bardziej bezpośrednie powiązanie semantyczne z kodem. Wymagania w tym kontekście były pomijane ze względu na trudność w przekształceniu nieformalnych specyfikacji w artefakty techniczne.

Próbą uwzględnienia wymagań w podejściu MDSD jest względnie nowy nurt inżynierii wymagań sterowanej modelami (ang. Model-Driven Requirements Engineering – MDRE) [31]. Badania prowadzone w ramach tego nurtu koncentrują się na konstruowaniu precyzyjnych modeli wymagań, które umożliwiałyby ich automatyczną transformację do artefaktów niższego poziomu. Otwartym pozostaje pytanie czy możliwa i zasadne jest w pełni automatyczna transformacja precyzyjnych modeli wymagań na kompletny kod aplikacji realizującej te wymagania w konkretnych technologiach. Prowadzi nas to do postawienia następującej hipotezy:

> Możliwe jest znaczące zwiększenie efektywności wytwarzania oprogramowania poprzez automatyczną generację kodu aplikacji bezpośrednio z modelu wymagań, z użyciem jednoetapowej transformacji.

Niniejsza praca jest próbą zweryfikowania tej hipotezy. Prezentuje ona podejście, w którym wymagania traktowane są w procesie wytwarzania oprogramowania jako jednostki pierwszej kategorii (ang. first-class citizens) [32], mające bezpośrednie powiązanie z finalnym kodem. W podejściu tym, wymagania formułowane są w postaci precyzyjnych modeli [33], które zapewniają wysoki poziom komunikatywności, ale przede wszystkim posiadają dobrze

zdefiniowaną semantykę, która umożliwia ich translację bezpośrednio w kod docelowego systemu. Istotnym założeniem jest, aby generowany automatycznie kod był na tyle kompletny, aby można go było skompilować i uruchomić bez konieczności ręcznego uzupełniania. Musi on zatem implementować nie tylko sposób działania aplikacji wyrażoną wymaganiami, ale musi również realizować konkretne wzorce architektoniczne i projektowe oraz wszystkie aspekty technologiczne wymagane przez konkretne środowisko aplikacyjne (ang. application framework). Takie podejście daje szanse na istotne zwiększenie efektywności procesu wytwarzania oprogramowania.

W celu weryfikacji postawionej hipotezy, w ramach niniejszej pracy wykorzystano istniejące środowisko ReDSeeDS udostępniające język specyfikacji wymagań – Requirements Specification Language (RSL) – posiadający charakterystykę niezbędną do realizacji założeń tej pracy [34], a także efektywną infrastrukturę (narzędzia i proces) do definiowania i wykonywania transformacji modeli [35]. Głównym efektem pracy jest semantyka translacyjna dla języka RSL. Ma ona formę zbioru reguł definiujących sposób translacji konstrukcji języka RSL na kod w języku Java. Skupiono się w szczególności na konstrukcjach wyrażających logikę aplikacji w formie scenariuszy przypadków użycia wraz z powiązanymi elementami dziedzinowymi. Aby kod będący wynikiem translacji był kompletny, zdefiniowano abstrakcyjne środowisko translacyjne definiujące strukturę docelowego kodu zgodną ze wzorcem MVP. Aby zweryfikować zdefiniowaną semantykę, zaimplementowano reguły translacji w formie transformacji „RSL to Java”. Transformacja została napisana w języku MOLA, co umożliwiło jej uruchomienie w środowisku ReDSeeDS. Implementacja transformacji, której rezultatem jest kompilowalny i uruchamialny kod aplikacji, wymagało określenia konkretnego środowiska aplikacyjnego realizującego abstrakcyjne środowisko translacyjne. Następnie wykonano studium przypadku. Obejmowało on przygotowanie kompletnej specyfikacji wymagań w języku RSL dla realnego systemu, wykonanie zaimplementowanej transformacji oraz kompilację uzyskanego kodu i uruchomienie docelowej aplikacji. Pozwoliło to ocenić poprawność reguł translacji na poziomie ich definicji oraz implementacji. Na koniec wykonano eksperyment pozwalający ocenić wpływ opracowanego podejścia na efektywność procesu wytwarzania oprogramowania i walidacji postawionej hipotezy.

Niniejsza praca stanowi kontynuację prac badawczych autora, który brał udział w tworzeniu środowiska ReDSeeDS w ramach projektu o tej samej nazwie[1]. Autor miał istotny udział w opracowaniu języka RSL [36] [37] [38], reguł translacji języka RSL na modele architektoniczne oraz projektowe [39] [40] [41], narzędzia ReDSeeDS [42] [43] oraz bazującej na nim metody wytwarzania oprogramowania [44] [45]. Wyniki tych prac były następnie rozwijane przez autora w ramach projektu REMICS[2] [46]. W ramach tego projektu środowisko ReDSeeDS zostało rozbudowane o mechanizm ekstrakcji logiki aplikacji z istniejących systemów w celu zautomatyzowanej migracji do nowoczesnych technologii chmurowych. Częścią tego mechanizmu była transformacja umożliwiająca przekształcenie wyekstrahowanej logiki, wyrażonej w języku RSL, do częściowego kodu docelowej aplikacji [47] [48] [49] [50] [51]. Uzyskane rezultaty zostały podsumowane w książce Śmiałka i Nowakowskiego [52]. Niniejsza praca stanowi istotne rozwinięcie i doprecyzowanie dotychczasowych rezultatów. Przedstawiona w pracy semantyka translacyjna wraz z towarzyszącym jej środowiskiem translacyjnym, została zdefiniowana od podstaw a następnie zaimplementowana w języku MOLA z uwzględnieniem spostrzeżeń oraz wniosków zebranych w ramach dotychczasowych prac autora. Finalnym efektem jest działająca transformacja „RSL to Java" umożliwiająca generowania kompletnego kodu w warstwie logiki aplikacji oraz interfejsu użytkownika z wykorzystaniem konkretnego środowiska aplikacyjnego. Uzyskane rezultaty są komplementarne względem rezultatów opisanych w pracy Rybińskiego [53] – dzięki odpowiedniemu rozszerzeniu języka RSL, możliwa jest automatyczna generacja kodu w warstwie logiki biznesowej. Połączenie rezultatów obu tych prac daje szansę na stworzenie rozwiązania pozwalającego przekształcać modele wymagań w kod aplikacji implementujący wszystkie warstwy typowej architektury trójwarstwowej.

Wraz z niniejszym wprowadzeniem, praca składa się z dziewięciu rozdziałów. Rozdział 2 stanowi zarys problematyki poruszanej w pracy. Podaje on kontekst oraz motywację dla przeprowadzonych badań, skupiając się na roli modeli na rożnych poziomach abstrakcji w procesie wytwarzania oprogramowania, w szczególności roli modeli wymagań. Rozdział ten

[1] Requirements-Driven Software Development System (ReDSeeDS) – projekt współfinansowany przez Unię Europejską w ramach 6 Programu Ramowego, kontrakt nr IST-2006-33596.

[2] Reuse and Migration of Legacy Applications to Interoperable Cloud Services (REMICS) – project współfinansowany przez Unię Europejską w ramach 7 Programu Ramowego (kontrakt nr ICT-257793) oraz Ministerstwo Nauki i Szkolnictwa Wyższego (kontrakt nr 2161/7. PR/2011/2).

opisuje również środowisko ReDSeeDS stanowiące punkt wyjścia do przeprowadzonych badań. Rozdział 3 przedstawia składnię konkretną języka RSL oraz jego semantykę z punktu widzenia osoby tworzącej specyfikację wymagań. Jest to nieformalny sposób przedstawienia semantyki języka. Składnia abstrakcyjna języka RSL – w postaci metamodelu – jest opisana w rozdziale 4. Dodatkowo, rozdział ten opisuje wybrane elementy metamodelu języka SCL, który dostarcza mechanizmy zapewniające spójność i integralność między źródłowym modelem wymagań a modelem docelowym będącym wynikiem transformacji. Rozdziały 5 oraz 6 stanowią zasadniczą część pracy. Pierwszy z nich zawiera definicję semantyki translacyjnej dla języka RSL, poprzedzony opisem środowiska translacyjnego. Drugi stanowi opis implementacji transformacji „RSL to Java" w języku MOLA dla konkretnego środowiska aplikacyjnego. Rozdział 7 omawia przeprowadzone studium przypadku. Prezentuje on źródłową specyfikację wymagań dla przykładowego systemu a także omawia ogólną strukturę wygenerowanego kodu oraz – bardziej szczegółowo – wybrane jego fragmenty. Wreszcie rozdział 8 opisuje wyniki ewaluacji rozwiązania przeprowadzonej w formie kontrolowanego eksperymentu. Pracę kończy rozdział 9 zawierający podsumowanie oraz wnioski z przeprowadzonych badań.

2 Zarys problematyki badań

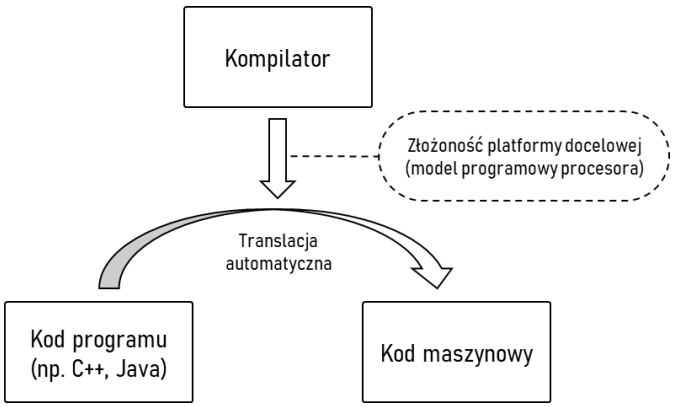
Rozdział ten stanowi zarys problematyki badań zaprezentowanych w niniejszej pracy. Przedstawione są tu najważniejsze podejścia, które powstały z myślą o zwiększeniu efektywności tworzenia oprogramowania poprzez usprawnienie procesu przekształcania wymagań wobec tworzonego systemu w jego finalny kod. Cechą wspólną tych koncepcji jest dążenie do podniesienia komunikatywności oraz poziomu abstrakcji na jakim tworzone są specyfikacje systemu w połączeniu z automatyzacją procesu przekształceń tych specyfikacji do docelowego kodu. Rozdział wyjaśnia również motywację do zwiększania roli modeli wymagań w zautomatyzowanym procesie wytwórczym szerokiej klasy systemów biznesowych. Przedstawione jest również środowisko ReDSeeDS będące realizacją koncepcji wytwarzania oprogramowania sterowanego modelami, które w centrum procesu wytwórczego stawia precyzyjne modele wymagań. Środowisko to posłużyło do weryfikacji hipotezy badawczej postawionej w niniejszej pracy.

2.1 Modelowanie wymagań

Wyobraźmy sobie scenariusz, w którym wymagania wobec systemu oprogramowania pozyskane od zamawiającego oraz użytkowników końcowych, przeanalizowane i spisane w postaci specyfikacji wymagań, są w pełni automatycznie przekształcane w finalny kod systemu, gotowy do skompilowania i uruchomienia [54] [55] [49] [56]. Aby zrealizować taki scenariusz, niezbędny byłby mechanizm, który z jednej strony potrafiłby interpretować wymagania zapisane w języku naturalnym, rozstrzygając ewentualne niejednoznaczności, z drugiej strony – potrafiłby połączyć wiedzę wyekstrahowaną z wymagań z wiedzą na temat aspektów technologicznych wynikowej aplikacji, m.in. wzorców architektonicznych i projektowych, stosu technologicznego, docelowego środowiska uruchomieniowego, czy też konwencji przyjętych w docelowym języku programowania.

Podobny scenariusz, lecz na niższym poziomie abstrakcji, przedstawia Rysunek 1. Jest to dobrze znany mechanizm kompilacji [57] przekształcający kod źródłowy programu napisanego w języku trzeciej generacji (np. C++, Java, Rust) w kod maszynowy (lub kod pośredni), który może być wykonywany na określonej platformie systemowo-sprzętowej. W uproszczeniu, kompilator analizuje kod źródłowy pod kątem syntaktycznym i semantycznym a następnie przekształca go w kod maszynowy, niejako „wstrzykując” do niego wiedzę na temat modelu programowego danej platformy (m.in. listy rozkazów procesora, typów danych, trybów adresowania pamięci, dostępnych rejestrów, zasad obsługi wyjątków i przerwań). Dzięki temu,

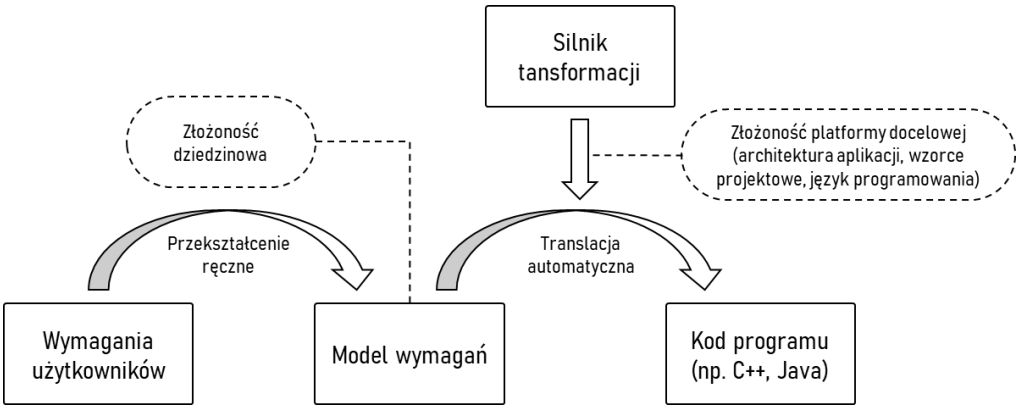
języki trzeciej generacji znacząco zwiększają efektywność tworzenia oprogramowania, gdyż pozwalają programistom skupić się na aspektach algorytmicznych, abstrahując od złożoności technologicznej platformy uruchomieniowej. Dodatkową zaletą jest łatwe przenoszenie kodu źródłowego pomiędzy platformami – wystarczy użyć kompilatora dla danej platformy.



Rysunek 1 – Translacja kodu w języku programowania trzeciej generacji w kod maszynowy

Proces kompilacji jest bezpośredni i deterministyczny, gdyż języki trzeciej generacji, mimo że charakteryzują się wyższym poziomem abstrakcji niż języki maszynowe czy języki assemblera, nadal są językami formalnymi o bardzo precyzyjnej składni i semantyce. Język naturalny jest, dla odmiany, językiem nieformalnym. Wymagania zapisane z użyciem języka naturalnego mogą być, w zależności od zdolności lingwistycznych autora, mniej lub bardziej precyzyjne i jednoznaczne. Bezpośrednie, deterministyczne przekształcenie takiej specyfikacji wymagań w kod docelowej aplikacji wydaje się mało realistyczne.

Scenariusz taki stałby się bardziej realistyczny, jeśli wymagania byłyby zapisane w języku formalnym, posiadającym precyzyjną składnię i semantykę. Rysunek 2 przedstawia schematycznie proces, w którym wymagania pozyskane od użytkowników są formułowane w postaci precyzyjnego modelu wymagań. Model ten opisuje pożądaną funkcjonalność systemu (logikę aplikacji), przetwarzane dane oraz elementy interfejsu użytkownika. Jest on zatem nośnikiem informacji o złożoności dziedzinowej jaką ma odzwierciedlać docelowy system.



Rysunek 2 – Translacja modelu wymagań w kod docelowej aplikacji

Istotne jest, aby język w którym wyrażony jest model wymagań łączył dwie cechy. Z jednej strony powinien być na tyle wysokopoziomowy, aby być czytelnym dla odbiorców nietechnicznych (m.in. dla zamawiającego, użytkowników czy analityków biznesowych), aby umożliwić im tworzenie i weryfikację modeli wymagań. Z drugiej strony, powinien być na tyle precyzyjny, aby możliwe było jego automatyczne przetwarzanie przez silnik transformacji w celu wygenerowania kodu wynikowego. Silnik transformacji w toku translacji „wstrzykuje" do wynikowego kodu wiedzę dotyczącą złożoności technologicznej docelowej platformy, która ma istotny wpływ na kształt i kompletność tego kodu, aby możliwe było jego skompilowanie i uruchomienie. Analogicznie jak w przypadku kompilacji, można wyobrazić sobie sytuację, w której ten sam model wymagań przekształcany jest na kod dla różnych platform docelowych, np. aplikację internetową oraz mobilną. Scenariusz ten będziemy dalej rozpatrywać w kontekście postawionej hipotezy badawczej.

Motywacją do realizacji powyższego scenariusza jest dążenie do skupienia większości uwagi na zasadniczej złożoności problemu przy jednoczesnej redukcji złożoności incydentalnej. Podział złożoności na zasadniczą (ang. essential complexity) oraz incydentalną (ang. accidental complexity) zaproponował Frederick Brooks [58] [59]. Zasugerował on, że większość problemów napotykanych w procesie wytwarzania oprogramowania wynika ze złożoności dziedziny problemu (złożoności zasadniczej): danych i ich powiązań, sposobu przetwarzania tych danych oraz logiki działania systemu, który ma odzwierciedlać daną dziedzinę. Reprezentacja tych elementów w konkretnym języku programowania stanowi, natomiast, o złożoności incydentalnej procesu wytwórczego. Co za tym idzie, istotne zwiększenie efektywności wytwarzania oprogramowania mogą przynieść przede wszystkim rozwiązania skupiające się na złożoności zasadniczej i ukrywające jak najwięcej złożoności incydentalnej.

W naszym scenariuszu (Rysunek 2), złożoność zasadnicza jest wyrażona modelem wymagań, a złożoność incydentalną skrywa silnik transformacji. Oznacza to istotne podniesienie poziomu abstrakcji na jakim definiowane jest oprogramowanie względem języków trzeciej generacji. Odbywa się to jednak pewnym kosztem.

Języki trzeciej generacji pozwalają wyrażać konstrukcje programistyczne na dużo wyższym poziomie niż języki maszynowe czy języki assemblera. Znacząco ułatwia i przyspiesza to proces tworzenia oprogramowania. Odbywa się to kosztem zmniejszenia stopnia swobody korzystania z natywnych mechanizmów danej platformy uruchomieniowej, co skutkuje mniej optymalnym kodem. Korzyści z podniesienia poziomu abstrakcji w przypadku języków trzeciej generacji dalece przewyższają jednak związane z tym koszty, zwłaszcza w przypadku aplikacji biznesowych, gdzie czas i budżet przeznaczony na ich wytworzenie są zazwyczaj kluczowe.

Analogicznie, w rozpatrywanym scenariuszu, silnik transformacji zwalnia twórców modelu wymagań z podejmowania nie tylko decyzji projektowych dotyczących platformy uruchomieniowej, lecz także decyzji projektowych dotyczących całego stosu technologicznego aplikacji (np. architektury logicznej aplikacji, sposobu utrwalania danych, sposobu przygotowania danych do prezentacji za pomocą interfejsu użytkownika, przekazywania kontroli pomiędzy komponentami aplikacji, itp.), czyli większości decyzji, jakie trzeba podjąć tworząc oprogramowanie w języku trzeciej generacji. Takie zmniejszenie stopnia swobody stanowi pewien koszt „programowania” na poziomie wymagań [60] [61] [62].

Do realizacji takiego scenariusza niezbędne są trzy podstawowe elementy:

- źródłowy język modelowania wymagań posiadający formalną składnię,
- odpowiednia semantyka translacyjna, która definiuje jak przekształcać konstrukcje w języku źródłowym na język docelowy (język programowania),
- środowisko transformacji modeli umożliwiające zaimplementowanie reguł translacji oraz ich wykonywanie.

Należy zauważyć, że zdefiniowanie oraz implementacja semantyki translacyjnej dla języka specyfikacji wymagań wiąże się ze znaczącym nakładem pracy. Zakładając jednak, że transformacja taka jest uniwersalna i może być wielokrotnie stosowana dla wielu różnych specyfikacji wymagań (dla systemów z różnych dziedzin biznesowych), wysiłek ten jest uzasadniony – analogicznie jak w przypadku kompilatorów, które tworzone są raz dla danej platformy docelowej.

W kontekście rozpatrywanego scenariusza, istotna wydaje się odpowiedź na pytanie: dlaczego formułować wymagania akurat w postaci modeli? Istnieją ku temu dwa główne powody. Po pierwsze, modele cechują się wysoką komunikatywnością. Zwłaszcza modele posiadające graficzną składnię konkretną mogą uczynić specyfikację wymagań dużo łatwiejszą do przyswojenia i stosowania przez ludzi nietechnicznych biorących udział w procesie formułowania i weryfikacji wymagań (m.in. zamawiającego, użytkowników końcowych, analityków biznesowych) [63] [64] [65] [66]. Po drugie, modele wymagań, zachowując swoją komunikatywność, mogą być zdefiniowane w sposób formalny za pomocą metamodelu. Dzięki precyzyjnej składni abstrakcyjnej możliwe jest ich automatyczne przetwarzanie z użyciem języków transformacji modeli [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77]. Poszczególnym elementom składniowym można, natomiast, nadać precyzyjną semantykę niezbędną do zdefiniowania reguł translacji do języków niższego poziomu – modeli projektowych (np. w języku UML [78] [79]) oraz kodu.

2.2 Wytwarzanie oprogramowania sterowane modelami

Typowy proces wytwarzania oprogramowania jest w dużej mierze procesem manualnym. Jeżeli wykorzystywane są w nim modele, pełnią one zazwyczaj rolę dokumentacyjną. Używane są też niekiedy w dosyć ograniczonym zakresie do generowania fragmentów kodu, np. w oparciu o modele klas w języku UML. Znaczące zwiększenie roli modeli w procesie wytwórczym nastało wraz z pojawieniem się podejścia Model-Driven Software Development (MDSD) czyli wytwarzania oprogramowania sterowanego modelami. W literaturze określane również terminem Model-Driven Software Engineering (MDSE) lub Model-Driven Development [23] [24] [22] [80].

Po raz pierwszy podejście takie zaproponowała organizacja Object Management Group (OMG)[3] pod nazwą Model-Driven Architecture (MDA) [81] [18] [19] [16] [17] [82] [20] [21]. W podejściu tym wszystkie pośrednie artefakty w procesie wytwarzania oprogramowania mają formę modeli, przy czym modele na niższym poziomie abstrakcji powstają w wyniku transformacji modeli na wyższym poziomie abstrakcji. Przykładem może być generowanie modeli interfejsu użytkownika na podstawie modelu dziedziny [83]. MDA definiuje trzy

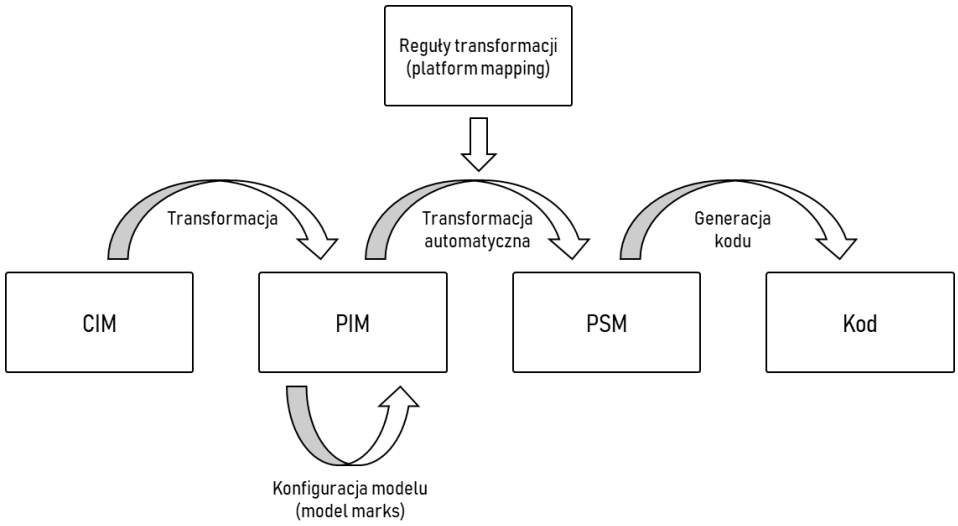
[3] https://www.omg.org

poziomy abstrakcji dla modeli. Poniżej przytoczone są definicje tych poziomów, zaczerpnięte z oficjalnego przewodnika po MDA [82] (tłum. własne).

- Computation-Independent Model (CIM) – „*CIM jest widokiem systemu z perspektywy niezależnej od środków informatyki. CIM nie ukazuje szczegółów struktury systemów. CIM bywa nazywany modelem dziedziny, w specyfikacji którego używana jest terminologia znana specjalistom w danej dziedzinie.*"
- Platform-Independent Model (PIM) – „*PIM jest widokiem systemu z perspektywy niezalaeżnej od platformy. PIM wykazuje określony stopień niezależności od platformy, aby możliwe było jego użycie dla wielu różnych platform podobnego typu.*"
- Platform-Specific Model (PSM) – „*PSM jest widokiem systemu z perspektywy konkretnej platformy. PSM łączy specyfikacje systemu na poziomie PIM ze szczegółami określającymi w jaki sposób dany system używa konkretnej platformy.*"

Rysunek 3 przedstawia proces tworzenia oprogramowania zgody z koncepcją MDA. Finalny kod generowany jest poprzez kolejne transformacje modeli na zdefiniowanych poziomach abstrakcji. Najwięcej uwagi MDA poświęca automatycznej transformacji pomiędzy modelami PIM i PSM. O tym, jak ma wyglądać takie przekształcenie, decydują reguły transformacji (ang. platform mappings). Co więcej, reguły te mogą podlegać konfiguracji poprzez dodanie do modelu źródłowego określonych znaczników (ang. model marks) sterujących wykonaniem reguł. Model PSM może posłużyć w dalszym etapie bezpośrednio do generacji kodu docelowego systemu dla danej platformy.

Podejście MDA niewiele uwagi poświęca natomiast modelom w warstwie CIM. Pomija również kwestie automatyzacji transformacji tych modeli do warstwy PIM. MDA dąży tym samym do redukcji złożoności incydentalnej i abstrahuje od kwestii złożoności zasadniczej.



Rysunek 3 – Wytwarzanie oprogramowania w podejściu Model-Driven Architecture (MDA)

Istnieje szereg narzędzi, które w większym lub mniejszym stopniu realizują paradygmat MDA. Większość z nich to narzędzia do modelowania obiektowego, jak np. Enterprise Architect[4] czy Modelio[5], które umożliwiają generację szkieletu kodu w różnych językach programowania na podstawie modeli klas. Niektóre rozwiązania idą o krok dalej, umożliwiając generowanie struktury całego systemu na podstawie szeregu diagramów w języku UML. Przykładem takiego narzędzia jest AndroMDA[6]. Najbardziej zaawansowane narzędzia, takie jak Bluage[7], czy Integranova[8], umożliwiają generowanie w pełni funkcjonalnego kodu na podstawie modeli UML uzupełnionych o adnotacje (model marks) w języku OCL [84] lub OASIS [85]. Modelami źródłowymi nie są tu jednak wymagania lecz modele projektowe. Nawet, jeśli są one niezależne od konkretnej platformy docelowej, muszą zawierać określone decyzje projektowe. Są zatem bardziej szczegółowe i bardziej techniczne, niż specyfikacja wymagań, dlatego posługiwanie się nimi przez osoby nietechniczne jest dosyć ograniczone. Charakterystyka istniejących rozwiązań MDA sprawia, że nie mają one zastosowania w realizacji scenariusza przedstawionego na Rysunek 2.

O ile podejście MDA nie zyskało szerokiej akceptacji, o tyle zaproponowana tam idea transformacji modeli jest realizowana w szerszym kontekście w podejściu MDSD. Podejście to

[4] https://www.sparxsystems.com

[5] https://www.modelio.org

[6] http://www.andromda.org

[7] https://bluage.com

[8] https://integranova.com

zakłada budowę dowolnego łańcucha modeli w całym cyklu wytwórczym oprogramowania (m.in. modeli specyficznych dziedzinowo, bez ograniczenia do języka UML, jak w przypadku MDA), na podstawie których generowany jest finalny kod oprogramowania oraz ewentualne inne artefakty [80] [86]. MDSD jest praktykowane z użyciem różnych dostępnych narzędzi. Najbardziej rozpowszechnione wydają się być narzędzia do modelowania typu CASE[9], z których większość posiada – wbudowane lub dostępne w postaci dodatkowych modułów – silniki transformacji. Silniki te są z reguły specyficzne dla danego narzędzia i najczęściej pozwalają na transformację modeli wyrażonych w UML – najbardziej rozpowszechnionym języku modelowania oprogramowania. Najczęściej też silniki te oferują zestaw predefiniowanych transformacji z możliwością konfiguracji określonego zestawu parametrów. Istnieją również kompletne środowiska transformacji modeli oparte na ustandaryzowanych językach transformacji modeli [87]. Są one niezależne od narzędzi do modelowania, lecz umożliwiają wymianę modeli z tymi narzędziami poprzez dostęp do ich repozytoriów lub standardowe formaty danych (np. XMI[10]).

Pełny proces wytwórczy w podejściu MDSD obejmuje tworzenie zasadniczo tych samych artefaktów, co w procesie tradycyjnym: specyfikacji wymagań, modelu architektury, projektu szczegółowego i kodu. W praktyce, istniejące narzędzia (mimo szerszej ich dostępności niż narzędzi MDA) wspierają automatyczną transformację modeli na niższym poziomie abstrakcji niż wymagania. Badania empiryczne [88] [89] potwierdzają wzrost efektywności procesu wytwarzania oprogramowania w podejściu MDSD, jednak ich szersza akceptacja napotyka szereg wyzwań. Zasadniczym wyzwaniem jest budowa kompleksowego środowiska MDE wspierającego cały proces wytwórczy od wymagań do kodu, co wymaga integracji szeregu narzędzi. Dodatkowo, efektywne używanie tych narzędzi obarczone jest wysokim progiem wejścia, zwłaszcza dla mniej technicznych uczestników projektów. Zasadnym wydaje się więc zaproponowanie rozwiązania, które istotnie skróciłoby ścieżkę od wymagań do kodu jak w scenariuszu przedstawionym wcześniej (Rysunek 2).

2.3 Metody zwinne

Zwinne metody wytwarzania oprogramowania [13] [90] [91] [92] [93] [94] [11] [12] dążą do zniwelowania luki semantycznej pomiędzy wymaganiami a finalnym produktem, skracając tym

[9] Computer-Aided Software Engineering

[10] XML Metadata Interchange

samym ścieżkę od wymagań do kodu. Wymagania w podejściach zwinnych tworzone są zazwyczaj z użyciem prostych notacji jak np. historyjki użytkownika (ang. user stories), wzbogaconych niekiedy o nieformalne diagramy procesów czy dziedziny problemu. Ma to na celu jedynie uchwycenie zakresu systemu, a dodatkowe szczegóły opisujące sposób jego funkcjonowania oraz cechy jakościowe są stopniowo ustalane między zamawiającym a zespołem deweloperskim w kolejnych iteracjach i zapisywane nieformalnie, np. w postaci tzw. kryteriów akceptacji. Krótkie iteracje, w których wymagania przekształcane są w działające fragmenty systemu, dają zamawiającemu szansę na częstą weryfikację oraz ewentualną adaptację wymagań. Aby możliwe było przejście pełnej ścieżki od wymagań do kodu w każdej iteracji, projektowanie systemu *a priori*, typowe dla tradycyjnego cyklu wytwórczego, jest mocno zredukowane – większość decyzji architektonicznych i projektowych podejmowanych jest bezpośrednio w trakcie tworzenia kodu. Decyzje te są często optymalne lokalnie i nie są wystarczająco udokumentowane. Znacząco zwiększa to szybkość dostarczania finalnego produktu ale skutkuje powstawaniem długu technologicznego. Zniwelowanie tego długu w fazie dalszego rozwoju i utrzymania gotowego systemu, chociażby ze względu na brak wystarczającej dokumentacji projektowej, okazuje się działaniem kosztownym [95] [96].

Wytwarzanie oprogramowania sterowane modelami wydaje się odpowiadać zarówno na potrzebę szybkiego wytwarzania oprogramowania (dzięki automatycznym transformacjom), jak i na potrzebę zapewnienia odpowiedniej dokumentacji w postaci modeli systemu na różnych poziomach abstrakcji. Początkowy nakład pracy związany z wytworzeniem wystarczająco precyzyjnego modelu źródłowego, który może być następnie w sposób zautomatyzowany przekształcony do kodu, wydaje się mieć w tym przypadku uzasadnienie. Zysk może być tym większy, im więcej złożoności incydentalnej ukrywa model źródłowy, np. model na poziomie wymagań.

2.4 Języki DSL

Dążenie do skrócenia procesu wytwórczego poprzez istotne ukrycie złożoności incydentalnej, doprowadziło do powstania tzw. języków specyficznych dla dziedziny (Domain-Specific Languages – DSL) [27] [28] [29] [30]. Zadaniem tych języków jest umożliwienie precyzyjnego wyrażania zagadnień związanych z konkretną dziedziną problemu (złożoność dziedzinowa) oraz umożliwienie automatycznego przekształcenia takich specyfikacji w docelowy kod w konwencjonalnych językach programowania lub bezpośredniego wykonania takiej specyfikacji w odpowiednim środowisku uruchomieniowym. Popularnym przykładem języka

DSL jest język SQL[11] przeznaczony do tworzenia i przeszukiwania relacyjnych zbiorów danych w znacząco prostszy sposób niż w przypadku języków programowania ogólnego przeznaczenia. W swojej podstawowej formie nie ma on jednak zastosowania do jakichkolwiek innych zadań, do których można zastosować języki konwencjonalne, np. tworzenie graficznych interfejsów użytkownika. Język SQL jest przykładem języka programowania specyficznego dla dziedziny, ale języki DSL mogą również służyć do modelowania konkretnej dziedziny. Mówimy wtedy o językach modelowania specyficznych dla dziedziny (Domain-Specific Modeling Languages – DSML) będących podzbiorem języków DSL.

Język DSL musi być wystarczająco ekspresywny i komunikatywny, aby mógł być z łatwością używany przez przedstawicieli danej dziedziny do opisania danego problemu, a jednocześnie musi być językiem formalnym, aby umożliwić jego automatyczną transformację do kodu. Taka charakterystyka sprawia, że są one tworzone dla konkretnej dziedziny lub są rozwijane razem z tą dziedziną. Aby języki DSL spełniały swoją rolę, powinny umożliwiać generowanie kompletnego kodu dla danego zagadnienia, implementującego np. zarówno logikę dziedzinową oraz logikę aplikacji dla danego problemu.

Podejście do wytwarzania oprogramowania w oparciu o języki DSL może skutkować znaczącym wzrostem produktywności [97]. Ma to miejsce zwłaszcza w przypadku potrzeby wytwarzania całej rodziny systemów (wariantów) w pewnej wąskiej dziedzinie, np. dla różnych klientów czy dla różnych segmentów rynku. Mówimy wtedy o tzw. liniach produkcyjnych oprogramowania czyli Software Product Lines (SPL) [98] [99] [100] [101] [102]. Początkowy nakład na opracowanie odpowiedniego języka lub języków DSL, reguł transformacji do kodu oraz przygotowanie środowiska narzędziowego, zwraca się dzięki efektowi skali. Przykłady udanych realizacji SPL można znaleźć m.in. w dziedzinie telekomunikacji, elektroniki konsumenckiej, przemyśle motoryzacyjnym czy w diagnostyce medycznej [30].

2.5 Inżynieria wymagań sterowana modelami

W przypadku wytwarzania systemów dla wielu różnych dziedzin, wzrost efektywności związany z zastosowaniem języków DSL jest raczej trudny do osiągnięcia ze względu na konieczność zdefiniowania osobnych języków specyficznych dla tych dziedzin. Znaczący wzrost produktywności w takim przypadku, mógłby przynieść język ogólnego przeznaczenia,

[11] Structured Query Language

który miałby zastosowanie do wielu dziedzin typowych dla aplikacji biznesowych, ukrywałby – podobnie jak języki DSL – złożoność incydentalną oraz umożliwiałby definiowanie precyzyjnych reguł translacji do artefaktów niższego poziomu i finalnego kodu dla różnych platform docelowych. Rozpatrując taki język w kontekście wytwarzania szerokiej klasy systemów biznesowych, wymagających intensywnej interakcji z użytkownikami końcowymi, powinien on umożliwiać tworzenie specyfikacji na poziomie wymagań z uwzględnieniem następujących elementów:

- wymagań funkcjonalnych opisujących logikę działania aplikacji, realizującą określone procesy biznesowe;
- pojęć z dziedziny problemu reprezentujących obiekty biznesowe, które system miałby przetwarzać oraz sposobu ich przetwarzania (logika dziedzinowa);
- pojęć z dziedziny aplikacji określających sposób prezentacji danych zawartych w obiektach biznesowych poprzez interfejs użytkownika;
- wymagań niefunkcjonalnych określających cechy jakościowe systemu, które implikowałyby określone rozwiązania projektowe.

Język o powyższej charakterystyce wraz z odpowiednim środowiskiem do tworzenia i wykonywania transformacji specyfikacji w tym języku są elementami niezbędnymi do pełnej realizacji scenariusza rozpatrywanego w rozdziale 2.1 (Rysunek 2), gdzie w wyniku automatycznej transformacji modelu wymagań otrzymalibyśmy kompletny kod aplikacji docelowej.

Podejście takie wpisuje się w nurt inżynierii wymagań sterowanej modelami – Model-Driven Requirements Engineering (MDRE) [103] [104] [105] [106] [107], będącej połączeniem dwóch dziedzin, które wcześniej rozwijały się niezależnie, tj. inżynierii wymagań (Requirements Engineering, w skrócie: RE) oraz MDSD. MDRE uściśla kontekst i proponuje generację artefaktów projektowych i kodu na podstawie modeli wymagań.

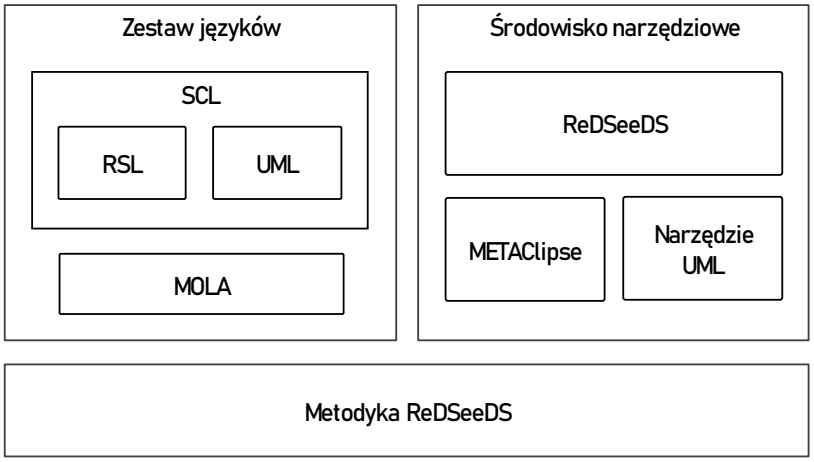
2.6 Środowisko ReDSeeDS

Jednym z rozwiązań realizujących ideę MDRE jest środowisko ReDSeeDS (Requirements-Driven Software Development System)[12] [108] [109] [110] [43] [45]. Jest to otwarte środowisko dostarczające notację, narzędzia oraz proces umożliwiające tworzenie modeli wymagań oraz transformacji przekształcających wymagania w artefakty niższego poziomu. Wybór środowiska ReDSeeDS podyktowany był m.in. faktem, że dostarcza ono języka specyfikacji wymagań ogólnego przeznaczenia – Requirements Specification Language (RSL) [36] [48], który posiada wszystkie cechy wymienione w podrozdziale 2.5. Jego użyteczność została pozytywnie zwalidowana przez Mukasa et al. [34]. Ponadto, stadium wykonalności przeprowadzone przez Jedlitchka et. al. [35] wskazuje na istotny potencjał środowiska ReDSeeDS w zastosowaniu do wytwarzania oprogramowania sterowanego modelami, w tym do realizacji pełnej ścieżki od wymagań do kodu. Język RSL wraz z całym środowiskiem stanowią więc niezbędną infrastrukturę do realizacji rozpatrywanego scenariusza i postawionej hipotezy badawczej. Kolejny rozdział (2.6.1) zawiera bardziej szczegółowy opis środowiska ReDSeeDS.

2.6.1 Czym jest ReDSeeDS

ReDSeeDS jest spójnym środowiskiem wspierającym wytwarzanie oprogramowania sterowane modelami wymagań oraz wielokrotne używanie gotowych artefaktów. Środowisko to umożliwia tworzenie precyzyjnych modeli wymagań, które następnie mogą być przekształcane w wyniku automatycznych transformacji w artefakty niższego poziomu: modele architektoniczne, modele projektowe oraz kod. Na środowisko ReDSeeDS składają się trzy główne komponenty (Rysunek 4): zestaw języków do modelowania artefaktów oraz ich transformacji, środowisko narzędziowe umożliwiające edycję modeli i transformacji oraz metodyka definiująca proces wytwórczy z użyciem języków i narzędzi.

[12] Środowisko ReDSeeDS powstało w ramach projektu współfinansowanego przez Unię Europejską w ramach 6 Programu Ramowego, kontrakt nr IST-2006-33596.

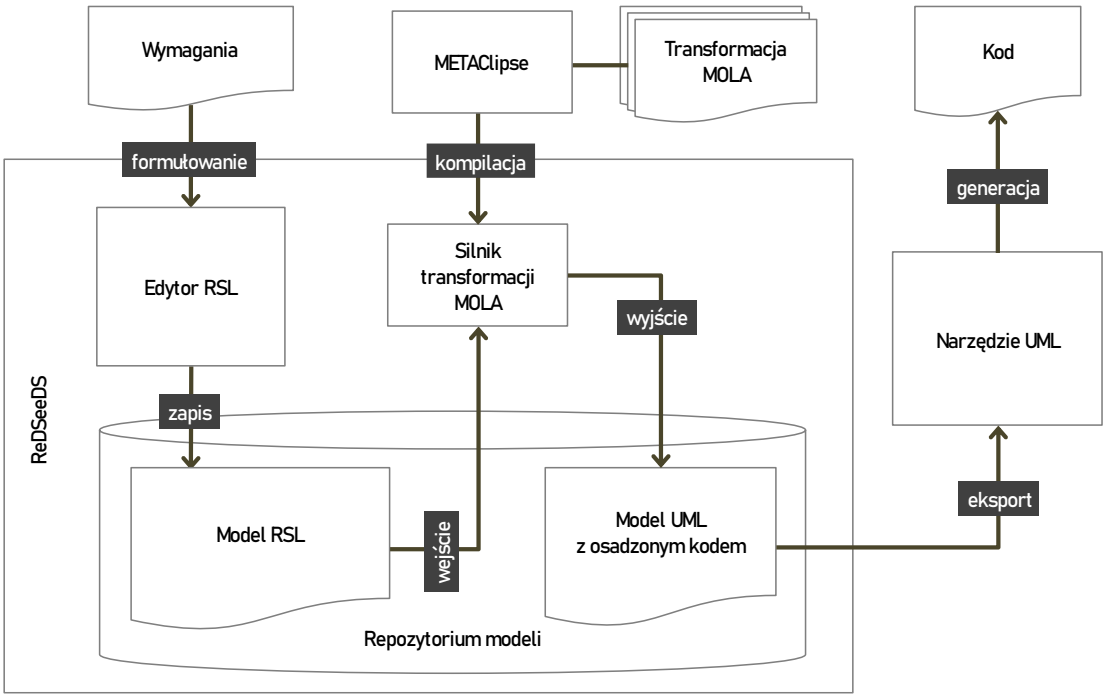


Rysunek 4 – Główne elementy środowiska ReDSeeDS

Jako że punktem wyjścia do tworzenia oprogramowania w środowisku ReDSeeDS są wymagania, kluczowym elementem jest język RSL. Modele niższego poziomu są zapisywane z użyciem podzbioru języka UML [39]. Oba języki zintegrowane są w ramach kompleksowego języka Software Case Language (SCL), który dodatkowo definiuje mechanizmy do śledzenia zależności pomiędzy modelami na różnym poziomie abstrakcji. Język SCL w połączeniu z językiem transformacji modeli MOLA [75] [111] oraz wsparciem narzędziowym stanowi spójne środowisko modelowania i transformacji modeli na ścieżce od wymagań do kodu.

Ogólny schemat takiej ścieżki prezentuje Rysunek 5. Zebrane wymagania formułowane są w języku RSL z wykorzystaniem edytora tego języka. Edytor, dzięki wbudowanym mechanizmom walidacji, zapewnia zgodność tworzonego modelu ze składnią języka. Model wymagań zapisywany jest w repozytorium modeli. Repozytorium implementuje metamodel języka SCL, dzięki czemu jest w stanie przechowywać zarówno modele wymagań jak też artefakty niższego poziomu tworzące Software Case. Artefakty te (np. modele projektowe wraz z osadzonym w nich kodem) umieszczane są w repozytorium jako wynik transformacji modelu źródłowego, wykonywanej przez silnik transformacji MOLA w oparciu o reguły transformacji dostępne w narzędziu ReDSeeDS. Do tworzenia tych reguł służy zewnętrzne narzędzie METAclipse, które udostępnia edytor języka MOLA oraz kompilator generujący pliki wykonywalne dla silnika transformacji. Modele w języku UML będące efektem transformacji, eksportowane są do zewnętrznego standardowego narzędzia typu CASE, posiadającego wbudowany generator kodu dla modeli UML. W obecnej wersji narzędzie ReDSeeDS

zintegrowane jest z narzędziami Enterprise Architect[13] oraz Modelio[14] poprzez interfejsy programistyczne tych narzędzi (API). Dzięki temu proces eksportu i generacji kodu może być wyzwolony automatycznie bezpośrednio po wykonaniu transformacji.



Rysunek 5 – Schemat działania środowiska ReDSeeDS

Od strony technologicznej narzędzie ReDSeeDS bazuje na platformie Eclipse[15] [112] [113] oraz środowisku Graphical Modeling Framework (GMF)[16] [114]. Repozytorium przechowuje modele w postaci struktur TGraphs, które są efektywną reprezentacją metamodelu języka SCL [115]. System ReDSeeDS posiada również mechnizm wyszukiwania i ponownego wykorzystania Software Case'ow lub ich fragmentów poprzez porównanie podobieństw między specyfikacjami wymagań [116] [108].

W ramach projektu REMICS [46] powstało rozszerzenie języka RSL – RSL-AL – umożliwiające korzystanie z gotowych wzorców scenariuszy przypadków użycia powiązanych

[13] https://sparxsystems.com

[14] https://www.modelio.org

[15] https://www.eclipse.org

[16] https://www.eclipse.org/modeling/gmp

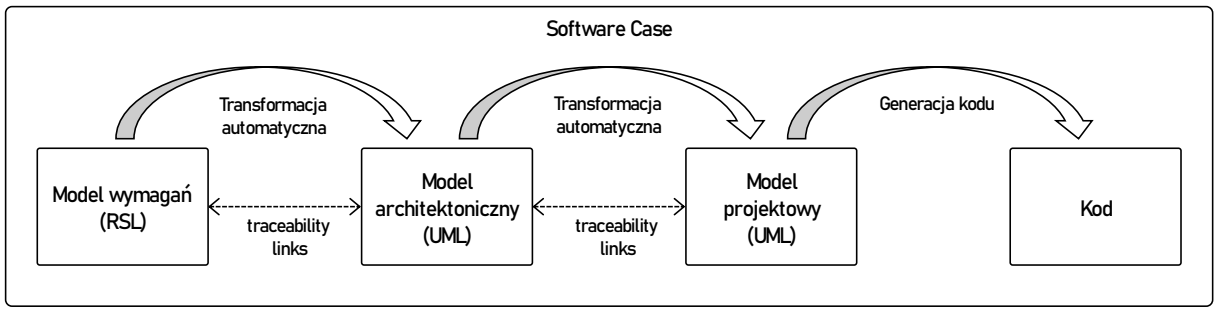
z elementami dziedzinowymi (Application Logic Patterns - ALPs) [117] [118], co pozwala zredukować nakład pracy związany z tworzeniem specyfikacji wymagań.

Powstało również inne istotne rozszerzenie języka RSL wraz z precyzyjną semantyką, które umożliwia generowanie kodu w warstwie logiki biznesowej na bazie modeli dziedzinowych w rozszerzonej notacji RSL-DL [53]. Notacja ta została zaprojektowana jako uniwersalny deklaratywny język modelowania logiki dziedzinowej, niezależny od dziedziny problemu. Dla notacji zostały zdefiniowane i zaimplementowane reguły translacji do kodu w języku Java.

Efektem niniejszej pracy jest z kolei precyzyjna semantyka translacyjna dla języka RSL, która definiuje zestaw reguł przekształcania konstrukcji tego języka na język Java dla konkretnego środowiska aplikacyjnego. Pozwoliło to zaimplementować transformację, której efektem jest kompletny kod w warstwach logiki aplikacji oraz interfejsu użytkownika. Uzyskane rezultaty są komplementarne względem pracy [53]. Połączenie rezultatów tych prac pozwala generować kod aplikacji we wszystkich warstwach typowej architektury trójwarstwowej.

2.6.2 Język SCL

Podczas każdego typowego projektu związanego z wytwarzaniem oprogramowania tworzone są artefakty na różnym poziomie abstrakcji, między którymi istnieją mniej lub bardziej formalne powiązania: specyfikacje, modele systemu, kod. Wszystkie razem powinny tworzyć kompleksowy artefakt będący finalnym produktem projektu, gdzie wymagania wobec systemu znajdują odzwierciedlenie w finalnym kodzie.



Rysunek 6 – Główna idea Software Case'a w środowisku ReDSeeDS

Taki kompleksowy artefakt w środowisku ReDSeeDS nosi nazwę Software Case. Jest to zbiór powiązanych ze sobą modeli oraz kodu docelowego systemu lub fragmentu systemu stanowiącego pewną funkcjonalną całość. Taka definicja wynika z faktu, że ReDSeeDS był

tworzony również z myślą o ponownym wykorzystaniu określonych kompletnych fragmentów systemów wyszukiwanych na podstawie precyzyjnego modelu wymagań traktowanego jako swego rodzaju indeks dla artefaktów niższego poziomu [45] [108]. W kontekście niniejszej pracy interesuje nas jednak inna idea stojąca za Software Casem. Idea ta zilustrowana jest na Rysunek 6. Artefakty na różnych poziomach abstrakcji wchodzące w skład Software Case'a wyrażone są w formie modeli w językach modelowania formalnie zdefiniowanych za pomocą metamodelu i posiadających graficzną lub tekstową składnię konkretną. Modele na najniższym poziomie abstrakcji (modele projektowe) mogą być dodatkowo wzbogacone o kod powiązany z określonymi elementami tego modelu (np. operacjami klas). Elementy poszczególnych modeli mogą być ze sobą powiązane odpowiednimi relacjami (traceability links). Software Case definiuje cztery poziomy modeli: model wymagań, model architektoniczny, model projektowy oraz kod.

Aby umożliwić tworzenie kompletnych i wewnętrznie spójnych Software Case'ów, stworzony został specjalny język – Software Case Language (SCL) [39] [119]. Założeniem projektowym tego języka było dostarczenie konstrukcji niezbędnych do wyrażania modeli i ich powiązań na wszystkich zdefiniowanych poziomach abstrakcji oraz zapewnienie spójnej przestrzeni do transformacji modeli. Zostało to osiągnięte poprzez integrację w ramach SCL dwóch innych języków:

- Requirements Specification Language (RSL) – do wyrażania wymagań,
- Unified Modeling Language (UML) – do wyrażania modeli architektonicznych i projektowych.

Innym ważnym założeniem projektowym była rozszerzalność. Stąd, język SCL został zdefiniowany w oparciu o metamodel w języku MOF. Dzięki temu, każdy inny język modelowania zdefiniowany w MOF może być łatwo zintegrowany z językiem SCL.

Język RSL [120] [36] powstał w ramach projektu ReDSeeDS. Od początku był projektowany jako język specyfikacji wymagań, który połączyłby dwie cechy:

- prostą składnię konkretną składającą się z elementów graficznych oraz tekstu, umożliwiającą rozumienie specyfikacji wymagań przez osoby nieposiadające doświadczenia w obszarze inżynierii wymagań i wytwarzania oprogramowania (np. użytkowników końcowych),

- precyzyjną składnię abstrakcyjną umożliwiającą automatyczne przetwarzanie specyfikacji wymagań w celu generowania modeli projektowych oraz kodu aplikacji.

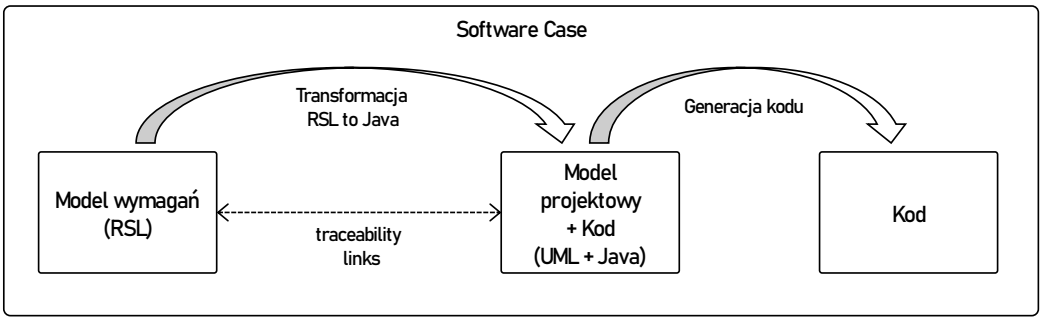
Język RSL w dużym stopniu opiera się na popularnych konstrukcjach znanych m.in. z języka UML (np. przypadki użycia, aktywności, klasy), przy czym konstrukcje te zostały znacząco zmodyfikowane, ujednolicone i doprecyzowane zarówno pod względem składni jak i semantyki. Istotną częścią notacji RSL są konstrukcje wyrażane w języku naturalnym oraz ograniczonym języku naturalnym (ang. constrained natural language). Dzięki takiej charakterystyce, zrealizowano pierwsze z wymienionych powyżej założeń projektowych języka [34]. Podobnie jak w przypadku języka UML, język RSL jest również zdefiniowany w postaci metamodelu w języku MOF, co – jak wspomniano powyżej – pozwoliło zintegrować oba języki w ramach SCL. Szczegółowy opis języka RSL zawierają rozdziały 3 oraz 4 niniejszej pracy. Omawiana jest tam jego składnia konkretna i abstrakcyjna, co jest niezbędne do zrozumienia semantyki translacyjnej dla języka RSL oraz implementacji transformacji „RSL to Java" będących zasadniczą częścią tej pracy.

Język RSL został stworzony jako zupełnie nowy język, gdyż nie istniał inny, powszechnie używany język do modelowania wymagań spełniający założenia projektowe środowiska ReDSeeDS. Nie było natomiast zasadne tworzenie nowego języka do wyrażania pozostałych artefaktów. Rolę tę z powodzeniem pełni język UML, który przez lata stał się standardem *de facto* w dziedzinie modelowania systemów oprogramowania.

Artefakty niższego poziomu niż wymagania, są tworzone w ramach Software Case'a w wyniku transformacji. Zatem to zastosowana transformacja (lub zestaw transformacji) decyduje o zawartości Software Case'a. W ramach projektu ReDSeeDS powstał zestaw dwóch transformacji. Jedna przekształcająca model wymagań w model architektoniczny systemu oraz druga – przekształcająca otrzymany model architektoniczny w bardziej szczegółowy model projektowy (jak zilustrowano na Rysunek 6). Są to jednak modele niezależne od konkretnej platformy technologicznej czy wzorców projektowych. Model projektowy reprezentuje klasy prostych obiektów POJO[17] z pustymi ciałami metod. Na podstawie takiego modelu możliwe jest wygenerowanie jedynie prostego kodu szkieletowego dla klas [40] [45].

[17] Plain Old Java Object – prosty obiekt nieobciążony dziedziczeniem, nieposiadający atrybutów wymaganych do współpracy ze specyficznym środowiskiem tworzenia aplikacji (ang. framework).

W późniejszym czasie powstała nieco bardziej rozbudowana transformacja umożliwiająca przekształcanie modelu wymagań do kodu uwzględniającego dynamikę aplikacji zgodnej ze wzorcem MVP/MVC[18] [50] [49] [51]. Transformacja ta realizowała względnie prosty zestaw reguł translacji dla modelu RSL a wynikowy kod nie był na tyle kompletny aby umożliwić jego bezpośrednią kompilację oraz uruchomienie.



Rysunek 7 – Software Case tworzony przy użyciu transformacji „RSL to Java"

W ramach niniejszej pracy opracowana została transformacja „RSL to Java" implementująca istotnie rozszerzoną semantykę translacyjną dla języka RSL. Transformacja ta generuje model projektowy specyficzny dla konkretnego środowiska docelowego bezpośrednio z modelu wymagań (Rysunek 7). Model ten wzbogacony jest o kompletny kod w języku Java w warstwie logiki aplikacji oraz interfejsu użytkownika i realizuje konkretny stos technologiczny. Dzięki temu możliwe jest wygenerowanie kompletnego kodu, który może być skompilowany i uruchomiony bez konieczności ręcznej edycji.

2.6.3 Język transformacji modeli MOLA

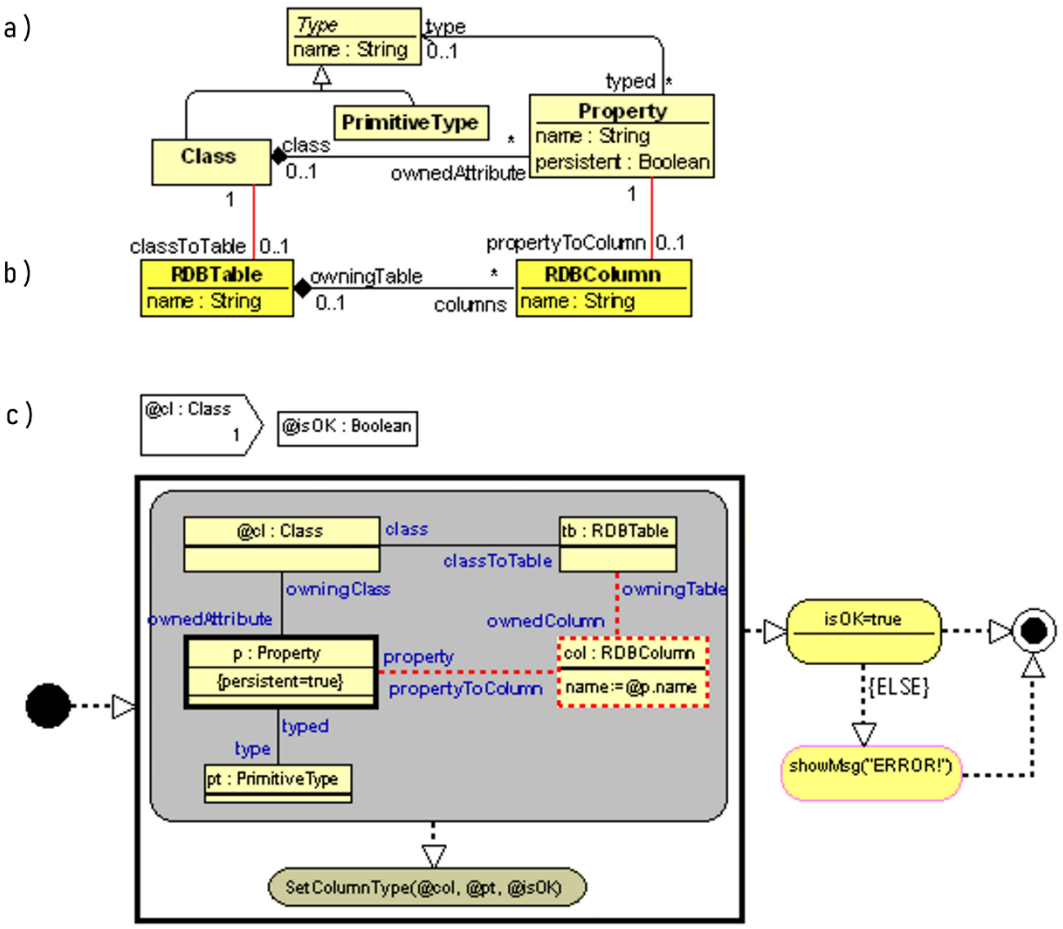
Budowa Software Case'a w języku SCL wymaga przekształcenia (lub szeregu przekształceń) jednego modelu w drugi, zgodnie z określonymi regułami translacji. W przypadku, kiedy oba modele wyrażone są w językach posiadających dobrze zdefiniowany metamodel (jak w przypadku języka SCL), można do tego celu użyć język transformacji modeli. Środowisko ReDSeeDS wykorzystuje język transformacji modeli MOLA [75] [111]. Język ten, wraz ze środowiskiem deweloperskim, został stworzony na Uniwersytecie Łotewskim w Rydze, głównie z myślą o zastosowaniach w obszarze Model-Driven Software Development (MDSD).

[18] MVP – Model View Presenter; MVC – Model View Controller

MOLA jest graficznym językiem transformacji modeli, który umożliwia przekształcanie modelu źródłowego w model docelowy. Definicja transformacji w tym języku składa się z definicji metamodelu źródłowego i docelowego oraz procedur. Metamodel źródłowy i docelowy definiowane są w języku metamodelowania MOLA, który jest w dużej mierze tożsamy z językiem Essential MOF (EMOF) [121]. Dzięki temu jest czytelny dla osób zaznajomionych z modelowaniem i metamodelowaniem, w szczególności ze standardem MOF. Podział na metamodel źródłowy i docelowy jest w języku MOLA jedynie semantyczny – metamodele nie są odseparowane syntaktycznie. W szczególności, model źródłowy i docelowy mogą mieć ten sam metamodel. Procedury w języku MOLA stanowią część wykonawczą transformacji, która definiuje sposób przekształcenia modeli. Każda transformacja może składać się z dowolnej liczby procedur, przy czym jedna z nich musi być procedurą główną, od której rozpoczyna się działanie transformacji. Procedury są tworzone zgodnie z paradygmatem imperatywnym – instrukcje wykonywane są w ściśle określonym porządku. Procedura może być wywoływana przez inną procedurę lub rekurencyjnie. Zasadniczymi konstrukcjami języka są jednak reguły działające w oparciu o dopasowanie wzorca (ang. pattern match) oraz pętle oparte na regułach. Deklaratywny charakter reguł operujących na elementach metamodelu źródłowego i docelowego, znacząco ułatwia definiowanie i czytanie transformacji. Ponadto, dzięki graficznej składni konkretnej, transformacje w języku MOLA są w dużym stopniu samodokumentujące. Te cechy sprawiają, że MOLA świetnie nadaje się do implementacji transformacji „RSL to Java” implementującej semantykę translacyjną dla języka RSL.

Rysunek 8 przedstawia prosty przykład transformacji w języku MOLA. Posługując się tym przykładem, omówimy podstawowe elementy języka, niezbędne do zrozumienia transformacji „RSL to Java” opisanej w Rozdziale 6. Kompletny opis języka MOLA można znaleźć w jego specyfikacji [111] oraz w tutorialu na oficjalnej stronie internetowej MOLA[19].

[19] http://mola.mii.lu.lv/



Rysunek 8 – Podstawowe elementy języka MOLA: metamodel źródłowy (a), metamodel docelowy (b) oraz procedura główna (c)

Część a oraz b omawianego rysunku przedstawia metamodel źródłowy oraz docelowy. W oparciu o ten metamodel możemy tworzyć bardzo proste modele składające się z klas (metaklasa `Class`) posiadających nazwę (metaatrybut `name` odziedziczony po metaklasie `Type`) oraz zawierających dowolną liczbę właściwości (metaklasa `Property`). Każda właściwość posiada nazwę (metaatrybut `name`) oraz może mieć określony typ będący klasą bądź typem prostym (metaklasa `PrimitiveType`). Właściwość może być trwała lub nie, co określa wartość metaatrybutu `persistent`.

Metamodel docelowy jest jeszcze prostszy. W oparciu o niego można modelować tabele relacyjnej bazy danych (`RDBTable`) posiadających dowolną liczbę kolumn (`RDBColumn`). Warto zwrócić uwagę, że między elementami metamodelu źródłowego i docelowego występują metaasocjacje – klasa może być powiązana z tabelą a właściwość z kolumną.

Rysunek 8c przedstawia procedurę stanowiącą fragment transformacji. Kolejność wykonywania działań w ramach procedury określają elementy sterujące: element startowy

(czarna kropka), przepływy kontroli (strzałki z przerywaną linią i otwartym grotem) oraz element końcowy (czarna kropka w okręgu).

Głównym elementem przedstawionej procedury jest pętla, której składnia konkretna ma postać czarnego prostokąta. Każda pętla musi posiadać tzw. głowę pętli, czyli regułę (szary prostokąt z zaokrąglonymi rogami). Reguła może się składać ze wzorca i/lub akcji. Wzorzec odnosi się do instancji elementów metamodelu: metaklas i metaasocjacji. Reprezentacja instancji metaklasy w regule ma postać zbliżoną do metaklasy – żółty prostokąt z czarnym obramowaniem. Musi ona posiadać nazwę konkretnej instancji (nazwę zmiennej) oraz wskazywać nazwę metaklasy. Obie nazwy rozdziela dwukropek. Znak „`@`" poprzedzający nazwę instancji oznacza referencję do elementu zadeklarowanego w innym miejscu. Instancje mogą ponadto definiować warunki dla atrybutów w postaci wyrażeń OCL. Podczas wykonywania transformacji, próbuje ona znaleźć zadeklarowany we wzorcu układ elementów w modelu źródłowym lub docelowym z uwzględnieniem zadanych ograniczeń. Reguły z wzorcami są wyrażeniami warunkowymi – jeżeli we wzorcu występują akcje, są one wykonywane tylko w przypadku pomyślnego dopasowania wzorca. Ponadto, z reguły mogą wychodzić dwa przepływy kontroli – dla pomyślnego i niepomyślnego wykonania reguły. Ten drugi oznaczany jest jako `{ELSE}`. Akcje mogą tworzyć nowe instancje metaklas oraz metaasocjacji (oznaczane są wtedy czerwoną przerywaną linią) lub je usuwać (oznaczane są wtedy czarną przerywaną linią). Akcje mogą również przypisywać wartości atrybutom istniejących lub tworzonych instancji.

W przypadku pętli, jeden z elementów wzorca musi pełnić rolę zmiennej pętli. Zmienna oznaczana jest pogrubionym obramowaniem. Pętla wykonywana jest raz dla każdej instancji zmiennej pętli, pod warunkiem dopasowania wzorca, którego elementem jest zmienna pętli. Pętle mogą być dowolnie zagnieżdżane. Pętle zagnieżdżone mogą odwoływać się do zmiennych pętli nadrzędnych poprzez referencję.

W przedstawionym przykładzie reguła pętli wykonywana jest dla każdej właściwości należącej do klasy `@cl` (przekazanej jako parametr wejściowy procedury, reprezentowany przez biały pięciokąt z nazwą parametru, jego typem oraz numerem porządkowym) oraz pod warunkiem, że atrybut `persistent` tej właściwości ma wartość „true", jest ona typu prostego, a klasa, do której ta właściwość należy, jest powiązana z tabelą bazodanową (`tb`). Jeżeli te warunki są spełnione, tworzona jest kolumna o nazwie takiej samej jak nazwa właściwości. Razem

z kolumną tworzone są asocjacje łączące ją z właściwością oraz tabelą. Są one instancjami odpowiednich metaasocjacji.

W przykładowej pętli, oprócz reguły stanowiącej głowę pętli, znajduje się jeszcze wywołanie procedury `SetColumnType()` w postaci zaoblonego prostokąta w kolorze khaki z nazwą wywoływanej procedury oraz parametrami wywołania. Wywołanie to jest wykonywane jedynie w przypadku pomyślnego wykonania reguły. Każda procedura może mieć dowolną liczbę parametrów wywołania. Parametry mogą być wejściowe lub wejściowo-wyjściowe. W przypadku tych drugich, procedura może nadpisywać ich wartość. W języku MOLA możliwe jest również wywoływanie procedur zewnętrznych napisanych w języku C++. Wywołanie takie ma postać zbliżoną do wywołania zwykłej procedury, przy czym ma kolor żółty z różowym konturem. Przykładem wywołania procedury zewnętrznej w omawianej transformacji jest `showMsg()`, której zadaniem jest wyświetlenie użytkownikowi uruchamiającemu transformację okna z komunikatem, którego tekst przekazywany jest jako parametr wywołania procedury zewnętrznej.

Oprócz parametrów wywołania, procedura może również definiować zmienne lokalne. Mają one postać białego prostokąta z nazwą i typem zmiennej. Zmienne, podobnie jak parametry, mogą być typu prostego (np. Boolean, String, Integer) lub złożonego. Typy złożone są referencjami do instancji metaklas.

Ostatnim istotnym elementem procedury w języku MOLA jest wyrażenie tekstowe. Ma ono postać żółtego zaoblonego prostokąta przedzielonego poziomą linią. W górnej części określane są warunki logiczne a w dolnej – instrukcje przypisania wartości zmiennym. Instrukcje w dolnej części wykonywane są jeżeli spełniony jest warunek w górnej części. Wyrażenie tekstowe może mieć, podobnie jak reguła, dwa wychodzące przepływy kontroli – dla przypadku spełnienia oraz niespełnienia warunku. W omawianym przykładzie, wspomniane wywołanie procedury `showMsg()` jest wykonywane tylko wtedy, gdy wartość zmiennej `isOK` (jej wartość jest ustawiana w trakcie wykonania procedury `SetColumn()`), jest różna od „true”. W przeciwnym razie, procedura kończy swoje działanie.

3 Język specyfikacji wymagań RSL

RSL jest językiem formalnym służącym do modelowania wymagań oprogramowania. Stworzenie nowego języka modelowania wymaga zdefiniowania trzech elementów: składni abstrakcyjnej, składni konkretnej oraz semantyki. Składnia abstrakcyjna stanowi gramatykę języka – określa wszystkie możliwe elementy języka oraz ich logiczną strukturę, abstrahując od sposobu prezentacji tych elementów, np. w postaci graficznej czy tekstowej. Wizualną formę elementów języka określa składnia konkretna. W przypadku graficznych elementów języka, składnia konkretna może określać m.in. ich dopuszczalne kształty, układ, linie, kolory, warianty prezentacji. Przykładowo, składnia konkretna dla przypadku użycia w języku UML, to elipsa z nazwą przypadku wewnątrz elipsy lub pod nią. W przypadku elementów tekstowych, składnia konkretna określa układ jednostek leksykalnych, które razem tworzą większe struktury jak np. wyrażenia, zdania, itp. Przykładowo, składnia konkretna dla atrybutu klasy w języku UML jest następująca: symbol widoczności, nazwa atrybutu, dwukropek, typ atrybutu, np.”+ name : String”. Ostatnim elementem definicji języka jest semantyka, która określa znaczenie poszczególnych elementów języka.

Celem tego rozdziału jest przedstawienie składni konkretnej języka RSL oraz jego semantyki z punktu widzenia osoby tworzącej specyfikację wymagań. Jest to nieformalny sposób przedstawienia semantyki. Ponieważ praca ta skupia się na części języka RSL, która służy do specyfikowania wymagań funkcjonalnych, dlatego też nieformalny opis semantyki sprowadza się do zaprezentowania jakie zachowania docelowego systemu wyrażają poszczególne elementy języka oraz jak może wyglądać warstwa prezentacji tego systemu. Możemy to nazwać „semantyką obserwowalną” aby odróżnić ją od semantyki translacyjnej (ang. translational semantics), która w sposób formalny opisuje znaczenie elementów danego języka poprzez określenie precyzyjnych reguł ich tłumaczenia na elementy innego języka o dobrze określonej semantyce. W niniejszej pracy semantyka translacyjna języka RSL opisana jest z użyciem języka Java w rozdziale 5. Składnia abstrakcyjna języka RSL, zdefiniowana w postaci metamodelu, opisana jest w rozdziale 4.

Ze względu na mnogość konstrukcji jakie oferuje język RSL, rozdział ten ogranicza się do opisu tych elementów języka, które zostały wykorzystane do zdefiniowania semantyki translacyjnej. Rozdział ten uwzględnia również dodatkowe reguły będące uzupełnieniem definicji języka RSL, które zostały wprowadzone na potrzeby definicji semantyki translacyjnej w celu ułatwienia implementacji transformacji „RSL to Java” w zakresie możliwym do realizacji

w ramach tej pracy. Oryginalną definicję języka można znaleźć w dokumencie będącym jego formalną specyfikacją [36].

Prezentowane w tym rozdziale konstrukcje języka zostały zilustrowane fragmentami specyfikacji wymagań dla aplikacji Patient Care System. Jest to prosta aplikacja pozwalająca zarządzać danymi pacjentów przychodni lekarskiej oraz ich wizytami lekarskimi. Pełne studium przypadku dla podobnej aplikacji, uwzględniające kod wygenerowany przy użyciu transformacji „RSL to Java", zostało przedstawione w rozdziale 7.

3.1 Struktura specyfikacji wymagań

W powszechnej praktyce [1] [2] [3] [4] [5] [6] [7] [8] [9] specyfikacje wymagań tworzone są w postaci dokumentów, na podstawie których inżynierowie oprogramowania tworzą projekt systemu a następnie implementują go w oparciu o przyjęte wzorce architektoniczne oraz stos technologiczny. Jest to zazwyczaj proces manualny, w którym efekt końcowy, w postaci działającej aplikacji, zależy w pewnym stopniu od indywidualnej interpretacji specyfikacji wymagań. Stopień ten jest tym wyższy, im mniej precyzyjna, spójna i jednoznaczna jest specyfikacja wymagań. Problem pojawia się wtedy, gdy wspomniana interpretacja jest rozbieżna z intencjami autora specyfikacji. Kluczowa w tym procesie jest więc jakość specyfikacji wymagań. Działaniem mającym poprawić jakość specyfikacji wymagań jest stosowanie szablonów specyfikacji opartych o powszechne standardy, np. IEEE 830-1998 [122]. Wzorce takie koncentrują się z reguły na dwóch kwestiach:

- czym jest wymaganie i jak je opisać,
- jak klasyfikować i grupować wymagania.

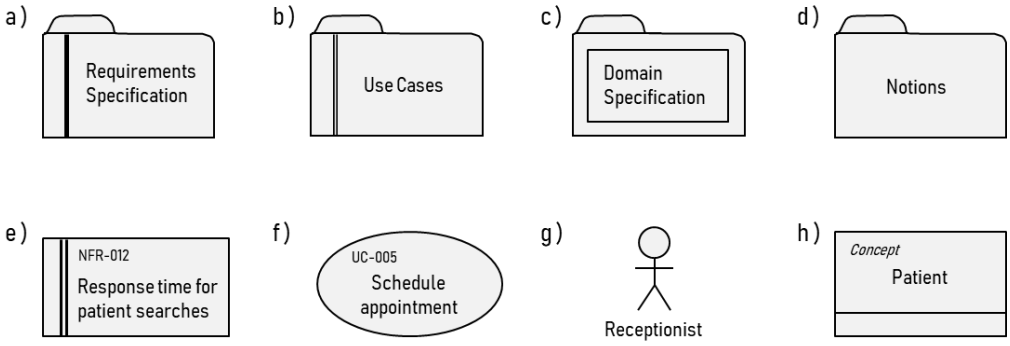
Nawet stosowanie szablonów nie rozwiązuje problemu spójności i jednoznaczności, np. konsekwentnego stosowania określonej terminologii, czy zależności między wymaganiami. Wymagania, mimo że są zorganizowane w uporządkowany dokument, nadal są wyrażane językiem nieformalnym a zależności między nimi muszą być utrzymywane ręcznie.

Aby jak najlepiej wspomagać tworzenie spójnych i jednoznacznych modeli wymagań, język RSL wprowadza precyzyjnie zdefiniowaną notację opartą o proste elementy graficzne (m.in. diagramy przypadków użycia) w połączeniu z ograniczonym językiem naturalnym (scenariusze przypadków użycia). Ponadto, język RSL wprowadza rozgraniczenie między opisem dziedziny problemu (pojęciami dziedzinowymi) a specyfikacją zachowania systemu (wymaganiami funkcjonalnymi), zapewniając jednocześnie mechanizmy utrzymywania spójności między tymi

dwiema częściami. Dzięki tym mechanizmom, edytor języka RSL może m.in. automatycznie utrzymywać zależności między wymaganiami a opisem dziedziny, np. tworzyć pojęcia dziedzinowe w trakcie pisania treści wymagań lub tworzyć asocjacje do już istniejących pojęć.

3.1.1 Pojęcia podstawowe

Kompletny model w języku RSL składa się z dwóch części: specyfikacji wymagań (Requirements Specification) oraz specyfikacji dziedziny (Domain Specification). Specyfikacje te przechowują, odpowiednio, wymagania oraz elementy dziedzinowe. Obie grupy elementów mogą być zorganizowane w hierarchiczne struktury za pomocą pakietów. Notacja dla tych elementów jest zbliżona do tej znanej z języka UML [79], z pewnymi różnicami mającymi na celu wizualne odróżnienie pakietów różnych typów. Przykłady notacji przedstawione zostały na Rysunek 9. Zarówno w ramach specyfikacji wymagań i specyfikacji dziedziny, pakiety tego samego typu mogą być zagnieżdżane i tworzyć struktury drzewiaste, w których liśćmi są wymagania lub elementy dziedzinowe – w zależności od typu pakietu.

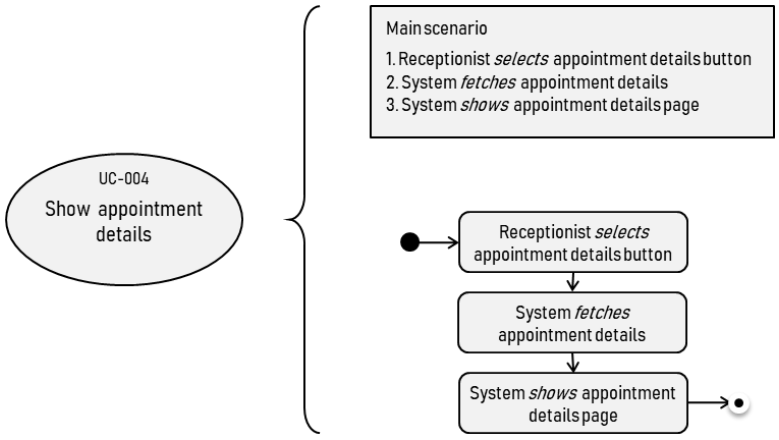


Rysunek 9 – Przykłady notacji dla: (a) specyfikacji wymagań, (b) pakietu wymagań, (c) specyfikacji dziedziny, (d) pakietu dziedzinowego, (e) wymagania, (f) przypadku użycia, (g) aktora, (h) elementu dziedzinowego

Język RSL wprowadza dwa poziomy abstrakcji dla opisu wymagań. Jest to zrealizowane poprzez ścisłe rozróżnienie między wymaganiami a ich reprezentacjami. Wymagania jako takie są prostymi elementami zawierającymi nazwę wymagania, identyfikator oraz ewentualne dodatkowe atrybuty określające np. priorytet wymagania, osobę odpowiedzialną za dane wymaganie, itp. Ten poziom abstrakcji jest wystarczający do zarządzania wymaganiami w projekcie, np. przydzielaniu wymagań do kolejnych iteracji w przyrostowym cyklu wytwórczym. Reprezentacje wymagań, natomiast, stanowią ich szczegółową i ustrukturyzowaną specyfikację – dużo bardziej precyzyjną niż język naturalny. Ten poziom

abstrakcji umożliwia definiowanie złożonej semantyki translacyjnej i automatyczne przetwarzanie modelu wymagań w kod aplikacji czy inne zaawansowane operacje, np. badanie podobieństwa między dwiema specyfikacjami wymagań w celu ich ponownego użycia [108] [116].

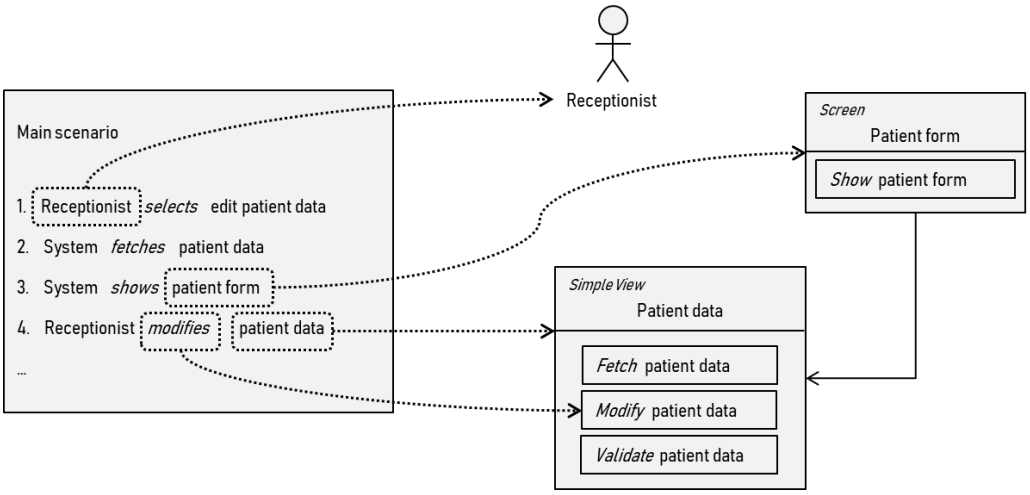
Z punktu widzenia semantyki translacyjnej opisanej w tej pracy, najważniejszym rodzajem wymagań w języku RSL są przypadki użycia. Podobnie jak w języku UML, służą one do wyrażania wymagań funkcjonalnych. Mają one swoje reprezentacje w postaci scenariuszy przypadków użycia wyrażonych w formie ustrukturyzowanego tekstu, przy użyciu ograniczonego języka naturalnego, bądź w równoważnej postaci graficznej. Przykłady obu reprezentacji przedstawia Rysunek 10. Model przypadków użycia wraz ze scenariuszami stanowi precyzyjny opis logiki działania budowanej aplikacji. Przez logikę aplikacji rozumiemy tutaj zachowanie systemu w interakcji z użytkownikiem, inaczej – sekwencje interakcji pomiędzy aktorem i systemem, który wykonuje określone operacje w odpowiedzi na akcje użytkownika. Nie należy mylić logiki aplikacji z logika biznesową, która określa sposób wewnętrznego przetwarzania danych przez system w ramach niektórych operacji, np. wykonanie odpowiednich algorytmów bądź reguł biznesowych [123] [124] [125] [126].



Rysunek 10 – Prosty przypadek użycia i jego reprezentacje

Rysunek 11 przedstawia (symbolicznie, za pomocą strzałek nie będących częścią notacji) hiperłącza (ang. hyperlinks) pomiędzy scenariuszem przypadku użycia „Edit patient data” a elementami dziedzinowymi różnych typów: aktorem „Receptionist” oraz pojęciami „Patient data” i „Patient Form”. Hiperłącza mogą odnosić sie również do elementów składowych pojęć – tzw. stwierdzeń dziedzinowych (ang. domain statements), które definiują cechy behawioralne

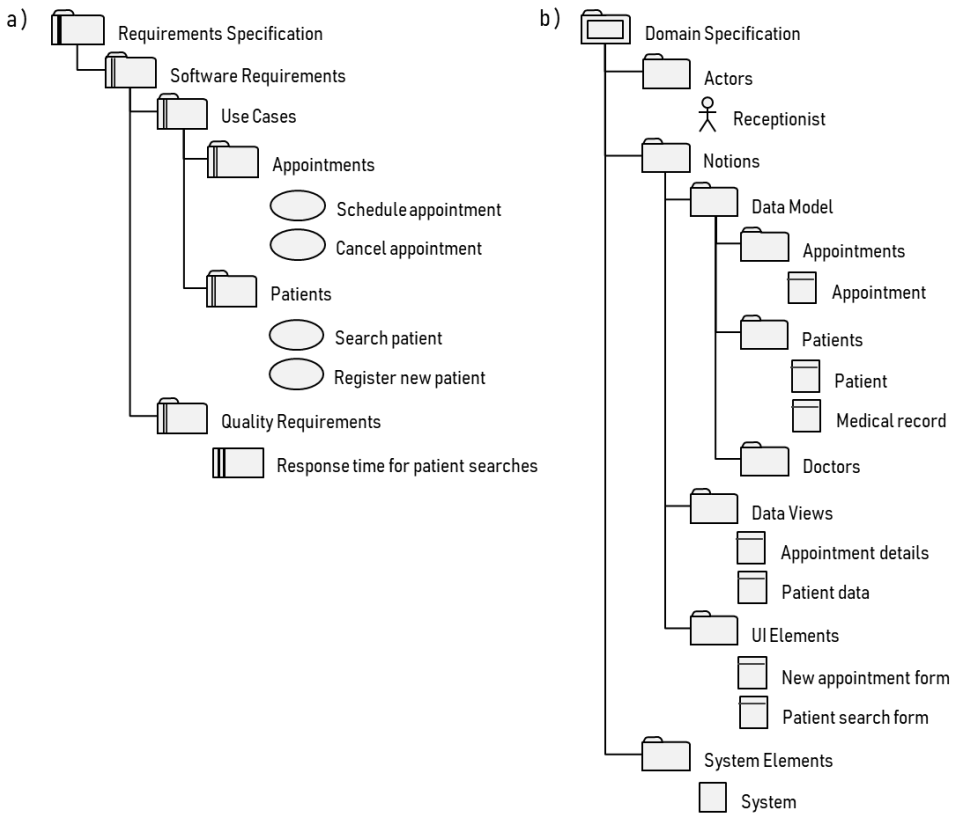
pojęć. W przedstawionym przykładzie jest to „modify patient data”. Mechanizm hiperłączy pozwala utrzymać całkowitą spójność pomiędzy modelem wymagań a modelem dziedziny.



Rysunek 11 – Hiperłącza pomiędzy reprezentacją przypadku użycia i elementami dziedzinowymi

3.1.2 Pakiety i ich struktura

Język RSL, jak wspomniano wcześniej, definiuje kilka rodzajów pakietów – każdy przeznaczony do przechowywania określonych typów elementów. Dzięki temu modele w języku RSL mają precyzyjną i czytelną strukturę. Poza podziałem całego modelu w języku RSL na specyfikacje wymagań oraz specyfikację dziedziny, definicja języka nie narzuca określonej struktury obu części. Struktura taka może mieć jednak znaczenie podczas transformacji modeli w języku RSL do modeli niższego poziomu oraz kodu. Semantyka translacyjna opisana w tej pracy oraz jej implementacja w postaci transformacji „RSL to Java” uwzględniają strukturę pakietów, która jest pośrednio odzwierciedlana w wygenerowanym kodzie w postaci pakietów klas. Poniżej przedstawione zostały proste reguły podziału specyfikacji na odpowiednie podpakiety.



Rysunek 12 – Podział modelu wymagań na pakiety dla (a) specyfikacji wymagań, (b) specyfikacji dziedziny

Rysunek 12 przedstawia strukturę modelu w języku RSL na przykładzie systemu „Patient Care System". Specyfikacja wymagań (Rysunek 12a), zgodnie z powszechnymi praktykami inżynierii wymagań, podzielona jest na pakiet zawierający wymagania funkcjonalne (pakiet „Use Cases") oraz wymagania niefunkcjonalne (pakiet „Quality Requirements"). Pakiet „Use Cases" zawiera wszystkie przypadki użycia stanowiące jednostki funkcjonalności systemu. Każda nietrywialna aplikacja może składać się z dziesiątek lub setek przypadków użycia, dlatego najczęściej niezbędny jest dalszy podział przypadków użycia na podpakiety. Podziału takiego można dokonać na podstawie różnych kryteriów. W przedstawionym przykładzie podpakiety grupują przypadki użycia według obszarów funkcjonalnych: pakiet „Appointments" zawiera wszystkie przypadki użycia związane z umawianiem pacjentów na wizyty lekarskie, natomiast pakiet „Patients" zawiera przypadki związane z ewidencją pacjentów przychodni.

Pakiet „Quality Requirements" zawiera wymagania niefunkcjonalne wobec tworzonego systemu. Również ta część zazwyczaj wymaga dalszego ustrukturyzowania. Podpakiety mogą, na przykład, odzwierciedlać powszechnie używaną klasyfikacje wymagań niefunkcjonalnych,

zgodnie z normą ISO/IEC 25010 [127]. Wymagania tego typu mogą być istotne z punktu widzenia transformacji modelu wymagań – niektóre z nich, np. te dotyczące bezpieczeństwa (ang. security) lub przenośności (ang. portability), mogą istotnie wpływać na generowany kod. W przypadku niniejszej pracy, wymagania niefunkcjonalne nie maja zdefiniowanej semantyki translacyjnej i nie będą dalej omawiane.

Drugą częścią modelu w języku RSL jest specyfikacja dziedziny. Jej wewnętrzna struktura (Rysunek 12b) składa się z trzech głównych pakietów: „Actors", „Notions", „System elements". Pakiet „Actors" zawiera aktorów dla przypadków użycia. Pakiet „System Elements" zawiera elementy reprezentujące główne logiczne komponenty systemu. Najczęściej jest to po prostu jeden element reprezentujący system lub aplikację (jak w przedstawionym przykładzie). W przypadku systemów złożonych z wielu aplikacji, każda z nich może być reprezentowana przez osobny element. Pakietem najistotniejszym z punktu widzenia semantyki translacyjnej jest pakiet „Notions" zawierający wszystkie pojęcia reprezentujące encje zarówno z dziedziny biznesowej jak i dziedziny aplikacji. Stąd wynika podział tego pakietu na podpakiety przechowujące pojęcia określonych typów: „Data Model" – pojęcia biznesowe reprezentujące dane przetwarzane przez system, „Data Views" – pojęcia z dziedziny aplikacji, reprezentujące widoki danych prezentowane użytkownikom za pośrednictwem interfejsu użytkownika oraz „UI Elements" – elementy interfejsu użytkownika, jak ekrany, komunikaty, kontrolki wyzwalające akcje. Podpakiety te mogą podlegać dalszemu podziałowi, jak w przypadku pakietu „Data Model", którego struktura odzwierciedla logiczny podział pojęć dziedzinowych.

3.2 Specyfikowanie dziedziny problemu i dziedziny aplikacji

Jak wspomniano wcześniej, kluczowa dla zachowania spójności i jednoznaczności modelu wymagań w języku RSL jest możliwość wiązania wymagań z elementami dziedziny za pomocą hiperłączy. Język RSL musi zatem zapewniać notację pozwalającą modelować wszystkie niezbędne elementy dziedzinowe w dwóch obszarach: elementy dziedziny problemu (dziedziny biznesowej) oraz elementy dziedziny aplikacji. Rozróżnienie to jest istotne, gdyż dziedzina problemu jest niezależna od aplikacji, która ma być zbudowana do obsługi danego problemu biznesowego. Elementy dziedziny problemu (logiczny model danych) stanowią abstrakcję fragmentu rzeczywistości i podlegają zmianom jedynie w przypadku zmian zachodzących w tym fragmencie rzeczywistości, np. zmian struktury organizacji czy zmian legislacyjnych mających wpływ na dany obszar biznesowy. Elementy dziedziny aplikacji, natomiast, ściśle zależą od projektowanej logiki systemu, tj. sposobu prezentacji encji biznesowych oraz zakresu

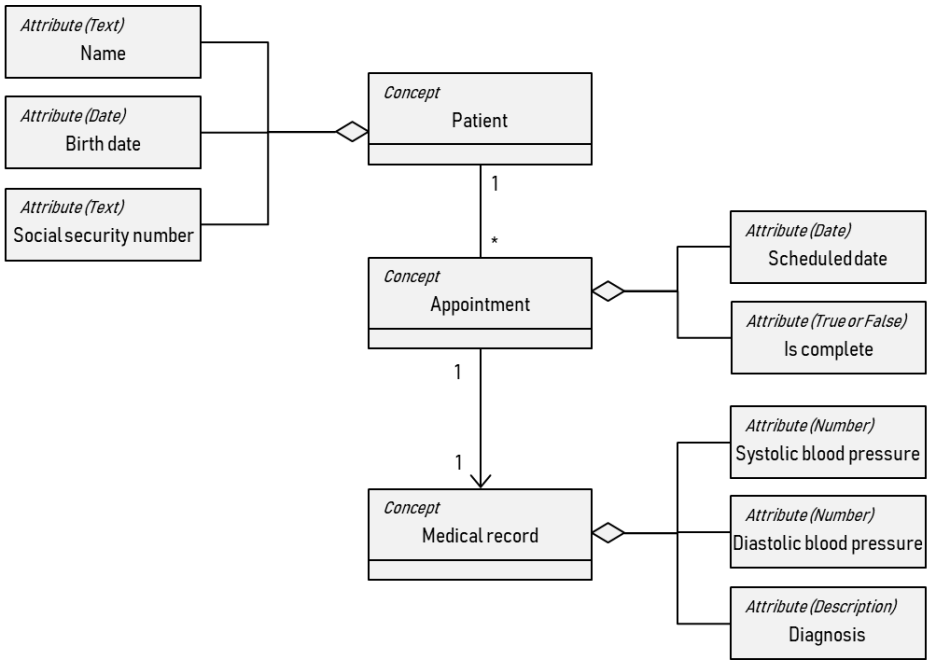
i sposobu ich przetwarzania. Przykładowo, logika działania aplikacji mobilnej będącej częścią jakiegoś systemu jest zazwyczaj nieco uproszczona (m.in. ze względu na mniejszy rozmiar ekranów urządzeń mobilnych) względem logiki aplikacji webowej czy desktopowej, mimo że wykorzystują one ten sam logiczny model danych.

W języku RSL dziedzina problemu modelowana jest za pomocą pojęć oraz ich atrybutów. Atrybuty mogą być wskazywane przez inne elementy modelu dziedzinowego – pojęcia z dziedziny aplikacji. Pojęcia te modelują elementy interfejsu użytkownika (ekrany, kontrolki, komunikaty, okna dialogowe) oraz zestawy danych prezentowanych za pomocą interfejsu użytkownika (tzw. widoki danych). Wszystkie elementy modelu dziedzinowego oraz reguły tworzenia takich modeli zostały opisane w dalszych podrozdziałach.

3.2.1 Dziedzina problemu

Kompletny model dziedziny problemu w języku RSL składa się z powiązanych ze sobą pojęć wraz z atrybutami. Rysunek 13 przedstawia przykładowy model dziedziny problemu. Aby odróżnić pojęcia reprezentujące elementy dziedziny problem (pojęcia biznesowe) od pojęć z dziedziny aplikacji, pojęcia maja typy określone za pomocą odpowiednich stereotypów. Mechanizm stereotypów daje możliwość definiowania dowolnych typów pojęć bez konieczności zmiany definicji języka. Pojęcia z dziedziny problemu mają typ „Concept”. Atrybuty w języku RSL modelowane są również za pomoca pojęć, przy czym posiadają stereotyp „Attribute”. Atrybuty powiązane z pojęciami typu „Concept” stanowią właściwości tych pojęć, podobnie jak właściwości (ang. properties) klas w języku UML. Każdy atrybut musi mieć przypisany jeden z typów podstawowych:

- „Text” – łańcuch znakowy,
- „Description” – opis tekstowy wielowierszowy,
- „Number” – liczba całkowita,
- „Floating point number” – liczba zmiennoprzecinkowa,
- „True or False” – wartość logiczna typu boolean,
- „Date” – data,
- „Time” – czas,
- „Secret text” – tekst ukryty, np. hasło.



Rysunek 13 – Model dziedziny problemu

Rysunek 13, oprócz pojęć i ich atrybutów, przedstawia również relacje między nimi. Do wyrażania relacji między pojęciami typu „Concept" używane są asocjacje – podobne do tych znanych z modelu klas w języku UML. Asocjacje mogą być skierowane lub nie i mogą posiadać krotności wyrażające zależności ilościowe pomiędzy pojęciami. W przedstawionym przykładzie, asocjacja między pojęciami „Patient" i „Appointment" oznacza, że dla danego pacjenta może istnieć wiele wizyt lekarskich a każda wizyta dotyczy tylko jednego pacjenta. Ponadto, każda wizyta powiązana jest z historią choroby („Medical record"). To ostatnie pojęcie reprezentuje m.in. wyniki badania przeprowadzonego podczas danej wizyty.

Drugim typem relacji jest relacja wyrażająca przynależność atrybutu do pojęcia typu „Concept". Notacja jest podobna do relacji agregacji w języku UML. W przykładzie przedstawionym na Rysunek 13, każde z pojęć posiada atrybuty typów prostych, np. dla pojęcia „Patient" są to: „Name", „Birth date" oraz „Social security number". Sposób graficznej reprezentacji atrybutów jako osobnych elementów wynika z faktu, że mogą być one powiązane relacjami nie tylko z pojęciami typu „Concept" lecz również z pojęciami reprezentującymi widoki danych. Pojęcia tego typu stanowią opis dziedziny aplikacji i zostały opisane w kolejnym podrozdziale.

Tak zdefiniowany model dziedziny problemu stanowi abstrakcję danej dziedziny biznesowej – jej kanoniczny model danych. Poziom szczegółowości modelu powinien wynikać z zakresu danych jakie będą przetwarzane przez docelowy system. Model taki jest zazwyczaj tworzony

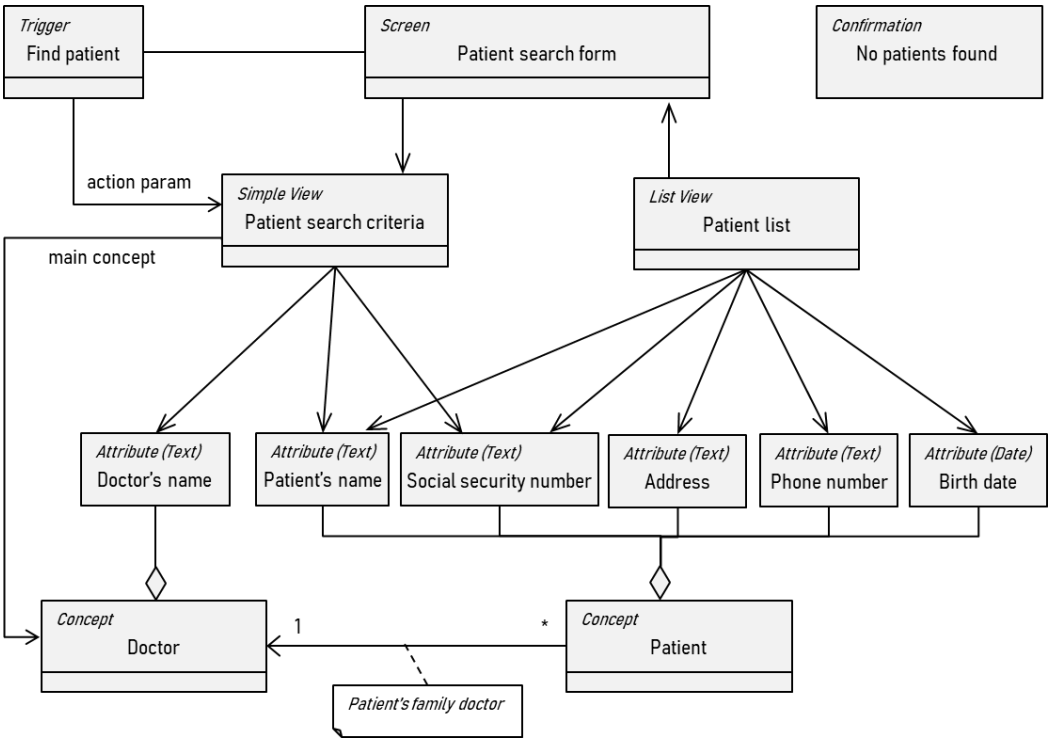
niezależnie od logiki aplikacji dla systemu. Ostatecznie, jednak, musi być on powiązany z elementami dziedziny aplikacji, gdyż definiuje zakres danych możliwych do wyświetlenia użytkownikowi ale nie definiuje gdzie, kiedy, i w jakim zestawieniu dane te mają być zaprezentowane.

3.2.2 Dziedzina aplikacji

Istotnym elementem każdej aplikacji jest jej warstwa prezentacji, zwana również interfejsem użytkownika (ang. User Interface – UI). Warstwa ta określa w jakiej formie dane zawarte w kanonicznym modelu danych prezentowane są użytkownikom i jest istotną częścią specyfikacji wymagań. Język RSL udostępnia specjalne typy pojęć dziedzinowych służących do modelowania elementów interfejsu użytkownika: „Screen", „Message", „Confirmation" oraz „Trigger". Podobnie jak w przypadku typu „Concept", wymienione typy są zdefiniowane za pomocą stereotypów, co oznacza, że ich zbiór może być łatwo rozszerzany w miarę potrzeb.

Pojęcia typu „Screen" reprezentują kontenery (ekrany), które mogą zawierać dowolne zestawy danych dziedzinowych oraz kontrolki. Ekrany służą zarówno do prezentacji danych użytkownikom jak również do ich wprowadzania bądź edycji. W przypadku graficznych interfejsów użytkownika, pojęcia typu „Screen" mają zazwyczaj postać okien aplikacji. Zakres danych prezentowanych w ramach ekranu jest definiowany za pomocą widoków danych, o czym będzie mowa poniżej.

Ekrany mogą zawierać elementy interaktywne, które służą do wyzwalania przez użytkowników akcji systemu, zgodnie ze zdefiniowaną logika aplikacji. W przypadku interfejsów graficznych mają one zazwyczaj postać przycisku lub ikony. Do modelowania takich elementów służą pojęcia typu „Trigger" – wyzwalacze akcji. Każdy ekran może zawierać dowolną liczbę wyzwalaczy, co jest modelowane za pomocą asocjacji pomiędzy pojęciami typu „Screen" i „Trigger". Zostało to zilustrowane na Rysunek 14, gdzie pojęcie „Patient search form" typu „Screen" powiązane jest z pojęciem „Find patient" typu „Trigger".



Rysunek 14 – Model dziedziny aplikacji

Oprócz pojęć typu „Screen”, RSL oferuje dwa dodatkowe typy pojęć reprezentujące elementy interfejsu użytkownika służące do prezentowania informacji: „Message” oraz „Confirmation”. Pierwsze z nich to prosty komunikat informujący użytkownika o jakimś zdarzeniu w systemie, np. o pomyślnym zapisaniu danych w systemie lub wprowadzeniu niepoprawnej wartości przez użytkownika. Komunikat taki powiązany jest z wyzwalaczem, za pomocą którego użytkownik, po zapoznaniu się z komunikatem, przechodzi do dalszej pracy z systemem. W przypadku interfejsu graficznego, komunikat taki ma zazwyczaj postać okna modalnego zawierającego tekst komunikatu (np. „Dodano nowego pacjenta”) oraz przycisk zamykający to okno (np. „OK” lub „Zamknij”). Pojęcie typu „Confirmation” również reprezentuje komunikat, jednak wymaga od użytkownika dokonania wyboru jednej z prezentowanych razem z komunikatem opcji (wyzwalaczy). Opcji może być wiele a wybór konkretnej z nich determinuje dalszą interakcję aktora z systemem – różną dla różnych opcji. W przypadku interfejsów graficznych, jest to zazwyczaj okno modalne z komunikatem (np. „Nie znaleziono szukanego pacjenta. Czy chcesz zmienić kryteria wyszukiwania?”) oraz zestawem przycisków do wyboru (np. „Tak”, „Nie”). Przykład pojęcia typu „Confirmation” pokazany jest na Rysunek 14. W przypadku obu typów komunikatów, ich treść specyfikowana jest jako atrybut pojęcia i nie jest widoczna na diagramie pojęć. W przypadku „Confirmation”, opcje (wyzwalacze) prezentowane razem

z komunikatem określane są w scenariuszu przypadku użycia poprzez utworzenie alternatywnych ścieżek przebiegu scenariusza.

Aby model dziedzinowy był kompletny, pojęcia reprezentujące elementy interfejsu użytkownika muszą być powiązane z pojęciami z dziedziny problemu. Dotyczy to przede wszystkim pojęć typu „Screen", gdyż ich powiązania z elementami modelu danych określają zawartości ekranów prezentowanych użytkownikom. Powiązanie to odbywa się za pośrednictwem widoków danych – pojęć typu „Simple View" oraz „List view". Widoki danych wskazują na atrybuty pojęć typu „Concept". Na diagramie (Rysunek 14) jest to wyrażone za pomocą asocjacji skierowanych od widoku danych do atrybutu. Pojedynczy widok danych może wskazywać na dowolną liczbę atrybutów przynależnych do jednego lub wielu różnych powiązanych ze sobą pojęć typu „Concept". W tym drugim przypadku, widok danych musi wskazywać pojęcie główne za pomocą relacji „main concept". Pozwala to określić właściwe relacje ilościowe danych reprezentowanych przez widok danych, co jest istotne w przypadku pojęć biznesowych powiązanych z krotnością jeden do wielu lub wiele do wielu. Relacja "main concept" jest rozszerzeniem języka RSL, dodanym w celu ułatwienia implementacji transformacji „RSL to Java". W tym samym celu przyjęto ograniczenie, że widok danych może wskazywać atrybuty należące do pojęcia „main concept" oraz pojęć bezpośrednio z nim powiązanych.

Formę prezentacji danych na ekranach aplikacji określa typ widoku danych. „Simple View" prezentuje wartości atrybutów dla pojedynczej instancji pojęcia „main concept" oraz wartości atrybutów dla jednej lub wielu instancji pojęć powiązanych z „main concept", w zależności od krotności między tymi pojęciami. Wartości atrybutów prezentowane są w postaci odpowiednich elementów interfejsu użytkownika wyświetlanych na ekranie. W przypadku „List View", dane prezentowane są w postaci listy, gdzie atrybuty definiują jej kolumny a w wierszach prezentowane są wartości atrybutów dla różnych instancji pojęć zawierających te atrybuty. W przykładzie na Rysunek 14, widok danych „Patient list" jest listą prezentującą dane pacjentów w pięciu kolumnach. Pojęcie „Patient" może zawierać dowolną liczbę atrybutów lecz akurat pięć wskazywanych przez „Patient list" jest istotnych w kontekście ekranu „Patient search form".

Przedstawione typy widoków danych rozszerzają język RSL poprzez nadanie pojęciom dziedzinowym odpowiednich stereotypów i określenie ich semantyki. W ten sam sposób można rozszerzać język o inne potrzebne w konkretnych zastosowaniach typy widoków (np. widok

reprezentujący strukturę drzewiastą czy widok typu master-details). W niniejszej pracy ograniczymy się do dwóch przedstawionych typów widoków.

Widoki danych, jak wspomniano powyżej, mogą być powiązane z ekranami („Screen"). Każdy ekran może być powiązany z dowolną liczba widoków danych, zarówno „Simple view" jak i „List view". Powiązanie takie wyraża się za pomocą asocjacji, jak zaprezentowano na Rysunek 14. Kierunek asocjacji ma istotne znaczenie, gdyż oznacza kierunek przepływu danych – z lub do ekranu.

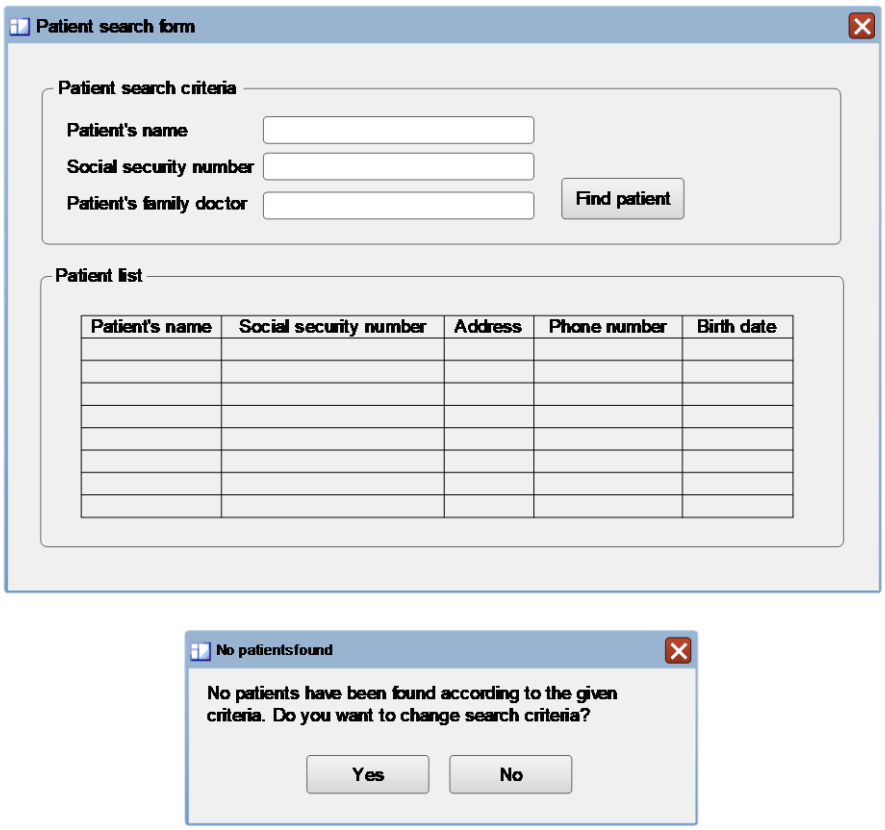
Relacja skierowana od ekranu do widoku danych (jak w przypadku „Patient search form" i „Patient search criteria" na Rysunek 14) jest to relacja typu „Input" i oznacza, że system oczekuje od użytkownika wprowadzenia danych zdefiniowanych przez dany widok. Dane te będą dalej przetwarzane przez system zgodnie ze zdefiniowaną logiką aplikacji. Dla zaprezentowanego przykładu, wartości wprowadzone przez użytkownika dla trzech atrybutów wskazywanych przez widok „Patient search criteria" będą wykorzystane jako kryteria wyszukiwania pacjentów. W przypadku interfejsu graficznego, każdy z atrybutów jest renderowany na ekranie w postaci elementów graficznych umożliwiających wprowadzanie wartości tych atrybutów.

Relacja skierowana od widoku danych do ekranu (jak w przypadku „Patient list" i „Patient search form") jest to relacja typu „Present" i oznacza, że zestaw danych zdefiniowany przez dany widok jest prezentowany użytkownikowi na ekranie do odczytu, bez możliwości edycji. W przypadku interfejsu graficznego, każdy z atrybutów wskazywanych przez widok danych jest renderowany na ekranie w postaci elementów graficznych tylko do odczytu, wraz z wartościami tych atrybutów dla konkretnej instancji pojęcia biznesowego. W przypadku „Simple view" jest to jedna instancja pojęcia „main concept" a w przypadku „List view" może to być wiele instancji. Dane do wyświetlenia muszą być wcześniej pobrane przez system ze źródła danych, np. bazy danych, pliku czy za pomocą web service'u. W przedstawionym przykładzie, widok „Patient list" będzie prezentowany na ekranie „Patient search form" jako lista pacjentów, gdzie dla każdego pacjenta zaprezentowany będzie zestaw pięciu atrybutów wskazywanych przez widok danych „Patient list". Na podstawie przedstawionego diagramu pojęć możemy się domyślać, że lista ta będzie wynikiem wyszukiwania pacjentów zgodnych z kryteriami wyszukiwania („Patient search criteria"), jednak nie wynika to bezpośrednio z modelu dziedzinowego lecz ze specyfikacji logiki aplikacji wyrażonej scenariuszami przypadków użycia (patrz rozdział 3.3).

Jeżeli asocjacja między ekranem i widokiem danych jest dwukierunkowa (brak grotów lub groty w obu kierunkach) jest to relacja typu „Modify" i oznacza, że wartości atrybutów pobrane przez system są prezentowane użytkownikowi, który może je następnie edytować.

W przypadku relacji między ekranem i widokiem danych, istotne znaczenie ma także typ atrybutów wskazywanych przez widok danych. Zależnie od typu renderowane są odpowiednie elementy interfejsu użytkownika, np. dla atrybutu typu „Text" może być wygenerowane proste pole tekstowe a dla atrybutu typu „True or False" – pole wyboru typu checkbox.

Ekrany i widoki danych mogą być również powiązane z wyzwalaczami (pojęcia typu „Trigger"). Ekrany, oprócz widoków danych, mogą również zawierać elementy umożliwiające użytkownikom wyzwalanie określonych akcji. Jest to modelowane za pomocą asocjacji między tymi elementami. Rysunek 14 przedstawia przykład takiej relacji: ekran „Patient search form" zawiera wyzwalacz „Find patient". Wyzwalacz uruchamia zazwyczaj jakąś operację na danych powiązanych z ekranem, do którego należy. Aby precyzyjnie wskazać, którego zbioru danych dotyczy ta operacja, należy dodatkowo utworzyć asocjację „action param" pomiędzy wyzwalaczem a widokiem danych. Ma to szczególne znaczenie w przypadku, gdy ekran powiązany jest z wieloma widokami danych. Należy wtedy jednoznacznie wskazać które dane będą przekazane do przetwarzania w ramach wyzwalanej operacji. W prezentowanym przykładzie, wyzwalacz „Find patient" wskazuje na widok danych „Patient search criteria", co znaczy, że wartości atrybutów wprowadzone w tym przypadku przez użytkownika będą parametrem operacji wyszukiwania pacjentów. Informacja jaką niesie taka relacja nie jest niezbędna z punktu widzenia osoby czytającej specyfikacje wymagań w języku RSL, gdyż może ona być wywnioskowana z kontekstu scenariusza przypadku użycia. Precyzyjne wskazanie parametru operacji znacząco ułatwia, natomiast, definicję semantyki translacyjnej i generację kodu a także pozwala na większą swobodę jeśli chodzi o kolejność zdań w scenariuszu.

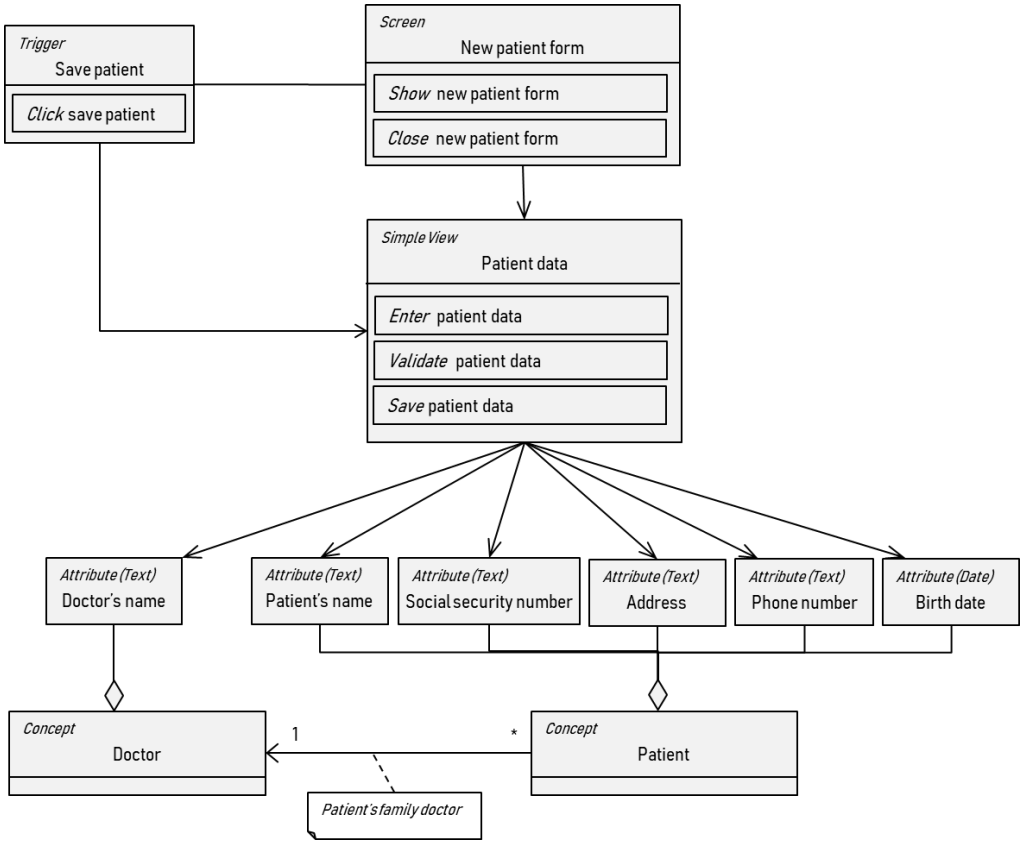


Rysunek 15 – Semantyka elementów interfejsu użytkownika

Semantykę opisanych powyżej konstrukcji najlepiej jest wyjaśnić dokonując ich translacji na konkretne elementy interfejsu użytkownika. Rysunek 15 przedstawia fragment interfejsu graficznego, który odpowiada modelowi dziedzinowemu przedstawionemu na Rysunek 14. Mamy tu okno odpowiadające pojęciu „Patient search form" typu „Screen" oraz komunikat typu „Confirmation". W górnej części okna widoczne są pola tekstowe umożliwiające użytkownikowi wprowadzenie wartości atrybutów zgodnie z widokiem danych „Patient search criteria". Widoczny jest też przycisk „Find patient" wyzwalający operację wyszukiwania pacjentów na podstawie wprowadzonych kryteriów. Dolna część okna zawiera listę pacjentów będąca realizacją widoku „Patient list". Należy jednak podkreślić, że modele w języku RSL są niezależne od konkretnej technologii interfejsu użytkownika. Typ i układ poszczególnych elementów również nie wynikają bezpośrednio z modelu RSL. Tego typu szczegóły technologiczne określane są podczas transformacji modelu wymagań w model projektowy oraz kod, co zostanie opisane w dalszej części pracy.

3.2.3 Stwierdzenia dziedzinowe

Przedstawione powyżej pojęcia pozwalają zamodelować strukturę dziedziny problemu oraz dziedziny aplikacji. Aby jednak model dziedzinowy był kompletny potrzebne są dodatkowe elementy pozwalające wyrażać cechy behawioralne pojęć dziedzinowych. Przez cechy behawioralne rozumiemy czynności jakie system lub aktorzy mogą wykonywać na encjach reprezentowanych przez pojęcia dziedzinowe. Czynności te można utożsamiać z operacjami klas w modelowaniu obiektowym. Pozostają one w ścisłym związku z logiką aplikacji zdefiniowaną dla danego systemu.



Rysunek 16 – Pojęcia dziedzinowe z widocznymi stwierdzeniami dziedzinowymi

W języku RSL, cechy behawioralne pojęć dziedzinowych modelowane są za pomocą tzw. stwierdzeń dziedzinowych (ang. domain statements). Stwierdzenia dziedzinowe mają postać fraz czasownikowych – składają się z czasownika oraz frazy rzeczownikowej. Fraza rzeczownikowa reprezentuje encję dziedzinową a czasownik akcję wykonywaną na tej encji. Szczegółowa budowa stwierdzenia dziedzinowego – ich składnia abstrakcyjna – zostanie przedstawiona w rozdziale 4.

Rysunek 16 przedstawia notację dla pojęć z widocznymi stwierdzeniami dziedzinowymi. Ekran „New patient screen” posiada dwa stwierdzenia dziedzinowe: „Show new patient screen” oraz „Close new patient screen”, które oznaczają, że na ekranie mogą być wykonane dwie akcje, odpowiednio: wyświetlenie ekranu użytkownikowi oraz zaprzestanie wyświetlania ekranu. Dane powiązane z ekranem, reprezentowane przez widok „Patient data” mogą być wprowadzone („Enter patient data”), zwalidowane („Validate patient data”) oraz zapisane („Save patient data”).

Każdy typ pojęcia dziedzinowego może zawierać stwierdzenia dziedzinowe. Stwierdzenia te mogą zawierać dowolne czasowniki, przy czym pewne grupy synonimicznych czasowników mogą być zarezerwowane do wyrażania akcji typowych dla większości standardowych aplikacji. Akcje takie mogą być mapowane na określone typowe operacje w kodzie aplikacji. Takie typowe operacje, wraz z wyrażającymi je czasownikami, zdefiniowane są dla niektórych typów pojęć dziedzinowych. Tabela 1 przedstawia predefiniowane operacje wraz z informacją o typach pojęć do których mają zastosowanie, semantyce tych operacji w kontekście wymienionych pojęć oraz zbiorem synonimicznych czasowników stanowiących słowa kluczowe, które użyte w stwierdzeniu dziedzinowym, wyrażają daną operację.

Tabela 1 – Predefiniowane operacje dla poszczególnych typów pojęć dziedzinowych

Operacja	Typy pojęć	Semantyka operacji	Słowa kluczowe
Show	“Screen”, “Message”, “Confirmation”	Zaprezentuj element (wraz z powiązanym widokiem danych) aktorowi za pomocą interfejsu użytkownika	“show”, „display”, „present”, „pop up”
Close	“Screen”	Zaprzestań prezentowania elementu aktorowi	“close”, „shut”, „remove”
Refresh	“Screen”	Odśwież prezentowany element w celu wyświetlenia uaktualnionego widoku danych	“refresh”, „renew”, „repaint”, „update”
Select	“Trigger”	Wyzwolenie akcji systemu: obsłuż zdarzenie powiązane z danym wyzwalaczem	“select”, „choose”, „press”, „click”, „push”

Create	“Simple view”, “List view”, “Concept”	Zapisz dane w systemie	“create”, „save”, „add”, „write”, „persist”
Read	“Simple view”, “List view”, “Concept”	Pobierz dane z systemu	“read”, „get”, „fetch”, „build”, „retrieve”, „search”
Update	“Simple view”, “List view”, “Concept”	Zastąp istniejące dane nowymi wartościami	“update”, „modify”, „edit”, „override”, „change”
Delete	“Simple view”, “List view”, “Concept”	Usuń dane z systemu	“delete”, „remove”, „erase”, „destroy”, „purge”
Validate	“Simple view”, “List view”, “Concept”	Wykonaj operację weryfikacji danych	“validate”, „verify”, „examine”, „check”, „inspect”

Typ operacji przypisywany jest stwierdzeniu dziedzinowemu poprzez nadanie mu odpowiedniego stereotypu. Stereotyp z kolei określany jest na podstawie słowa kluczowego[20]. Nie jest to część specyfikacji języka RSL i może być dowolnie konfigurowane w narzędziu ReDSeeDS. Dla wszystkich wymienionych powyżej operacji (oprócz „Validate”), możliwe jest wygenerowanie ich implementacji w postaci kodu, gdyż są to standardowe operacje nie wymagające zdefiniowania zaawansowanej logiki biznesowej w postaci algorytmów czy reguł. Niezbędne jest jednak określenie szczegółów architektonicznych, np. czy w przypadku operacji „Read” dane mają być pobrane z relacyjnej bazy danych, pliku lokalnego czy może poprzez wywołanie operacji web service’u. Szczegóły takie mogą być zawarte w transformacji lub być wyrażone w języku RSL po odpowiednim jego rozszerzeniu. W przypadku „Validate” lub innej operacji, nie predefiniowanej, do wygenerowania kodu niezbędna jest bardziej złożona specyfikacja logiki takiej operacji. Opisywana w niniejszej pracy wersja języka RSL nie udostępnia konstrukcji umożliwiających specyfikację logiki operacji. Przedstawiona

[20] Oprócz zdefiniowanych słów kluczowych, typ operacji może być wyznaczany również na podstawie innych wyrazów bliskoznacznych, które wykazują określone podobieństwo semantyczne. Jest to możliwe dzięki zastosowaniu w narzędziu ReDSeeDS słownika WordNet [148].

transformacja nie uwzględnia generacji kodu dla operacji dziedzinowych. Problem generacji kodu logiki biznesowej na podstawie rozszerzonego języka RSL, zwanego RSL-DL, został opisany w pracy doktorskiej Rybińskiego [53].

3.3 Specyfikowanie wymagań funkcjonalnych

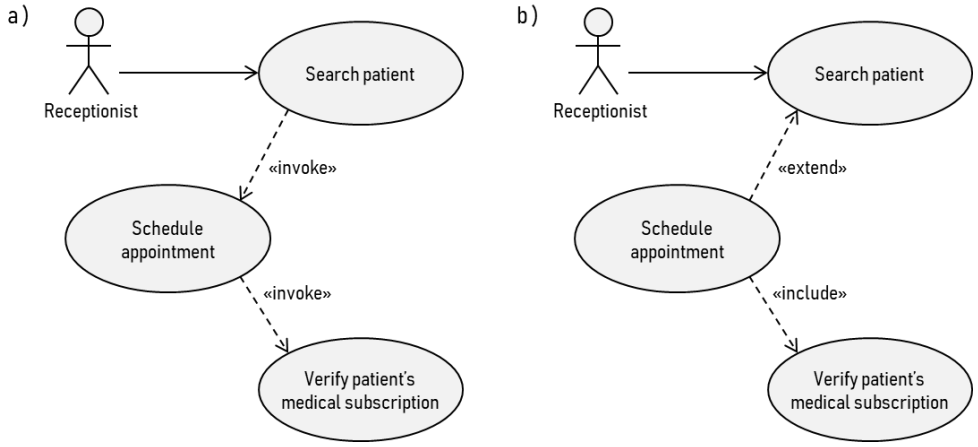
Wymagania funkcjonalne w języku RSL definiowane są za pomocą przypadków użycia. Model przypadków użycia w języku RSL jest bardzo zbliżony do tego znanego z języka UML. Wprowadza on jednak pewne zmiany i rozszerzenia mające na celu zwiększenie precyzji i jednoznaczności w stosunku do niejednoznacznej semantyki języka UML w zakresie przypadków użycia. Zmiany te polegają między innymi na wprowadzeniu nowego typu relacji między przypadkami użycia – „invoke". Zasadniczym rozszerzeniem jest jednak sposób reprezentacji zachowania systemu w interakcji z aktorami w ramach przypadków użycia. Język RSL wprowadza precyzyjny język specyfikacji scenariuszy przypadków użycia, który umożliwia powiązanie scenariuszy z modelem dziedzinowym. Język ten oparty jest na prostych zdaniach w ograniczonym języku naturalnym (ang. constrained natural language), których gramatyka zapewnia zgodność ze stwierdzeniami dziedzinowymi przedstawionymi w poprzednim podrozdziale oraz z logiką relacji „invoke". Pozwala to na opisywanie logiki aplikacji w sposób spójny i jednoznaczny, co ma znaczenie zarówno z punktu widzenia odbiorców specyfikacji wymagań, jak i z punktu widzenia automatycznych transformacji modeli wymagań.

3.3.1 Przypadki użycia systemu

Przypadki użycia, spopularyzowane w latach 90-tych przez Ivara Jacobsona [128], są dzisiaj powszechnie używaną techniką specyfikacji wymagań funkcjonalnych [129] [130] [131] [132] [133] [134] [135]. Również w języku RSL przypadki użycia stanowią najważniejszy element służący do opisu logiki aplikacji tworzonego systemu. Przypadek użycia rozumiany jest jako kompletna jednostka funkcjonalności systemu. Składa się ze scenariuszy (co najmniej jednego – głównego), które opisują sekwencje interakcji między zewnętrznym aktorem a systemem. Wynikiem tych interakcji jest osiągnięcie jakiegoś wymiernego efektu przez aktora, przy czym przypadek użycia może zakończyć się również porażką w osiąganiu zamierzonego celu (scenariusze alternatywne).

Rysunek 17a przedstawia przykładowy diagram przypadków użycia, na którym widoczne są dwa typy relacji: relacja typu „use” pomiędzy aktorem i przypadkiem użycia oraz relacja typu „invoke” pomiędzy dwoma przypadkami użycia.

Relacja typu „use” oznacza, że dany aktor inicjuje wykonanie określonego przypadku użycia z zamiarem osiągnięcia celu tego przypadku użycia. Jeżeli przypadek użycia jest celem relacji typu „use”, oznacza to, że reprezentowana przez niego funkcjonalność jest bezpośrednio dostępna dla danego aktora poprzez interfejs użytkownika aplikacji (wyzwalacz uruchamiający dany przypadek użycia). W przypadku interfejsu graficznego jest to zazwyczaj opcja w głównym menu aplikacji lub przycisk na stronie głównej.

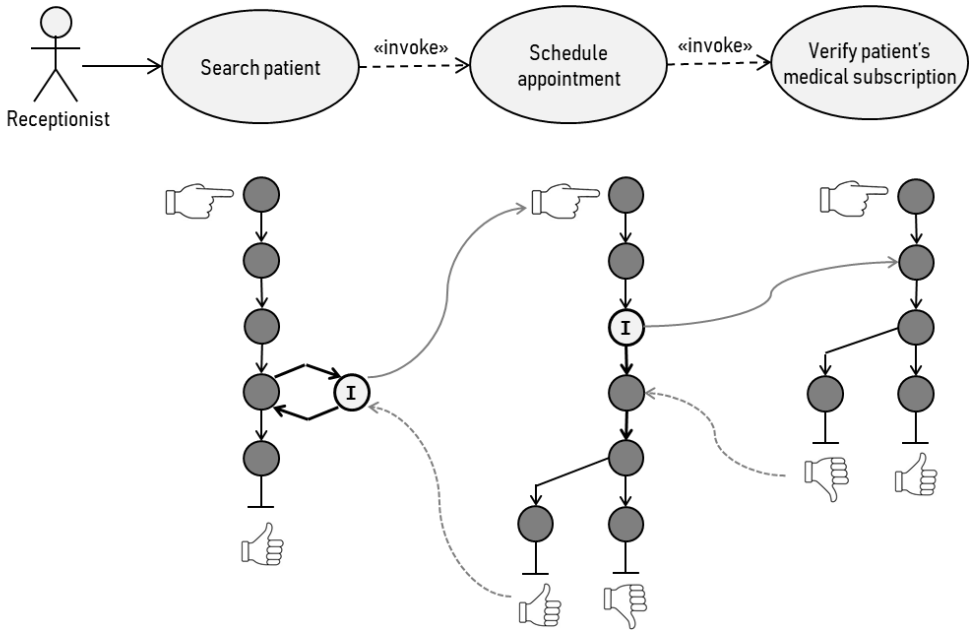


Rysunek 17 – Przykładowy diagram przypadków użycia: (a) z relacjami typu „invoke” w języku RSL, (b) z relacjami typu „include” i „extend” w języku UML

Relacja typu „invoke” reprezentowana jest za pomocą przerywanej strzałki ze stereotypem „invoke”, skierowanej od wywołującego przypadku użycia w stronę przypadku wywoływanego. Relacja „invoke” zastępuje znane z języka UML relacje „include” oraz „extend”. Zmiana reprezentacji graficznej w stosunku do języka UML jest niewielka (porównanie obu notacji przedstawia Rysunek 17), natomiast bardziej istotna jest zmiana semantyki tych relacji. Celem zmiany w języku RSL było wprowadzenie większej precyzji i jednoznaczności w opise przepływu kontroli (logiki działania modelowanej aplikacji) na poziomie przypadków użycia i ich scenariuszy. Efektem tych zmian jest ujednolicenie notacji oraz, co szczególnie istotne z punktu widzenia niniejszej pracy – uproszczenie reguł translacji i generacji kodu logiki aplikacji.

Semantyka relacji „include” zdefiniowana w specyfikacji języka UML [79] nie budzi większych wątpliwości – przypadek użycia wskazywany przez tę relację (przypadek włączany) jest w całości wykonywany w pewnym miejscu przypadku bazowego, po czym kontynuowane jest wykonanie przypadku bazowego. Można to porównać do wywołania podprogramu, po wykonaniu którego, sterowanie wraca do miejsca wywołania. Specyfikacja UML nie definiuje jednak punktu włączenia zewnętrznego przypadku użycia w ramach przypadku bazowego. W przypadku relacji „extend”, język UML definiuje wprawdzie punkty rozszerzeń, jednak ich semantyka jest nieprzydatna z punktu widzenia generacji kodu. Jest tak między innymi z tego powodu, że zachowanie zdefiniowane dla przypadku rozszerzającego może przeplatać się dowolnie z zachowaniem przypadku rozszerzanego, co można porównać do użycia instrukcji skoku „goto” w kodzie programu. Ponadto, nie jest jasne jak powiązać punkty rozszerzeń z opisem zachowania danego przypadku użycia a tym samym – jak precyzyjnie zdefiniować przepływ kontroli w ramach przypadków użycia powiązanych relacją „extend”. Wyczerpującą dyskusję na ten temat niejednoznaczności relacji „include” i „extend” można znaleźć w [136] [137] [138] [139] [140] [141].

Z powyższych powodów, w języku RSL relacje „include” i „extend” zostały zastąpione relacją „invoke”, która została zdefiniowana wystarczająco precyzyjnie, zarówno na poziomie modelu przypadków użycia jak i scenariuszy, aby na jej podstawie generować kod logiki aplikacji. Użycie relacji „invoke” można przyrównać do wywołań funkcji w językach programowania.



Rysunek 18 – Przepływ kontroli dla relacji typu „invoke”

W celu objaśnienia semantyki relacji „invoke” posłużymy się przykładowym modelem przypadków użycia przedstawionym na Rysunek 18. Przedstawione przypadki użycia powiązane są dwiema relacjami „invoke”. Jedna z nich (pomiędzy przypadkami „Search patient” i „Schedule appointment”) jest częściowo ekwiwalentna względem relacji „extend” a druga (pomiędzy przypadkami „Schedule appointment” i „Verify patient’s medical subscription”) – względem relacji „include”. Poglądowo zilustrowane scenariusze przedstawiają przepływ kontroli w ramach przypadków użycia z uwzględnieniem miejsc wywołań (oznaczone za pomocą „I”) odpowiadających relacjom „invoke” pomiędzy przypadkami.

Możemy tu wyróżnić dwie sytuacje. Pierwsza dotyczy relacji „invoke” pomiędzy przypadkiem użycia „Search patient” (przypadek wywołujący) a przypadkiem „Schedule appointment” (przypadek wywoływany). Jest ona częściowo ekwiwalentna względem relacji „extend”. Miejsce wywołania „Schedule appointment” jest powiązane z konkretnym krokiem scenariusza „Search patient”, w którym sterowanie jest po stronie aktora. W naszym przykładzie może to być na przykład ekran z listą pacjentów znalezionych przez system na podstawie podanych kryteriów wyszukiwania. Aktor może w tym miejscu scenariusza zdecydować czy wywołać dodatkową funkcjonalność „Schedule appointment” (umówienie wizyty dla wybranego pacjenta), czy kontynuować wykonywanie pierwotnego przypadku użycia. Wywołanie „Schedule appointment” odbywa się poprzez wykonanie pierwszego kroku scenariusza wywoływanego przypadku użycia. Zawsze w takim przypadku jest to interakcja aktor-system, np. wybranie przycisku „Schedule appointment” dla zaznaczonego pacjenta. Przepływ kontroli jest tym samym przekazywany do przypadku wywoływanego i jest kontynuowany zgodnie z jego scenariuszem. Po osiągnięciu punktu końcowego scenariusza przypadku wywoływanego (niezależnie czy wykonanie zakończyło się sukcesem czy porażką) sterowanie wraca do scenariusza „Search patient”, do kroku powiązanego z miejscem wywołania. Wracamy tym samym do punktu wyjścia – aktor może kontynuować wykonanie dalszej części scenariusza „Search patient” lub ponownie wywołać „Schedule appointment” (np. umówienie kolejnej wizyty). Podczas przekazywania sterowania z powrotem do przypadku wywołującego, przekazywany jest rezultat wykonania przypadku wywoływanego, co w niektórych sytuacjach może wpłynąć na dalszy przebieg przypadku wywołującego.

Druga sytuacja dotyczy relacji „invoke” pomiędzy przypadkiem użycia „Schedule appointment” (przypadek wywołujący) a przypadkiem „Verify patient’s medical subscription”

(przypadek wywoływany). Jest ona częściowo ekwiwalentna względem relacji „include". Miejsce wywołania jest w takiej sytuacji bezpośrednio włączone w ścieżkę scenariusza. Gdy przepływ sterowania dotrze do takiego miejsca wywołania, sterowanie zawsze i bezwarunkowo przekazywane jest do przypadku wywoływanego. Aktor nie ma wpływu na wywołanie dodatkowej funkcjonalności – nie ma potrzeby wykonywania interakcji aktor-system. Jeżeli scenariusz wywoływanego przypadku użycia zaczyna się taką interakcją, jest ona pomijana w momencie przekazywania kontroli z przypadku wywołującego. Po osiągnięciu punktu końcowego scenariusza przypadku wywoływanego, sterowanie wraca do scenariusza pierwotnego – do kroku następującego po miejscu wywołania. W przedstawionym przykładzie, zawsze przy umawianiu wizyty lekarskiej („Schedule appointment"), następuje automatyczna weryfikacja czy dany rodzaj wizyty jest objęty abonamentem pacjenta. Rezultat wykonania przypadku wywołanego ma, zatem, znaczenie dla dalszego przebiegu przypadku „Schedule appointment" – przykładowo, gdy abonament nie obejmuje danej wizyty, wymagana jest dodatkowa płatność za usługę medyczną.

Szczegółowa semantyka zdań stanowiących miejsca wywołania oraz innych typów zdań definiujących przepływ kontroli między przypadkami użycia zostanie przedstawiony w kolejnych dwóch podrozdziałach.

3.3.2 Typy zdań

Aby przypadki użycia mogły precyzyjnie wyrażać logikę budowanej aplikacji, język RSL definiuje różne typy zdań, stanowiących bloki budulcowe scenariuszy przypadków użycia. Są to zdania w ograniczonym języku naturalnym [142] [143]. W zależności od typu, zdania wyrażają akcje wykonywane przez aktora bądź system albo określają przepływ kontroli w ramach pojedynczego przypadku użycia oraz przypadków powiązanych. Tabela 2 przedstawia przykłady zdań w podziale na pozycję zdania w scenariuszu oraz podstawowy typ.

Tabela 2 – Przykłady zdań w podziale na pozycję zdania w scenariuszu oraz podstawowy typ

Pozycja zdania w scenariuszu	Podstawowy typ zdania	Przykłady
Zdania inicjujące	Precondition	Pre: A <param> patient </param> is selected

Treść scenariusza	SVO	Receptionist *enters* patient search criteria System *retrieves* patient list *according to* patient search criteria
	Condition	Cond: Patients found [1] Cond: No patients found [0]
	Invocation	Invoke: Schedule appointment
	Rejoin	Rejoin: Main scenario: Receptionist enters patient search criteria
Zdania końcowe	Final / Postcondition	Final: success Post: New appointment scheduled and saved

Przypadek użycia zaczyna się zawsze od zdania typu „Precondition", które określa warunek pod jakim przypadek użycia może być wykonany. Warunek może odnosić się m.in. do stanu systemu lub wykonania określonej akcji przez aktora. Spełnienie tego warunku oznacza, że przypadek użycia jest gotowy do wywołania. W przeciwnym razie, akcja inicjująca przypadek użycia będzie nieskuteczna lub system w ogóle nie dopuści do wykonania takiej akcji. Warunek zdefiniowany przez „Precondition" sprawdzany jest niezależnie od sposobu wywołania przypadku użycia – czy to bezpośrednio przez aktora czy w ramach relacji „invoke" (patrz Rysunek 18). W przypadku, gdy zdanie „Precondition" jest puste, przypadek użycia wykonywany jest zawsze po wykonaniu akcji inicjującej.

Przykład zdania „Precondition" w Tabela 2, określa warunek wstępny dla przypadku użycia „Schedule appointment" (patrz Rysunek 18). Aby zainicjować tę funkcję systemu, aktor (Receptionist) musi najpierw, w ramach przypadku „Search patient", wybrać z listy pacjenta, dla którego chce umówić wizytę. Dopóki tego nie uczyni, system nie pozwoli uruchomić funkcji umawiania wizyty, np. odpowiedni przycisk w graficznym interfejsie użytkownika będzie nieaktywny lub niewidoczny. Warunek jest wyrażony językiem naturalnym, przy czym została tutaj użyta para znaczników „<param> … </param>", które nie są częścią notacji RSL a zostały dodane w celu ułatwienia generacji kodu. Dokładne ich znaczenie zostało opisane w rozdziale 5.

Po zdaniu „Precondition" następuje zasadnicza treść scenariusza, która może się składać z szeregu zdań stanowiących możliwe kroki scenariusza. Mogą tu występować trzy

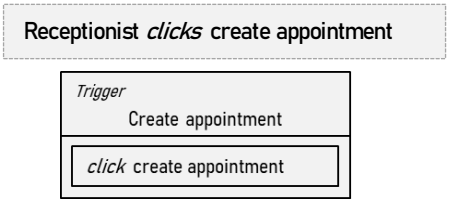
podstawowe typy zdań: zdania SVO, zdania warunkowe (Condition), zdania stanowiące miejsca wywołań innych przypadków użycia (Invocation) oraz zdania łączące scenariusze (Rejoin).

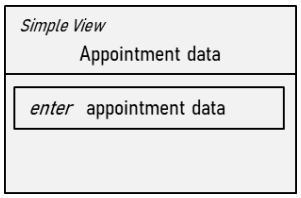
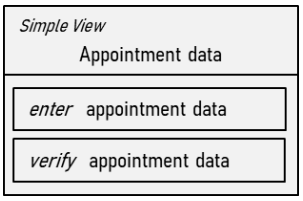
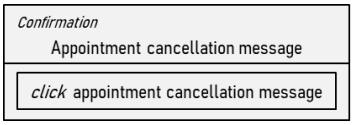
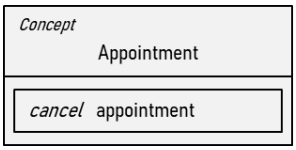
Zdania SVO wyrażają akcje podejmowane przez aktora lub przez system, zmierzające do realizacji celu przypadku użycia. Zdania te wyrażane są w języku naturalnym, ograniczonym do kilku prostych elementów składniowych: podmiotu (ang. Subject), orzeczenia (ang. Verb) oraz dopełnienia (ang. Object). Opcjonalnie, zdanie może zawierać dopełnienie dalsze (ang. Indirect object) poprzedzone przyimkiem (ang. Preposition). Przykłady dla obu wariantów pokazuje Tabela 2. Mimo prostej budowy, zdania SVO są wystarczające do wyrażania interakcji pomiędzy aktorem a systemem [66].

Zdania SVO są ściśle powiązane z elementami dziedzinowymi. Każde zdanie składa się z dwóch hiperłączy. Pierwszym hiperłączem jest podmiot, który wskazuje na aktora bądź element reprezentujący system i, tym samym, określa kto lub co jest wykonawcą akcji. Pozostała część zdania (orzeczenie wraz z dopełnieniem i ewentualnym dopełnieniem dalszym) jest hiperłączem do stwierdzenia dziedzinowego, które definiuje akcję wykonywaną na pojęciu dziedzinowym (patrz podrozdział 3.2.3).

W zależności od typu elementów wskazywanych przez hiperłącza, możemy wyróżnić kilka typów zdań SVO. Zestawienie wszystkich dopuszczalnych typów wraz z ich semantyką oraz przykładami prezentuje Tabela 3.

Tabela 3 – Semantyka poszczególnych typów zdań SVO

Typ zdania SVO	Semantyka	Przykład zdania wraz z powiązanym pojęciem dziedzinowym
Actor-to-Trigger	Akcja wykonywana przez aktora na elemencie typu Trigger w celu wyzwolenia akcji systemu.	

Actor-to-DataView	Akcja wykonywana przez aktora na elemencie reprezentującym widok danych (SimleView lub ListView), np. wprowadzenie lub edycja danych.	Receptionist *enters* appointment data 
System-to-DataView	Akcja wykonywana przez system na elemencie reprezentującym widok danych (SimpleView lub ListView), np. odczyt, zapis, edycja, aktualizacja lub walidacja danych.	System *verifies* appointment data 
System-to-Screen	Akcja wykonywana przez system na elemencie typu Screen, np. wyświetlenie, zamknięcie lub odświeżenie okna aplikacji.	System *shows* new appointment form 
System-to-Message lub System-to-Confirmation	Akcja wykonywana przez system, związana z komunikatem dla użytkownika (pojęcia typu Message lub Confirmation), np. wyświetlenie komunikatu lub okienka dialogowego.	System *shows* appointment cancellation message 
System-to-Concept	Akcja wykonywana przez system na obiekcie dziedzinowym (pojęcie typu Concept), np. odwołanie wizyty lub usunięcie pacjenta z systemu.	System *cancels* appointment 

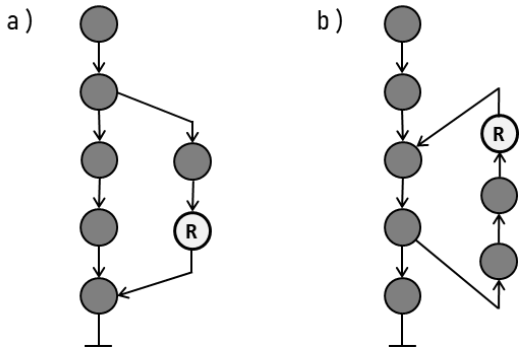
Hiperłącza w zdaniach SVO pozwalają utrzymywać ścisłą zależność między specyfikacją funkcjonalną a specyfikacją dziedziny. Edytor języka RSL w narzędziu ReDSeeDS utrzymuje tę zależność w sposób automatyczny. Podczas pisania zdań SVO, hiperłącza do istniejących

elementów dziedziny tworzone są automatycznie a jeżeli dane pojęcie dziedzinowe jeszcze nie istnieje, edytor tworzy je w odpowiednim pakiecie. Istotnie wspomaga to utrzymanie spójności modelu wymagań.

Oprócz zdań SVO wyrażających konkretne akcje, niezbędne są również zdania pozwalające precyzyjnie określać przepływ kontroli pomiędzy krokami scenariuszy. Podstawowym zdaniem z tej grupy jest zdanie warunkowe (Condition). Pełni ono funkcję warunkowego rozgałęzienia scenariusza. Gdy przepływ kontroli dotrze do zdania warunkowego, sprawdzany jest zawarty w nim warunek i jeśli jest on spełniony, przepływ kontroli przechodzi do zdania następującego po zdaniu warunkowym. Jeżeli warunek nie jest spełniony, sterowanie przechodzi do scenariusza alternatywnego. Przykłady zdań warunkowych zostały przedstawione w Tabela 2. Specyfikacja języka RSL nie narzuca (oprócz słowa kluczowego „Cond:") żadnej konkretnej struktury zdania warunkowego – warunek może być wyrażony w postaci dowolnego tekstu. Podobnie jak w przypadku zdania Precondition, mogą być tutaj używane predefiniowane znaczniki ułatwiające generację kodu. Przykładowo, w nawiasach kwadratowych mogą być podane wartości będące rezultatem sprawdzenia warunku lub opcje (wyzwalacze) prezentowane razem z komunikatem typu „Confirmation" (patrz 3.2.2). Sposób generowania kodu dla znaczników jest przedstawiony w rozdziale 5.

Kolejnym typem zdania sterującego przepływem kontroli jest Invocation. Zdania te wskazują w scenariuszu miejsce wywołania innego przypadku użycia, co zostało wstępnie wyjaśnione w podrozdziale 3.3.1. Więcej szczegółów wraz z przykładami przedstawia kolejny podrozdział.

Zdania typu Rejoin służą do łączenia scenariuszy w ramach jednego przypadku użycia. Zdanie Rejoin użyte w danym scenariuszu wskazuje konkretny krok w innym scenariuszu, do którego zostanie przekazane sterowanie w momencie, gdy dojdzie do zdania Rejoin. Jak pokazano na przykładzie w Tabela 2, zdanie Rejoin ma ściśle określoną strukturę – wskazuje nazwę scenariusza oraz konkretne zdanie SVO w tym scenariuszu, stanowiące punkt połączenia. Dzięki temu, możliwe jest łączenie scenariuszy rozgałęzionych przy użyciu zdania warunkowego, przy czym punkt połączenia może znajdować się zarówno poniżej jak i powyżej miejsca rozgałęzienia. Oba przypadki przedstawiono schematycznie na Rysunek 19.



Rysunek 19 – Przepływ kontroli dla zdania typu Rejoin z punktem połączenia zlokalizowanym poniżej (a) oraz powyżej (b) rozgałęzienia

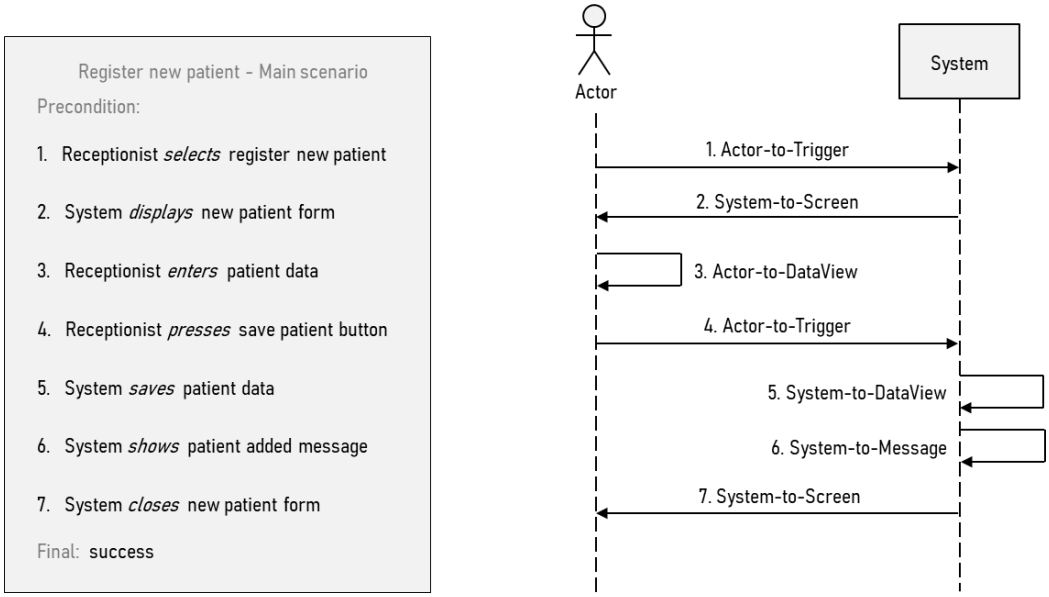
Każdy kompletny scenariusz musi być zakończony zdaniem końcowym Final/Postcondition. Zdanie to określa czy w wyniku wykonania danego scenariusza został osiągnięty cel przypadku użycia czy nie. Do określenia rezultatu służą dwa słowa kluczowe: „success" oraz „failure". Po osiągnięciu zdania końcowego, wykonanie przypadku użycia kończy się a sterowanie zostaje przekazane wraz z rezultatem do miejsca wywołania przypadku użycia. Przykładowe zdanie końcowe zaprezentowane w Tabela 2, dotyczy scenariusza przypadku użycia „Schedule appointment" i oznacza, że zakończył się on powodzeniem – wizyta lekarska została pomyślnie dodana. Dodatkowo, zdanie końcowe może zawierać warunek końcowy (ang. postcondition) określający stan systemu po wykonaniu przypadku użycia, niezależnie od osiągniętego rezultatu. Warunek końcowy może być wyrażony językiem naturalnym. Może również, podobnie jak zdanie Precondition, zawierać predefiniowane znaczniki, które mogą ułatwić automatyczne przetwarzanie modelu wymagań. Na potrzeby tej pracy nie zostały zdefiniowane żadne specjalne reguły wyrażania warunków końcowych.

3.3.3 Scenariusze

Każdy przypadek użycia może posiadać szereg scenariuszy. Określają one możliwe sekwencje akcji wykonywanych bądź to przez aktora, bądź przez system, prowadzące do pomyślnego osiągnięcia celu przypadku użycia lub kończących się niepowodzeniem. Jak opisano w poprzednim podrozdziale, język RSL definiuje różne typy zdań, które mogą być wykorzystane do tworzenia scenariuszy. Definicje zawarte w specyfikacji języka określają strukturę oraz znaczenie poszczególnych zdań. Specyfikacja nie definiuje, natomiast, reguł określających kolejność różnych typów zdań w scenariuszu oraz wynikającej z niej semantyki kontekstowej zdań. Reguły takie są niezbędne aby możliwa była bezpośrednia i precyzyjna translacja logiki aplikacji zawartej w scenariuszach na poprawne konstrukcje w języku

programowania dla określonego środowiska translacyjnego. Brak takich reguł w specyfikacji języka RSL wynika stąd, że (mimo, iż podstawowym założeniem projektowym była możliwość automatycznej transformacji modeli wymagań) nie był on tworzony z myślą o konkretnym środowisku translacyjnym. Z kolei, z punktu widzenia ludzi będących autorami lub odbiorcami specyfikacji wymagań, ścisłe reguły tworzenia scenariuszy są mniej istotne, gdyż dokładna ich interpretacja jest wynikiem wnioskowania pragmatycznego. Poniżej przedstawiono reguły tworzenia scenariuszy, przyjęte na potrzeby transformacji „RSL to Java" opisanej w tej pracy.

Kolejność zdań w scenariuszu oraz ich semantyka kontekstowa są powiązane z pojęciem *stanu dialogu*. Pojęcie to jest wyjaśnione na Rysunek 20, który przedstawia scenariusz główny dla przypadku użycia „Register new patient". Przedstawiony jest tam również diagram sekwencji, który ilustruje jak zmienia się stan dialogu podczas wykonywania kolejnych akcji reprezentowanych przez zdania SVO różnych typów.



Rysunek 20 – Stanu dialogu – zdania SVO

W danym momencie dialog może być w jednym z dwóch stanów: „aktor" lub „system". Aktualny stan determinuje wykonawcę kolejnego kroku scenariusza. Jeżeli aktualnym stanem dialogu jest „aktor", kolejnym zdaniem będzie zdanie „Actor-to-*". Dla stanu „system", kolejnym zdaniem zawsze będzie „System-to-*". Zmianę stanu dialogu mogą powodować określone typy zdań SVO, w których orzeczenie i dopełnienie wskazują odpowiednio

określony typ akcji na określonym pojęciu dziedzinowym. W niektórych przypadkach istotny jest również kontekst scenariusza.

Zmiana stanu z „aktor" na „system" jest następstwem zdania Actor-to-Trigger w przypadku, gdy oznacza ono uruchomienia wyzwalacza, po którym spodziewana jest reakcja systemu. W przykładzie na Rysunek 20, takimi zdaniami są: „Receptionist *selects* register new patient" oraz „Receptionist *presses* save patient button". Aby zdanie Actor-to-Trigger zmieniło stan dialogu, dopełnienie musi odnosić się do pojęcia typu Trigger a w orzeczeniu musi być użyty jeden z synonimicznych czasowników, które mapują się na operację typu „Select" (patrz Tabela 1 w rozdziale 3.2.3). W przypadku innych czasowników, zdanie Actor-to-Trigger nie zmienia stanu dialogu, np. „Receptionist *hovers over* save patient button". Nawet, jeżeli oczekiwanym efektem akcji *hover over* jest podświetlenie przycisku, jest to zachowanie związane z przyjętym stylem interfejsu użytkownika a nie z logiką aplikacji i jest w scenariuszu zbędne. Tego typu zachowanie interfejsu użytkownika mogłoby być, ewentualnie, definiowane za pomocą odpowiednich atrybutów pojęcia typu Trigger.

Oprócz zmiany stanu dialogu, efektem użycia w scenariuszu zdania Actor-to-Trigger jest powiązanie występującego w tym zdaniu pojęcia typu Trigger z pojęciem typu Screen reprezentującym ostatnio otwarty ekran aplikacji. Ekran ten jest określony zdaniem System-to-Screen, którego orzeczenie mapuje się na akcję typu „open" i które jest ostatnim występującym zdaniem tego typu przed zdaniem Actor-to-Trigger. W przypadku takiej konfiguracji scenariusza, edytor języka RSL może automatycznie utworzyć relację pomiędzy wspomnianymi pojęciami, co zwalnia użytkownika z konieczności ręcznego jej tworzenia. Semantyka takiej relacji została opisana powyżej w podrozdziale 3.2.2. Jest ona istotna z punktu widzenia translacji modelu RSL na kod aplikacji.

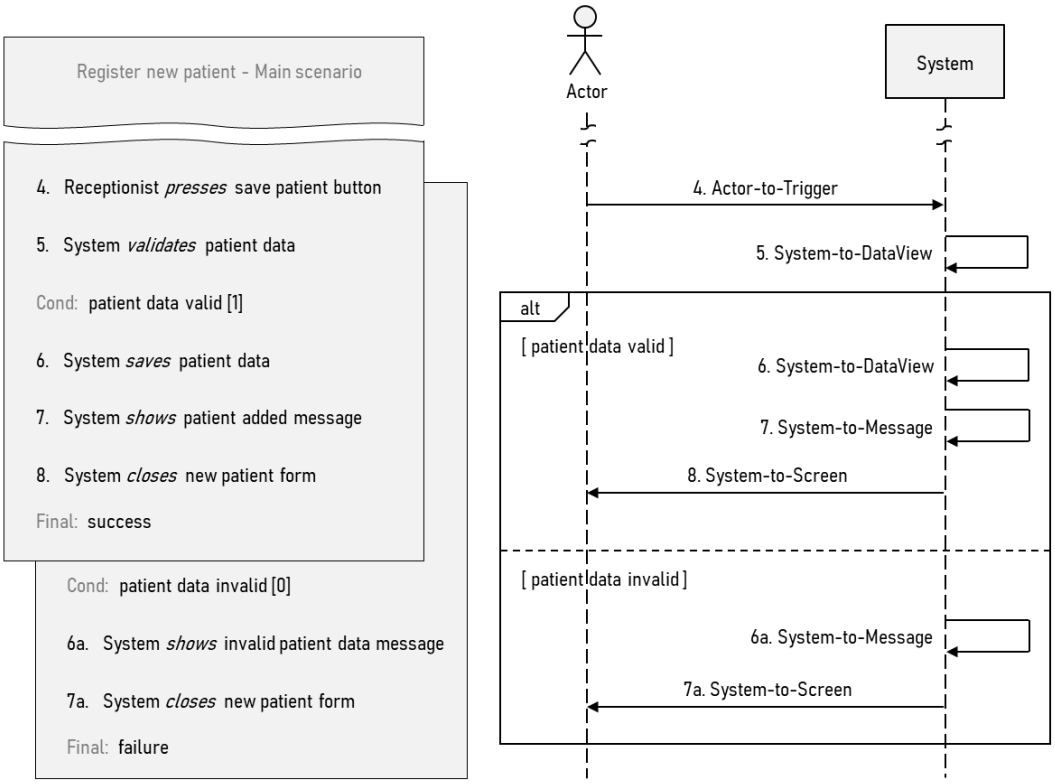
Zmiana stanu dialogu z „actor" na „system" może być następstwem zdania System-to-*, przy czym istotny w tym przypadku jest kontekst tego zdania. W przykładzie na Rysunek 20, taka zmiana stanu występuje w przypadku zdania „System *displays* new patient form". Jest to zdanie System-to-Screen, w którym orzeczenie mapuje się na operację „Show", której oczekiwanym efektem jest zaprezentowanie użytkownikowi ekranu do odczytu i/lub edycji danych. W przypadku zdania System-to-Screen, w którym orzeczenie mapuje się na operację „Close", np. „System *closes* new patient form", zmiana stanu dialogu zależy od kontekstu. W tym konkretnym przykładzie jest to ostatni krok scenariusza, po którym sterowanie wraca do miejsca wywołania przypadku użycia a więc następuje zmiana stanu dialogu. Jeżeli po tym

zdaniu następowałoby zdanie System-to-* (po zamknięciu ekranu system wykonuje jeszcze jakieś operacje, np. pobiera dane i otwiera nowe okno), stan dialogu nie uległby zmianie.

Zdanie System-to-Message (np. System *shows* patient added message), chociaż zbliżone do zdania System-to-Screen, nie zmienia stanu dialogu. Jak wyjaśniono w rozdziale 3.2.2 komunikaty (zarówno Message jak i Confirmation) zawsze powiązane są z co najmniej jednym wyzwalaczem, który musi być użyty przez aktora aby przejść do kolejnego kroku scenariusza. Dla uproszczenia scenariusza, nie jest wymagane umieszczanie w scenariuszu dodatkowego zdania Actor-to-Trigger po zdaniu System-to-Message.

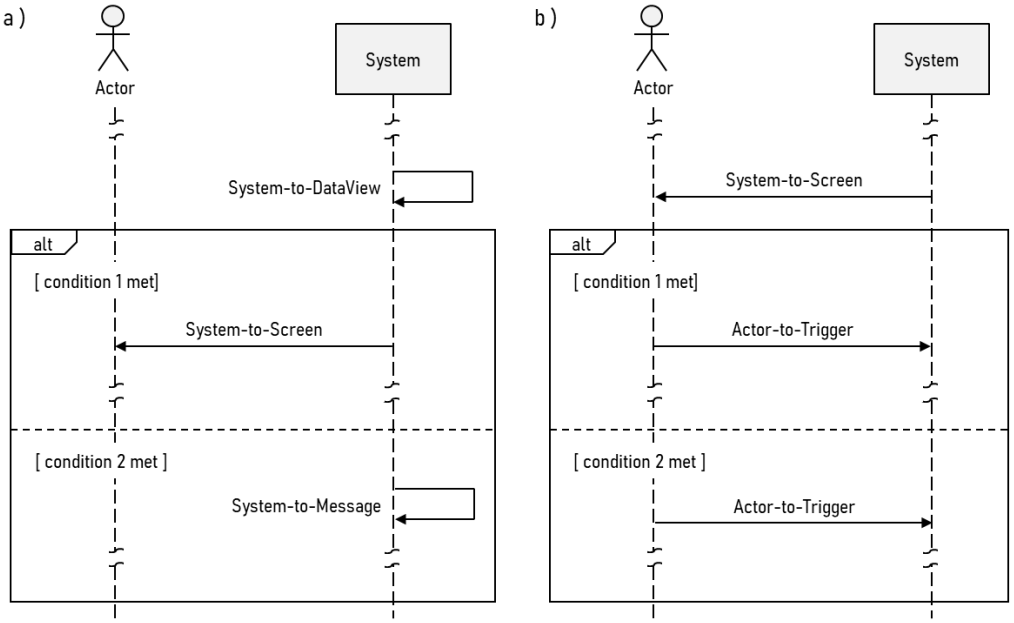
Scenariusz na Rysunek 20 zawiera też zdania typu Actor-to-DataView oraz System-to-DataView. Wyrażają one operacje na danych, które w danym kontekście nie zmieniają stanu dialogu. Więcej przykładów ilustrujących zmianę stanu dialogu znajduje się w dalszej części tego podrozdziału.

Zdania SVO mogą zmieniać stan dialogu ale nie mogą określać przepływu kontroli. Podstawowym typem zdania sterującego przepływem jest zdanie warunkowe (Condition). Zdania warunkowe mogą występować w grupach złożonych z dwóch lub więcej zdań. Warunki określone przez zdania w ramach takiej grupy muszą być rozłączne. Rysunek 21 przedstawia scenariusz główny i alternatywny dla rozbudowanego przypadku użycia „Register new patient” z poprzedniego przykładu. Tym razem dane pacjenta wprowadzane przez użytkownika są walidowane przez system a dalszy przepływ kontroli uwarunkowany jest wynikiem tej walidacji. W tym celu użyto dwóch zdań warunkowych, które stanowią punkt rozgałęzienia scenariusza głównego.



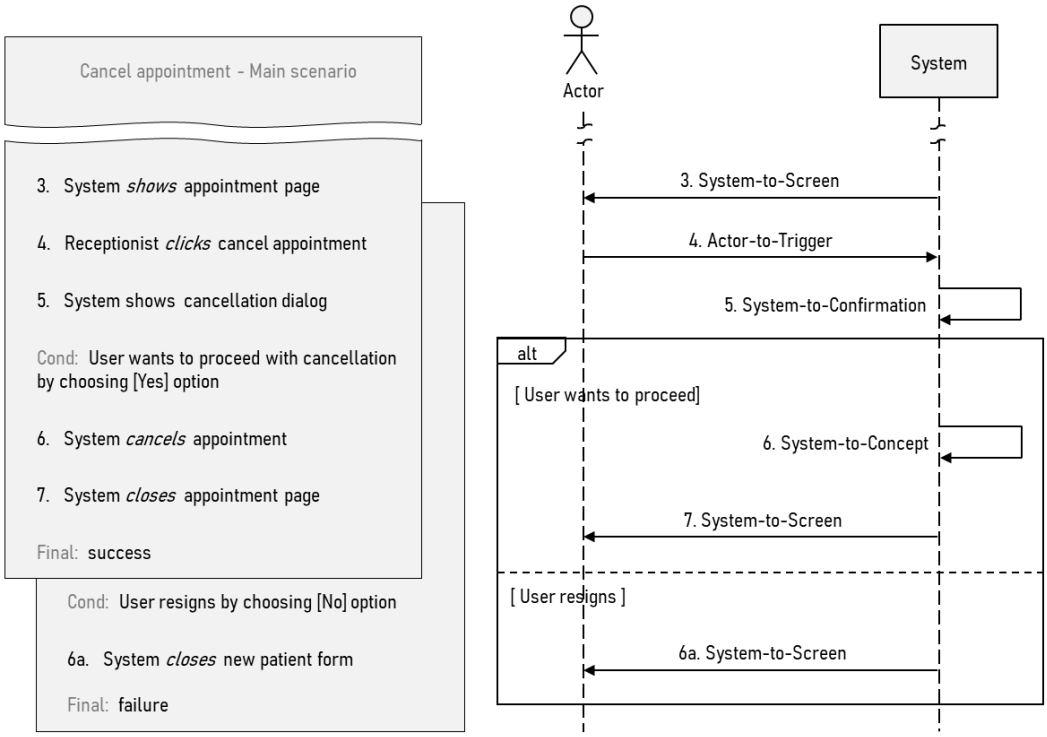
Rysunek 21 – Przepływ kontroli w scenariuszu – zdania warunkowe

W powyższym przykładzie, warunek dotyczy stanu systemu – wyniku operacji wykonanej na danych. Dzieje się tak zawsze, gdy zdanie warunkowe umieszczone jest w miejscu, gdzie stan dialogu wskazuje na „system". Jeżeli stanem dialogu jest „aktor", warunek powinien dotyczyć akcji wykonanej przez użytkownika, np. wyboru jednego z kilku dostępnych na danym ekranie przycisków, wyzwalających różne zachowania systemu. Zdanie warunkowe nie zmienia stanu dialogu. Porównanie obu sytuacji przedstawia Rysunek 22.



Rysunek 22 – Stan dialogu a zdania warunkowe

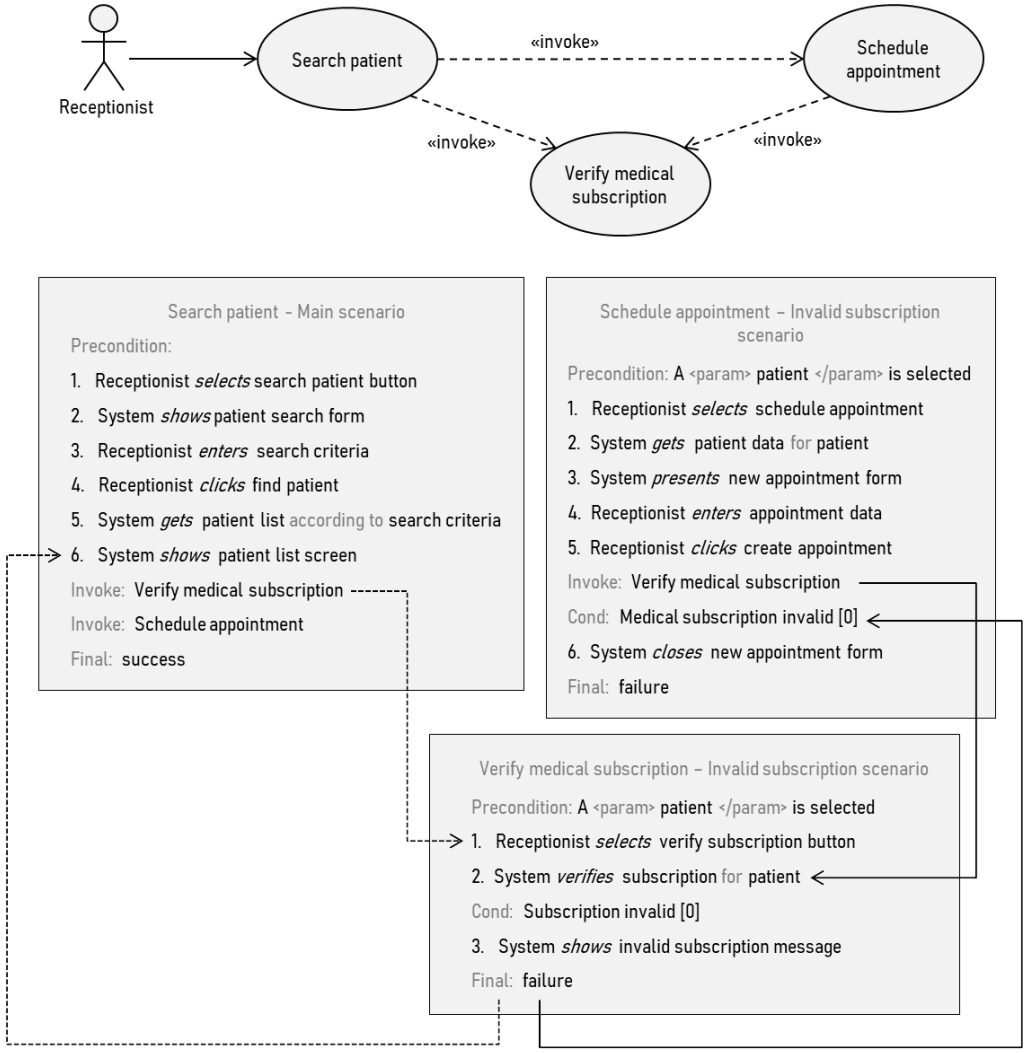
Zdanie warunkowe ma specjalne znaczenie w sytuacji, gdy występuje w kontekście zdania System-to-Confirmation. Jak wspomniano wcześniej, zdanie to nie wymaga tworzenia następujących po nim zdań Actor-to-Trigger oznaczających wybór opcji prezentowanych razem z komunikatem. Wystarczą same zdania warunkowe umieszczone bezpośrednio po zdaniu System-to-Confirmation. Każdy warunek musi odnosić się do jednego z dostępnych dla aktora wyzwalaczy. Wyzwalacze określane są w nawiasach kwadratowych w tekście zdania warunkowego. Od wybranego wyzwalacza zależy dalszy przepływ sterowania w ramach przypadku użycia. Przykład takiej konstrukcji przedstawia Rysunek 23.



Rysunek 23 – Zdania warunkowe w kontekście zdania System-to-Confirmation

Podobnie jak w przypadku zdań warunkowych, stan dialogu ma wpływ również na semantykę zdań typu Invoke. Jak omówiono w podrozdziale 3.3.1, przypadek użycia może być wywołany przez aktora lub niezależnie od niego. Pierwsza sytuacja ma miejsce zawsze, gdy zdanie Invoke znajduje się w punkcie, w którym stan dialogu wskazuje na aktora a druga – gdy stan dialogu wskazuje na system. Przykłady obu rodzajów wywołań prezentuje Rysunek 24.

W scenariuszu dla przypadku „Search patient", po wyświetleniu listy pacjentów (System *shows* patient list screen), użytkownik ma możliwość wykonania dla wybranego z listy pacjenta dwóch funkcji: umówienie wizyty lekarskiej („Schedule appointment") oraz sprawdzenie ważności abonamentu medycznego („Verify medical subscription"). Zdania Invoke odnoszące się do tych przypadków użycia umieszczone są po zdaniu System-to-Screen otwierającym nowy ekran – w miejscu, gdzie stan dialogu wskazuje na aktora. Oznacza to, że ekran „Patient list screen" musi zawierać wyzwalacze, które umożliwią aktorowi wywołanie wskazanych przypadków użycia, chociaż nie musi on z nich skorzystać. Zdania Invoke nie muszą być umieszczone bezpośrednio po zdaniu System-to-Screen. Jeżeli stan dialogu dla zdania Invoke wskazuje na aktora, to zawsze oznacza ono dostępność odpowiedniego wyzwalacza na ekranie aktualnie widocznym dla użytkownika, nawet jeżeli pomiędzy zdaniem System-to-Screen oraz Invoke występuje sekwencja innych zdań.



Rysunek 24 – Przepływ kontroli pomiędzy przypadkami użycia – zdania Invoke

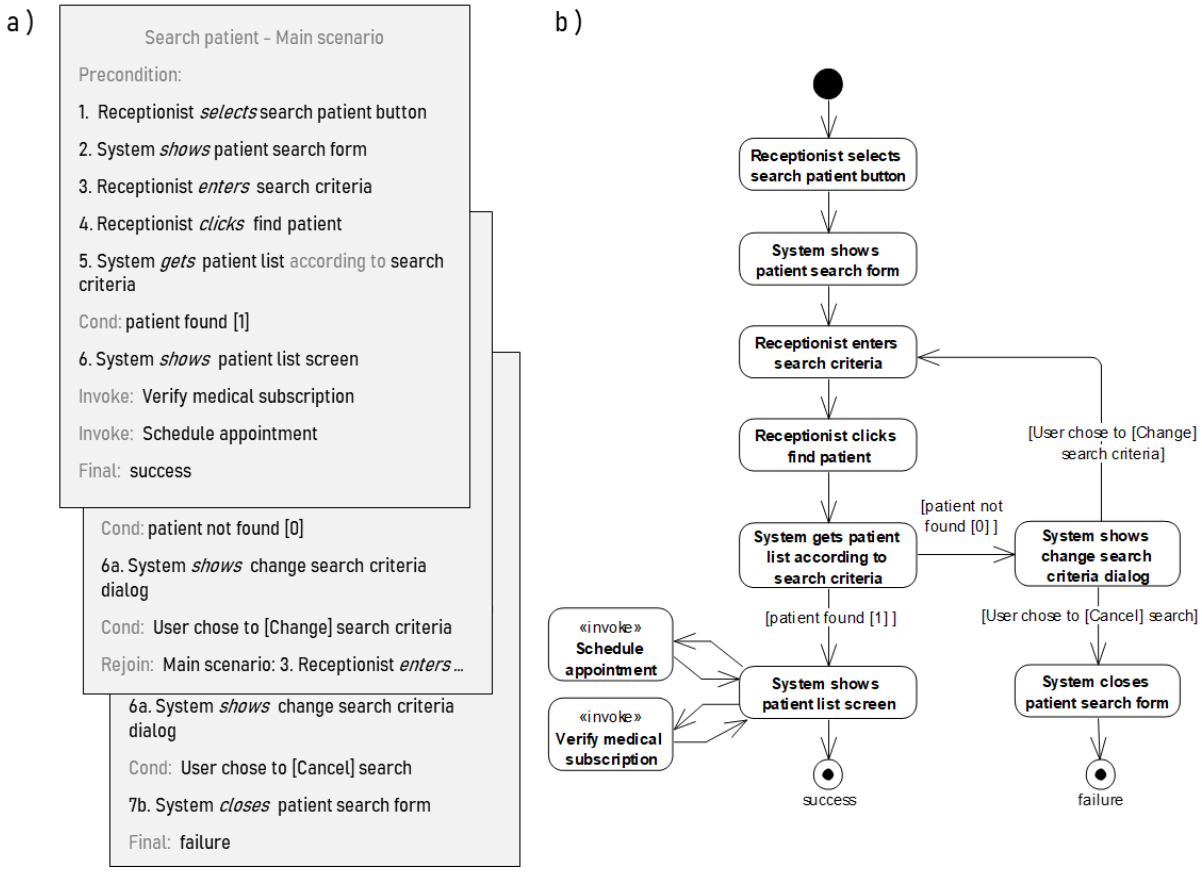
Decyzja aktora o wywołaniu zewnętrznego przypadku użycia oznacza przekazanie sterownia do pierwszego zdania scenariusza przypadku wywoływanego. W takiej sytuacji, zdanie to musi być typu Actor-to-Trigger i odnosić się do operacji „Select". Wykonanie tego zdania oznacza faktyczne uruchomienie przez aktora przypadku wywoływanego. W przykładzie na Rysunek 24 jest to zdanie „Receptionist *selects* verify subscription button" a przepływ kontroli oznaczony jest przerywaną strzałką. Należy zauważyć, że możliwość wywołania przypadku użycia przez aktora może być uzależniona od spełnienia warunku wstępnego określonego w zdaniu „Precondition" w scenariuszu przypadku wywoływanego. W prezentowanym przykładzie warunek określa, że w momencie wywoływania przypadku „Verify medical subscription" musi być wybrany konkretny pacjent, dla którego ma być wykonana weryfikacja abonamentu medycznego.

Po zakończeniu wykonywania wywołanego przypadku użycia, przepływ sterowania wraca do miejsca wywołania a stan dialogu ponownie wskazuje na aktora, niezależnie jak stan dialogu zmieniał się podczas wykonywania przypadku wywołanego. W tym momencie aktor może podjąć decyzję o ponownym wywołaniu tego samego lub innego dostępnego przypadku użycia.

Rysunek 24 prezentuje również wywołanie przypadku użycia przez system, niezależnie od decyzji aktora. Umówienie wizyty dla pacjenta (Schedule appointment) jest możliwe tylko wtedy, gdy pacjent posiada ważny abonament medyczny. W przedstawionym scenariuszu, gdy użytkownik wybiera opcję utworzenia wizyty, system automatycznie weryfikuje abonament pacjenta poprzez wywołanie przypadku „Verify medical subscription". Ponieważ w momencie wywołania stan dialogu wskazuje na system, sterowanie przekazywane jest do przypadku wywoływanego z pominięciem pierwszego zdania Actor-to-Trigger (patrz: strzałka ciągła na Rysunek 24).

Rezultat wykonania wywoływanego przypadku użycia może mieć znaczenie dla dalszego przebiegu przypadku wywołującego. W takiej sytuacji, po zdaniu Invoke mogą zostać użyte dwa zdania warunkowe tworzące alternatywne ścieżki – jedna, gdy przypadek wywoływany zakończy się powodzeniem, druga dla rezultatu negatywnego.

Warto zauważyć, że konstrukcję, w której zdanie Invoke jest wywoływane przez aktora, można zastąpić zdaniem Invoke wywoływanym przez system. W przypadku tej drugiej konstrukcji, każde zdanie Invoke musi być poprzedzone sekwencją złożoną kolejno ze zdania warunkowego i zdania Actor-to-Trigger. Oznacza to konieczność utworzenia scenariusza alternatywnego dla każdego zdania Invoke. Konstrukcja pierwsza pozwala tego uniknąć, co upraszcza specyfikowanie przypadków użycia, oraz ich czytelność a także ułatwia automatyczne przetwarzanie modelu wymagań.



Rysunek 25 – Alternatywne notacje dla scenariuszy: (a) notacja tekstowa, (b) diagram aktywności

Rysunek 25 przedstawia przykład kompletnej specyfikacji przypadku użycia „Search patient". Mamy tutaj scenariusz główny oraz dwa scenariusze alternatywne. Jedna ze ścieżek alternatywnych kończy się porażką („failure") a druga zdaniem typu Rejoin, które kieruje przepływ kontroli z powrotem do scenariusza głównego, tworząc pętlę. W przypadku zdania Rejoin, należy zwrócić uwagę aby, oprócz scenariusza zawierającego takie zdanie, istniał scenariusz alternatywny. Innymi słowy – pomiędzy zdaniem, na które wskazuje zdanie Rejoin a samym zdaniem Rejoin powinno występować zdanie warunkowe, które umożliwia wyjście z pętli. Inaczej, mielibyśmy scenariusz będący pętlą nieskończoną, bez zdania końcowego. Składnia języka RSL dopuszcza takie konstrukcje, jednakże edytor scenariuszy powinien je wykrywać i informować o nich użytkownika.

Scenariusze na Rysunek 25 zaprezentowane są z użyciem dwóch notacji jakie definiuje język RSL: notacji tekstowej (Rysunek 25a) oraz w postaci diagramu aktywności (Rysunek 25b) znanego z języka UML. Notacje te są w pełni ekwiwalentne i mogą być używane zamiennie. Notacja tekstowa jest nieco bardziej użyteczna podczas pisania scenariuszy, natomiast diagram

aktywności lepiej ukazuje przepływ kontroli pomiędzy scenariuszami przypadku użycia oraz pomiędzy przypadkami użycia w przypadku relacji Invoke.

Opisane w tym rozdziale konstrukcje i reguły wyrażania logiki aplikacji za pomocą scenariuszy przypadków użycia z pewnością nie wyczerpują wszystkich możliwych sytuacji jakie mogą mieć miejsce w złożonych aplikacjach. Wynika to z konieczności ograniczenia zakresu niniejszej pracy. Celem jest zdefiniowanie semantyki translacyjnej oraz jej implementacji uwzględniającej podstawową i najczęściej spotykaną logikę działania typowych aplikacji. Dzięki swojej elastyczności, język RSL może być rozbudowany o dodatkowe typy pojęć, typy zdań, reguły tworzenia scenariuszy, itd., które umożliwiłyby modelowanie bardzo złożonej logiki aplikacji. Oczywiście, wymagałoby to również zdefiniowania semantyki translacyjnej oraz implementującej jej transformacji, aby móc generować kod dla nowych elementów i reguł.

4 Metamodel języka RSL

Rozdział ten opisuje „techniczny” metamodel języka RSL, tzn. w postaci, w jakiej został on zaimplementowany w środowisku ReDSeeDS i MOLA. Dodatkowo, zostały tu opisane wybrane elementy metamodelu języka SCL, które stanowią nadbudowę na język RSL oraz UML zapewniającą spójność i integralność całego Software Case’a.

Rozdział ten ogranicza się do elementów języka SCL i RSL istotnych z punktu widzenia opisanej w tej pracy semantyki translacyjnej oraz jej implementacji. Kompletną definicję omawianych języków zawierają ich formalne specyfikacje [119] [39] [36]. Metamodel języka UML został w tej pracy pominięty – jest on powszechnie dostępny w oficjalnej specyfikacji języka UML [79].

4.1 Infrastruktura dla definiowania języków modelowania

Dwie najczęściej spotykane metody zapisu składni abstrakcyjnej języków formalnych to gramatyki bezkontekstowe i metamodele. Przez gramatykę bezkontekstową [144] rozumiemy gramatykę opisującą język, w którym znaczenie konkretnych elementów nie zależy od kontekstu, w którym one występują. Gramatyki definiuje się z pomocą reguł produkcji, które dokonują przekształceń na symbolach reprezentujących możliwe elementy języka. Typowym sposobem przedstawiania gramatyk bezkontekstowych jest notacja BNF (ang. Backus-Naur Form) [145] lub jej rozszerzona wersja EBNF (ang. Extended Backus-Naur Form) [146]. Notacje te są powszechnie używane do zapisu składni abstrakcyjnej języków programowania oraz protokołów komunikacyjnych. W przypadku języków modelowania, ich składnię abstrakcyjną definiuje się najczęściej w postaci metamodelu. Przyjmując, że model jest abstrakcją pewnego obiektu ze świata rzeczywistego, to metamodel jest abstrakcją ukazującą właściwości owego modelu. Innymi słowy, metamodel jest modelem języka modelowania [65]. Ponieważ metamodel jest modelem, sam może być reprezentowany za pomocą języka modelowania. Jednym z popularniejszych języków do wyrażania metamodeli jest Meta Object Facility (MOF) [121]. Jest to standard utrzymywany przez Object Management Group[21].

MOF przypomina uproszczony model klas zapisany w języku UML i jest minimalnym zbiorem elementów, które mogą być użyte w celu zdefiniowania innego języka modelowania. W rzeczywistości MOF składa się z dwóch podobnych języków: Essential MOF (EMOF) oraz

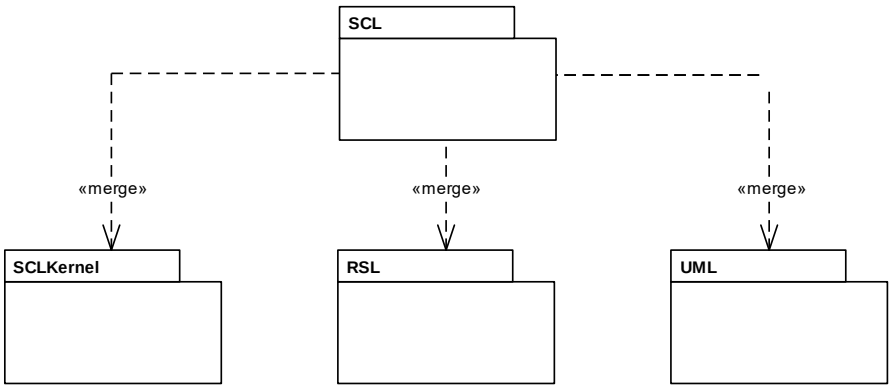
[21] www.omg.org/mof/

Complete MOF (CMOF). CMOF oferuje cechy ułatwiające definiowanie bardzo złożonych metamodeli, które są czytelne dla odbiorców i łatwe w utrzymaniu lecz trudne w implementacji. EMOF, z kolei, jest prostszym językiem, który kładzie nacisk na łatwość implementacji zdefiniowanych w nim metamodeli kosztem ekspresywności.

Oficjalne specyfikacje języków RSL [36] i UML [79] używają języka CMOF. Na potrzeby implementacji narzędzia ReDSeeDS przygotowane zostały uproszczone wersje metamodeli tych języków z użyciem EMOF. EMOF jest również wymagany przez środowisko transformacji MOLA.

4.2 Pakiet SCL Kernel

Rysunek 26 przedstawia ogólną strukturę języka SCL, który scala w jedną spójną całość elementy języka RSL oraz UML. Pakiet `SCLKernel` zawiera metaklasy definiujące podstawowe elementy, z których zbudowany jest każdy Software Case w języku SCL. Definiuje on również powiązania pomiędzy elementami języka RSL oraz UML za pomocą odpowiednich relacji.



Rysunek 26 – Struktura metamodelu języka SCL

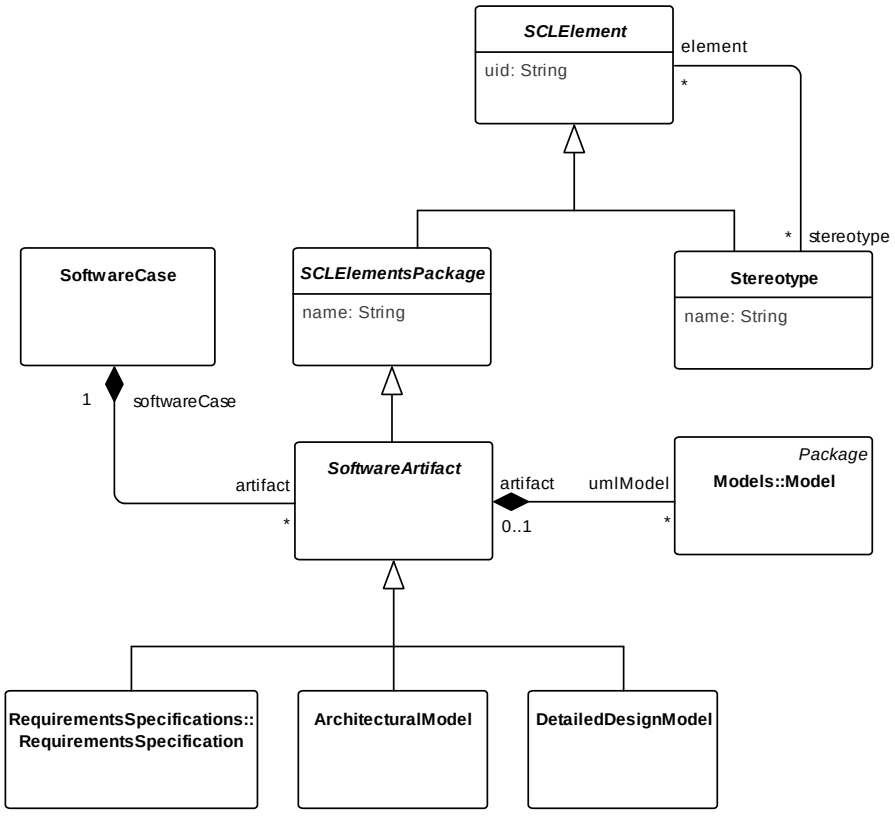
Rysunek 27 przedstawia strukturę Software Case'a. Podstawowym elementem jest tutaj `SCLElement` – abstrakcyjna metaklasa, po której dziedziczą pozostałe elementy tworzące Software Case. `SCLElement` definiuje właściwość `uid` – unikalny identyfikator elementu. Ponadto, `SCLElement`, a zatem wszystkie metaklasy dziedziczące po nim, może posiadać stereotyp (`Stereotype`) modyfikujący semantykę tego elementu.

Software Case jest reprezentowany przez metaklasę `SoftwareCase`, której elementami składowymi są artefakty (`SoftwareArtifact`). `SoftwareArtifact` to abstrakcyjny

element będący podtypem `SCLElement`, który dodatkowo posiada właściwości pakietu (`SCLElementsPackage`), m.in. nazwę (atrybut `name`). Elementem składowym każdego artefaktu są modele stanowiące zbiory elementów języka SCL modelujące określone aspekty systemu oprogramowania. Modele reprezentowane są za pomocą metaklasy `Model` z pakietu `Models` języka UML.

W obecnej wersji języka SCL istnieją trzy metaklasy reprezentujące konkretne rodzaje artefaktów:

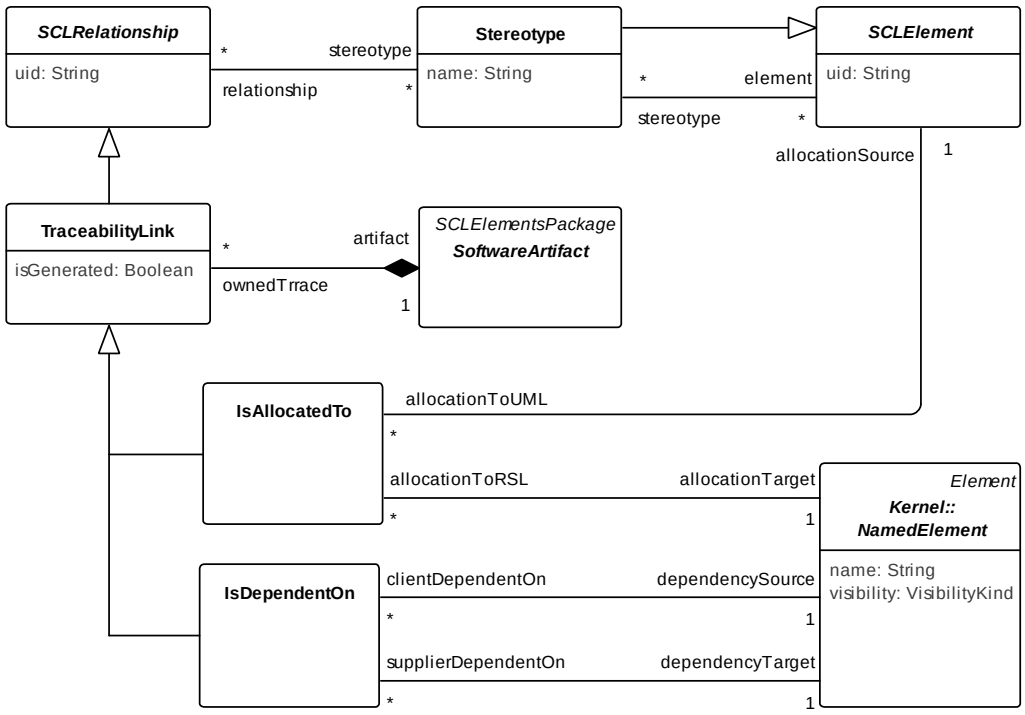
- `RequirementsSpecification` – reprezentuje specyfikację wymagań w języku RSL,
- `ArchitecturalModel` – reprezentuje model architektury w języku UML,
- `DetaledDesignModel` – reprezentuje model projektu systemu w języku UML.



Rysunek 27 – Metamodel języka SCL – elementy podstawowe

Oprócz elementów wchodzących w skład artefaktów Software Case, język SCL definiuje dodatkowo zależności między tymi elementami. Zależności te nie są używane do modelowania

systemu jako takiego lecz stanowią narzędzie do zapewniania identyfikowalności (ang. traceability) elementów modeli generowanych automatycznie w toku transformacji. Przykładowo, pozwalają identyfikować zależność elementów modelu projektowego wyrażonych w języku UML (np. operacji klas) od konkretnych elementów modelu wymagań (np. zdań scenariusza), na podstawie których te pierwsze zostały wygenerowane. Spełniają, zatem, ważną rolę w odzwierciedlaniu struktury modeli źródłowych przez struktury modeli docelowych [147].



Rysunek 28 – Metamodel języka SCL – powiązania między elementami języka

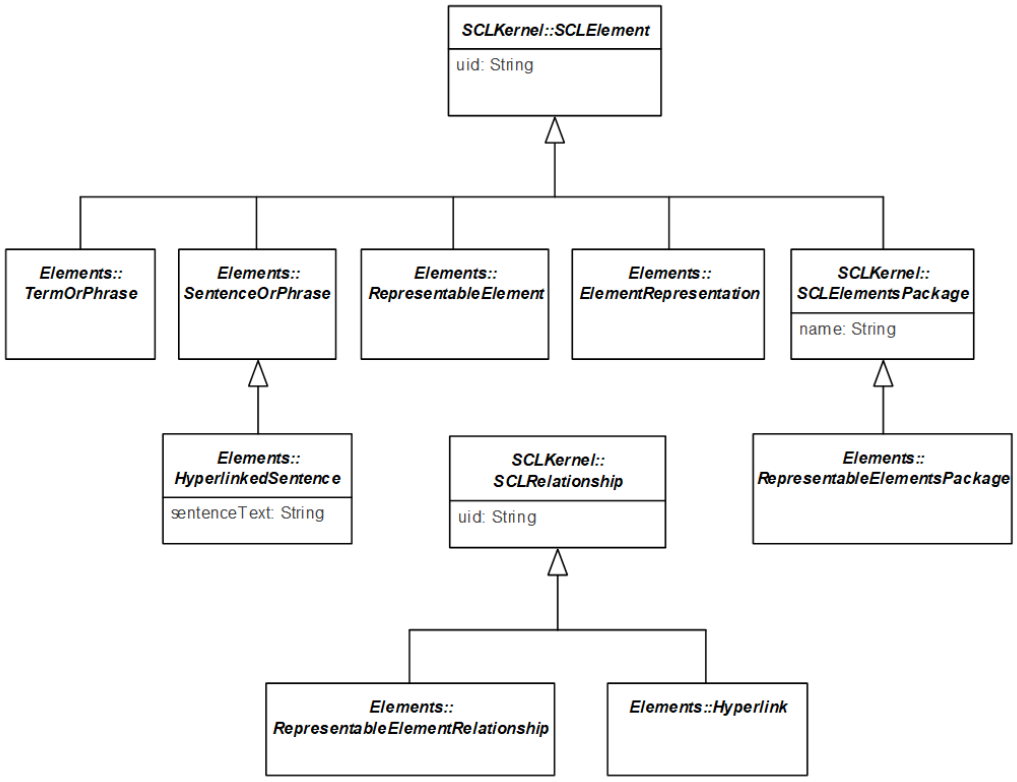
Rysunek 28 przedstawia fragment metamodelu definiujący relacje identyfikowalności pomiędzy elementami języka SCL. `TraceabilityLink` jest wspólną superklasą dla specjalizowanych typów relacji: `IsAllocatedTo` oraz `IsDependentOn`. Atrybut `isGenerated` w metaklasie `TraceabilityLink` wskazuje czy relacja została wygenerowana automatycznie w wyniku transformacji. `TraceabilityLink` jest specjalizacją `SCLRelationship`, dzięki czemu wszystkie relacje identyfikowalności mogą posiadać stereotypy modyfikujące ich semantykę, co zostało szeroko wykorzystane na potrzeby transformacji „RSL to Java” opisanej w tej pracy.

`IsAllocatedTo` jest relacją łączącą elementy języka RSL z elementami języka UML. Źródłem relacji (`allocationSource`) może być każdy element dziedziczący po `SCLElement`, czyli wszystkie podstawowe element języka RSL. Celem relacji (`allocationTarget`) może być, natomiast, każdy element będący specjalizacją metaklasy `NamedElement` z pakietu `Kernel` języka UML.

Relacje zależności pomiędzy elementami UML są reprezentowane przez metaklasę `IsDependentOn`. W odzróżnieniu od `IsAllocatedTo`, służy ona do wyrażania zależności jedynie pomiędzy elementami języka UML – źródłem i celem relacji są instancje metaklasy `NamedElement`. Relacja ta może być wykorzystywana jako relacja pomocnicza w czasie wykonywania transformacji.

4.3 Pakiet RSL Kernel

Podstawą definicji języka RSL jest pakiet `RSLKernel`. Zawiera on reprezentacje najbardziej podstawowych pojęć: elementów, ich reprezentacji, zależności oraz pakietów. Pojęcia te reprezentowane są przez abstrakcyjne metaklasy, które stanowią podstawę definicji pozostałych elementów języka RSL opisanych w dalszej części rozdziału.



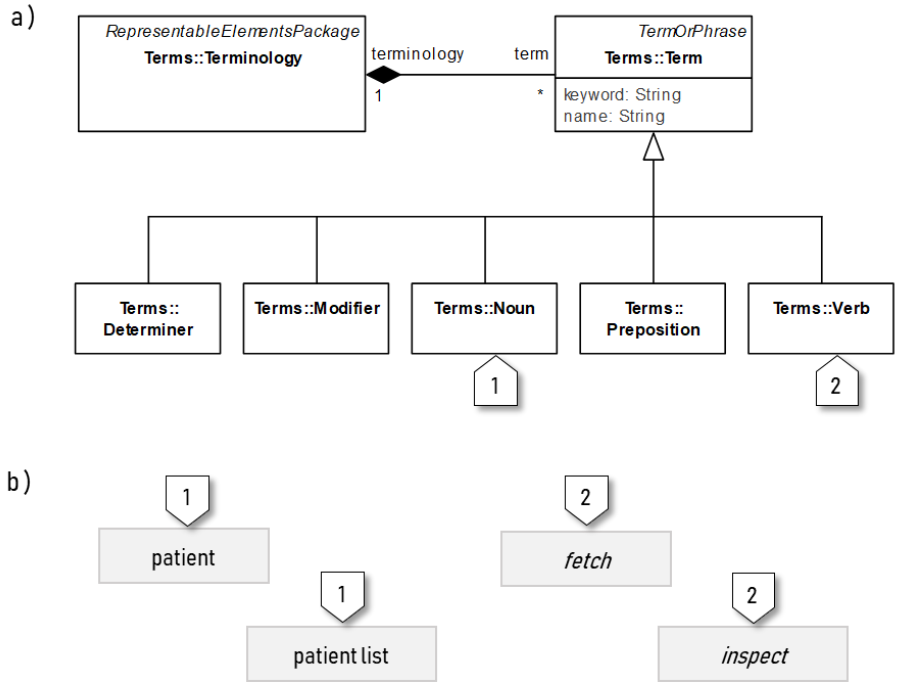
Rysunek 29 – Metamodel języka RSL – podstawowe elementy zdefiniowane w pakiecie Kernel

Diagram na Rysunek 29 prezentuje metaklasy w pakiecie `RSLKernel`. Wszystkie one dziedziczą z bazowych elementów zdefiniowanych w pakiecie `SCLKernel`, co zapewnia integralność w ramach całego Software Case'a.

Warto odnotować, że istnieją osobne metaklasy reprezentujące element języka RSL (`RepresentableElement`) oraz jego reprezentację (`ElementRepresentation`), co przejawia się m.in. odseparowaniem przypadków użycia od ich reprezentacji w postaci scenariuszy. Warto również zaznaczyć, że powiązania między elementami RSL mogą być realizowane albo za pomocą `RepresentableElementRelationship`, albo `Hyperlink`, co pomaga w zapewnieniu spójności całego modelu wymagań.

4.4 Terminy i frazy

Złożone elementy języka RSL, takie jak scenariusze i zdania, zbudowane są z szeregu prostszych elementów składowych. Najprostsze z nich, to terminy (ang. terms). Zbiór wszystkich terminów tworzy terminologię (ang. terminology), która jest wspólna dla wszystkich Software Case'ów. Metamodel definiujący tę cześć języka przedstawiony jest na Rysunek 30a.

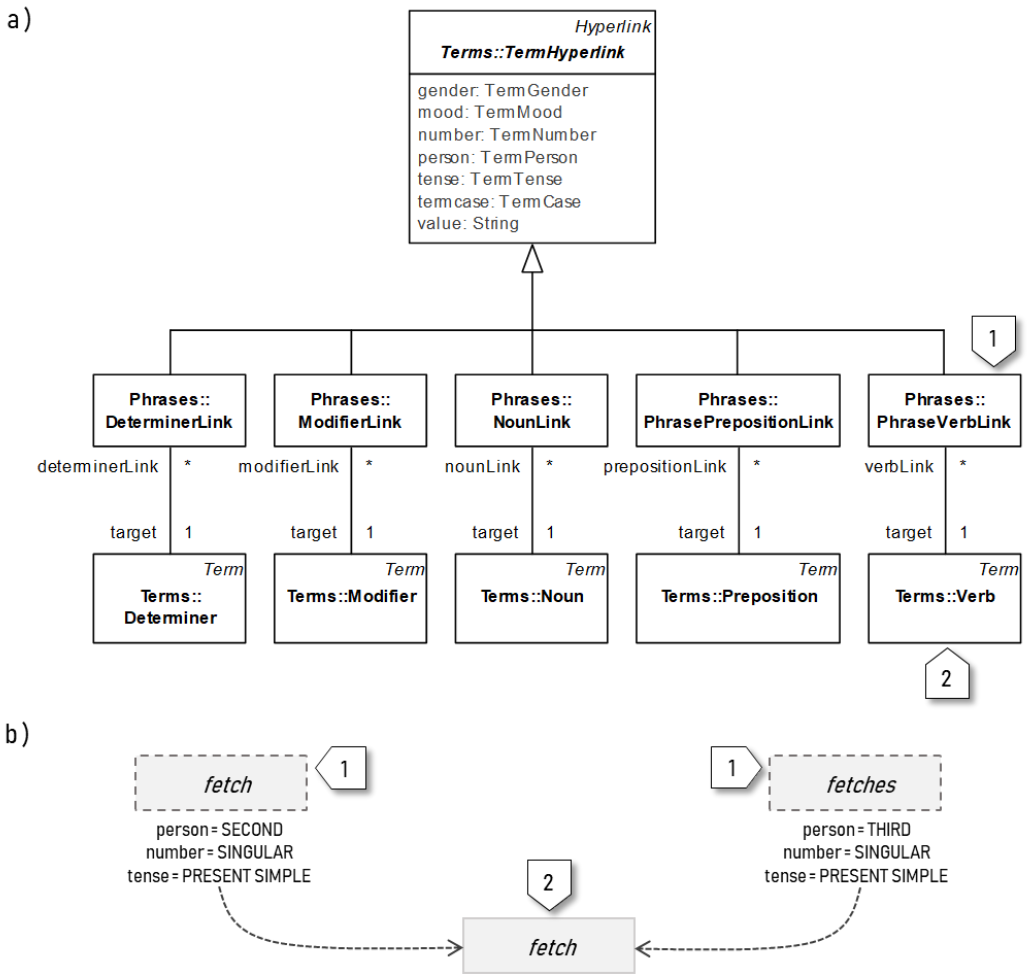


Rysunek 30 – Metamodel dla terminologii: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Terminologia reprezenowana jest przez metaklasę `Terminology` i może zawierać dowolną liczbę terminów (`Terms`). Atrybut `name` w metaklasie `Term` zawiera napis stanowiący podstawową formę wyrazową danego terminu. Atrybut `keyword` ma zastosowanie głównie dla czasowników i może wyrażać predefiniowaną operację systemu (np. READ, DELETE, VALIDATE) przypisaną danemu czasownikowi, jak opisano w rozdziale 3.2.3. Istnieje kilka podtypów metaklasy `Term`, reprezentujących różne część mowy. Dla najczęściej używanych części mowy – rzeczownika (`Noun`) i czasownika (`Verb`) – zostały przedstawione przykłady składni konkretnej na Rysunek 30b. Przykłady składni konkretnej oznaczone są etykietami z numerami odnoszącymi się do odpowiednich elementów metamodelu. Jak widać w przedstawionym przykładzie, nazwa terminu może być wielowyrazowa, np. „patient list".

Terminologia może być uzupełniana ręcznie, np. w trakcie pisania specyfikacji wymagań, lub poprzez zaimportowanie zawartości powszechnie dostępnych leksykalnych baz danych, np. WordNet [148]. Dzięki temu, specyfikacje wymagań w języku RSL mogą być tworzone w oparciu o zbiór wyrazów dostępnych w danym języku naturalnym. Takie podejście zapewnia m.in. spójność specyfikacji wymagań oraz możliwość porównywania różnych specyfikacji między sobą. Jest to ważna cecha języka RSL, która umożliwia określenie podobieństwa między przypadkami oprogramowania oraz ich ponowne wykorzystanie. Wykracza to jednak poza zakres tej pracy; więcej szczegółów na ten temat można znaleźć w [108] [39].

Terminy są singletonami – terminologia może zawierać tylko jedną instancję danego terminu. W związku z tym, konstrukcje wyższego rzędu, np. frazy (ang. phrases), odwołują się do terminologii za pomocą hiperłączy. Hiperłącza reprezentowane są przez metaklasę `TermHyperlink` (Rysunek 31a). Posiada ona szereg atrybutów, których wartości określają konkretne formy elementów terminologii, na które wskazuje dane hiperłącze, m.in. osoba, rodzaj, liczba, czas, itd. Atrybut `value` zawiera nazwę terminu wskazywanego przez hiperłącze w konkretnej formie określonej przez pozostałe atrybuty.



Rysunek 31 – Metamodel dla hyperlinków: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

`TermHyperlink` posiada wiele podtypów, z których każdy reprezentuje klasę hiperłączy dla danego typu terminu z terminologii, np. `NounLink` dla terminów typu `Noun`, `PhraseVerbLink` dla terminów typu `Verb`, itd. Przykład na Rysunek 31b przedstawia pojedynczy element terminologii wskazywany przez dwa rożne hiperłącza – jeden odpowiadający angielskiemu czasownikowi „fetch” w drugiej osobie liczby pojedynczej czasu Present Simple oraz drugi odpowiadający temu samemu czasownikowi w osobie trzeciej liczby pojedynczej czasu Present Simple.

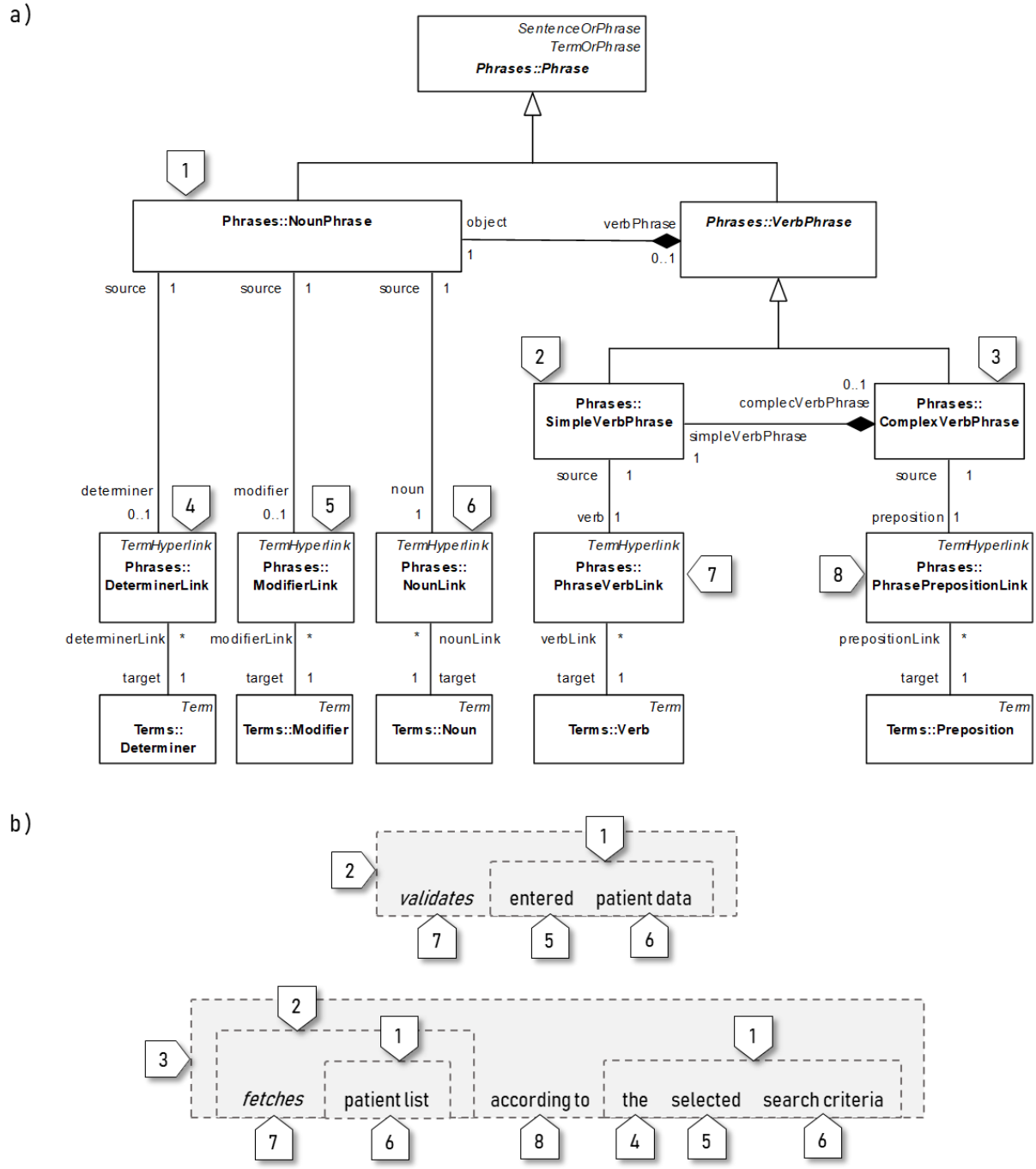
Jak wspomniano powyżej, hiperłącza wykorzystywane są do budowania fraz. Każda fraza w języku RSL składa się z zestawu hiperłączy wskazujących na terminy określonego typu. Rysunek 32a przedstawia składnię abstrakcyjną dla fraz. Wszystkie frazy reprezentowane są przez abstrakcyjna metaklasę `Phrase`, która ma dwa główne podtypy: frazę rzeczownikową (`NounPhrase`) oraz frazę czasownikową (`VerbPhrase`).

Fraza rzeczownikowa posiada dokładnie jedno hiperłącze typu `NounLink` wskazujące na rzeczownik (`Noun`) będący podstawą tej frazy. Fraza rzeczownikowa może dodatkowo posiadać maksymalnie jeden `DeterminerLink` oraz `ModifierLink`. Wskazują one na terminy stanowiące określniki rzeczownika: `Determiner` oraz `Modifier`. Przykłady fraz rzeczownikowych pokazane są na Rysunek 32b. Najprostszą z przykładowych fraz jest „patient list", składająca się jedynie z rzeczownika. Bardziej złożonym przykładem jest fraza „the selected search criteria", w której rzeczownik „search criteria" poprzedzony jest określnikami „the" (`Determiner`) oraz „selected" (`Modifier`).

Bardziej złożonym typem frazy jest fraza czasownikowa reprezentowana przez abstakcyjną metaklasę `VerbPhrase`. Ma ona dwa konkretne podtypy: prostą frazę czasownikową (`SimpleVerbPhrase`) oraz złożoną frazę czasownikową (`ComplexVerbPhrase`). Prosta fraza czasownikowa zawiera dokładnie jedną frazę rzeczownikową w roli dopełnienia (atrybut `object` dziedziczony z nadklasy) oraz jedno hiperłącze typu `PhraseVerbLink` wskazujące na czasownik z terminologii (`Verb`). Konstrukcja taka wyraża czynność wykonywaną na obiekcie. Rysunek 32b prezentuje dwa przykłady prostych fraz czasownikowych (oznaczone etykietami z numerem 2): „validates entered patient data" oraz „fetches patient list". Jak widać na przedstawionych przykładach, fraza rzeczownikowa tworząca frazę czasownikową, może składać się jedynie z obowiązkowego rzeczownika lub dodatkowo zawierać opcjonalne określniki.

Prosta fraza czasownikowa może istnieć samodzielnie lub stanowić element składowy złożonej frazy czasownikowej (`ComplexVerbPhrase`), która opisuje zależność behawioralną pomiędzy dwoma obiektami – dopełnieniem bliższym oraz dopełnieniem dalszym. Rolę dopełnienia dalszego pełni fraza rzeczownikowa odziedziczona po superklasie `VerbPhrase`, natomiast dopełnienie bliższe, wraz z czasownikiem wyrażającym akcję, zawarte jest w prostej frazie czasownikowej będącej elementem składowym frazy złożonej (patrz relacja kompozycji pomiędzy `ComplexVerbPhrase` i `SimpleVerbPhrase`). Uzupełnieniem złożonej frazy czasownikowej jest hiperłącze `PrepositionLink` wskazujące na przyimek `Preposition`, który łączy frazę zawierającą dopełnienie bliższe z frazą zawierającą dopełnienie dalsze. Przykład złożonej frazy czasownikowej przedstawiony jest na Rysunek 32b i oznaczony numerem 3. Przykładowa fraza zawiera prostą frazę czasownikową „fetches patient list", gdzie „patient list" jest dopełnieniem bliższym, oraz frazę rzeczownikową „the selected search

criteria” stanowiącą dopełnienie dalsze. Relację między oboma dopełnieniami wyraża przyimek „according to”.

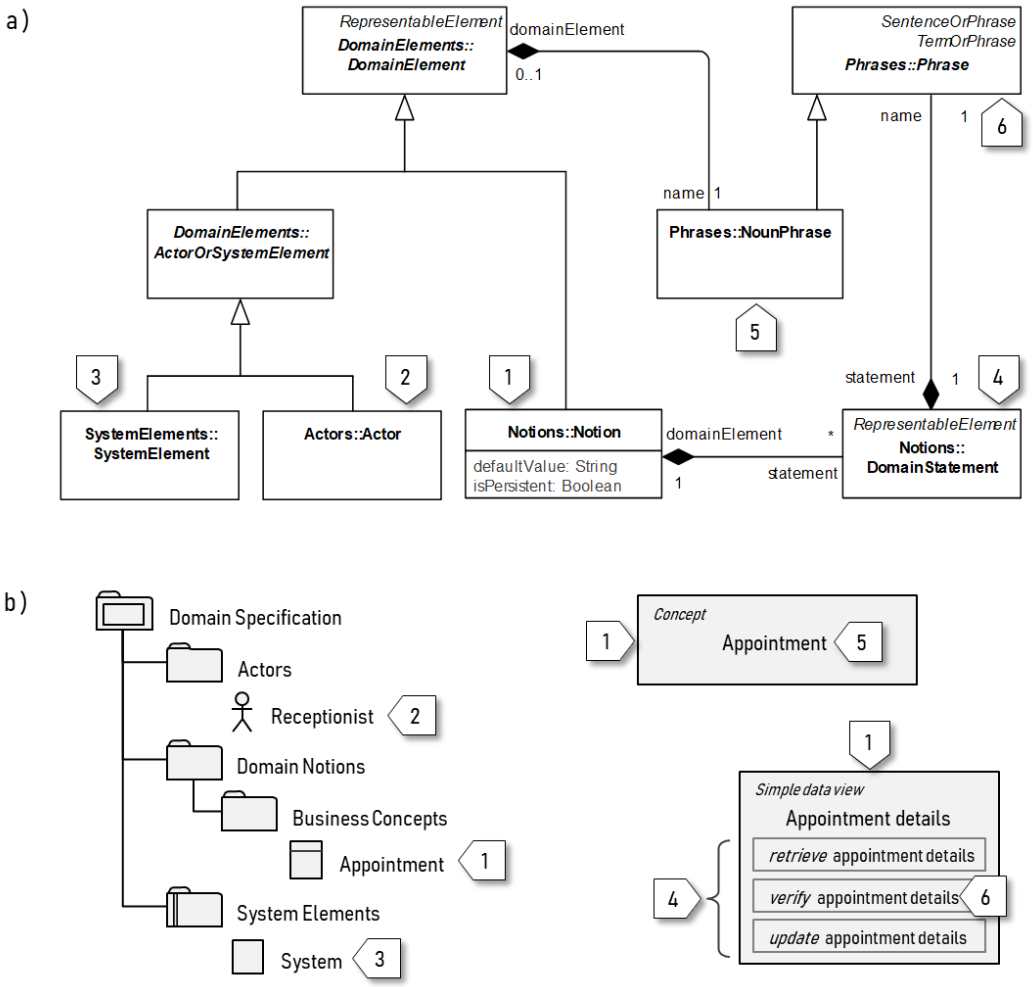


Rysunek 32 – Metamodel dla fraz: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

4.5 Elementy dziedzinowe i ich powiązania

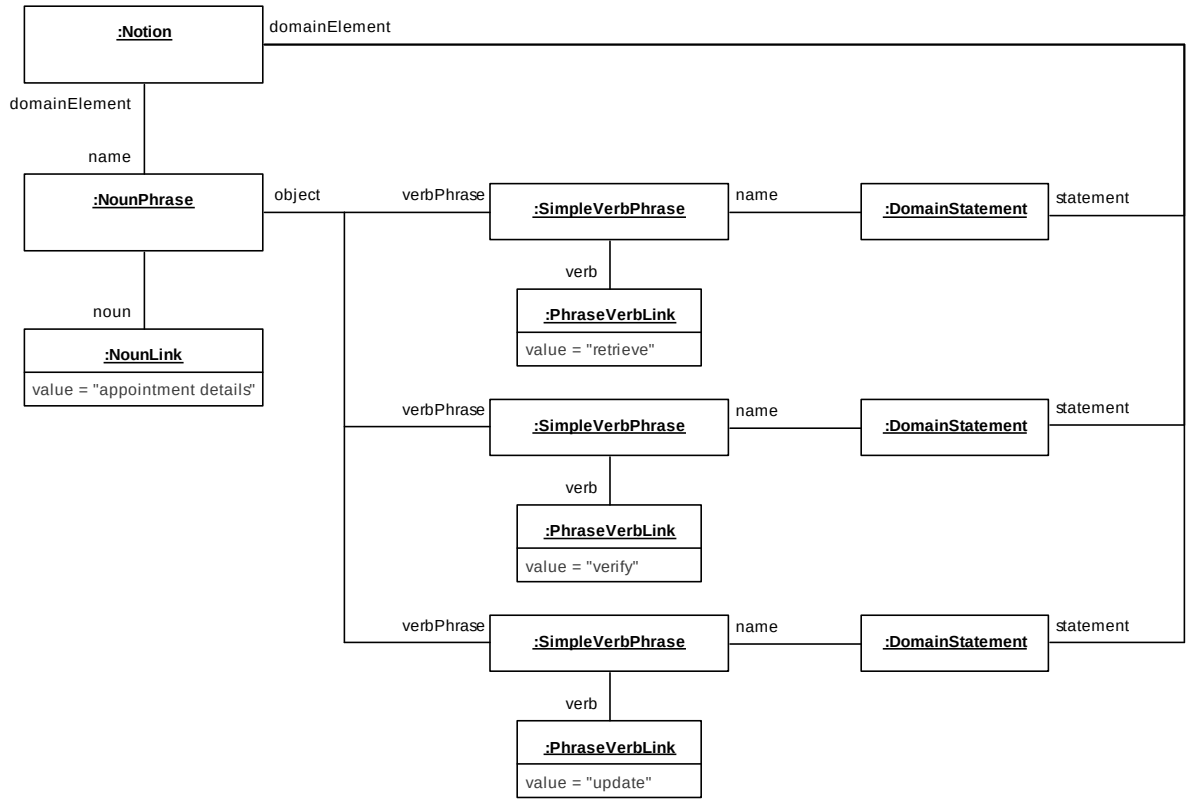
Jak zaznaczono wcześniej, terminy i frazy służą do budowania bardziej złożonych struktur, m.in. elementów dziedzinowych. Język RSL definiuje trzy typy takich elementów: pojęcie

(Notion), aktor (Actor) oraz system (SystemElement). Wszystkie wymienione metaklasy są podtypami abstrakcyjnej metaklasy DomainElement, co prezentuje fragment metamodelu na Rysunek 33a.



Rysunek 33 – Metamodel dla elementów dziedzinowych: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Cechą każdego elementu dziedzinowego odziedziczoną po nadklasie DomainElement jest nazwa tego elementu (atrybut name). Nazwa ma postać frazy rzeczownikowej (NounPhrase), która – jak pokazano w poprzednim podrozdziale – za pomocą odpowiedniego hiperłącza wskazuje na rzeczownik z terminologii. Rysunek 33b przedstawia przykłady składni konkretnej dla każdego typu elementu dziedzinowego. Na rysunku widać, że każdy element posiada nazwę w postaci rzeczownika. Nie widać tam jednak sposobu przechowywania nazwy, co zostało dodatkowo zilustrowane na Rysunek 34 w postaci diagramu obiektów dla pojęcia „Appointment details".

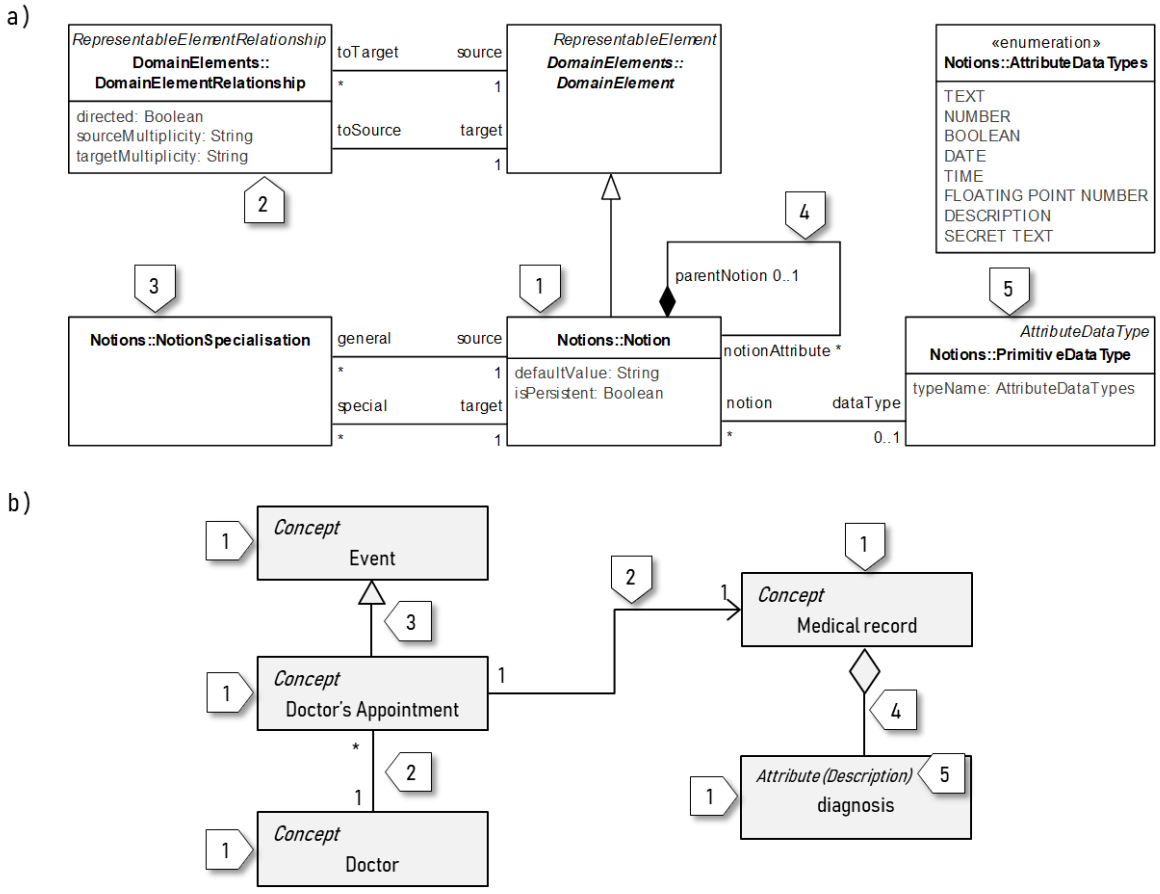


Rysunek 34 – Diagram obiektów ilustrujący składnię abstrakcyjnądla przykładowego elementu dziedzinowego

Najważniejszymi elementami dziedzinowymi są pojęcia reprezentowane przez metaklasę `Notion`. Pojęcia w języku RSL mogą modelować różnego rodzaju elementy z dziedziny problemu oraz dziedziny aplikacji (np. Concept, Data View, Screen, itp.). Do rozróżniania pojęć na poziomie semantyki wykorzystywane są stereotypy. Metaklasa `Notion` dziedziczy pośrednio po metaklasie `SCLElement` z pakietu `SCLKernel` (patrz rozdział 4.2), a zatem każde pojęcie może posiadać stereotyp (metaklasa `Stereotype`) określający znaczenie danego pojęcia.

Oprócz reprezentowania encji dziedzinowych, pojęcia mogą również modelować ich cechy behawioralne. Język RSL definiuje w tym celu tzw. stwierdzenia dziedzinowe. Każde pojęcie (`Notion`) może zawierać dowolną liczbę stwierdzeń dziedzinowych (`DomainStatement`), co jest wyrażone w metamodelu relacją kompozycji (Rysunek 33a). Element `DomainStatement`, z kolei, składa się z pojedynczej frazy, która pełni rolę jego nazwy (atrybut `name`). Istotne jest tutaj ograniczenie dotyczące wszystkich elementów typu

`DomainStatement` powiązanych z danym pojęciem `Notion` – ich nazwa (`name`) musi być frazą, która wskazuje na frazę rzeczownikową stanowiącą jednocześnie nazwę pojęcia, do którego należy `DomainStatement`. Zostało to zilustrowane na diagramie obiektów na Rysunek 34 oraz w przykładzie składni konkretnej dla pojęcia „Appointment details" na Rysunek 33b. Na przedstawionym diagramie obiektów widać, że fraza rzeczownikowa „appointment details" jest jednocześnie nazwą pojęcia oraz dopełnieniem (`object`) w prostych frazach czasownikowych („retrieve appointment data", „verify appointment data", „update appointment data") będących nazwami wyrażeń dziedzinowych dla danego pojęcia. W ten sposób, elementy `DomainStatement` mogą wyrażać czynności wykonywane na obiektach z dziedziny problemu oraz dziedziny aplikacji.



Rysunek 35 – Metamodel dla relacji między pojęciami dziedzinowymi: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Kompletny model dziedziny, oprócz samych elementów dziedzinowych, powinien odzwierciedlać zależności miedzy tymi elementami. W języku RSL zależności takie

reprezentowane są przez metaklasę `DomainElementRelationship` oraz `NotionSpecialization`. Odpowiedni fragment metamodelu przedstawiony jest na Rysunek 35a.

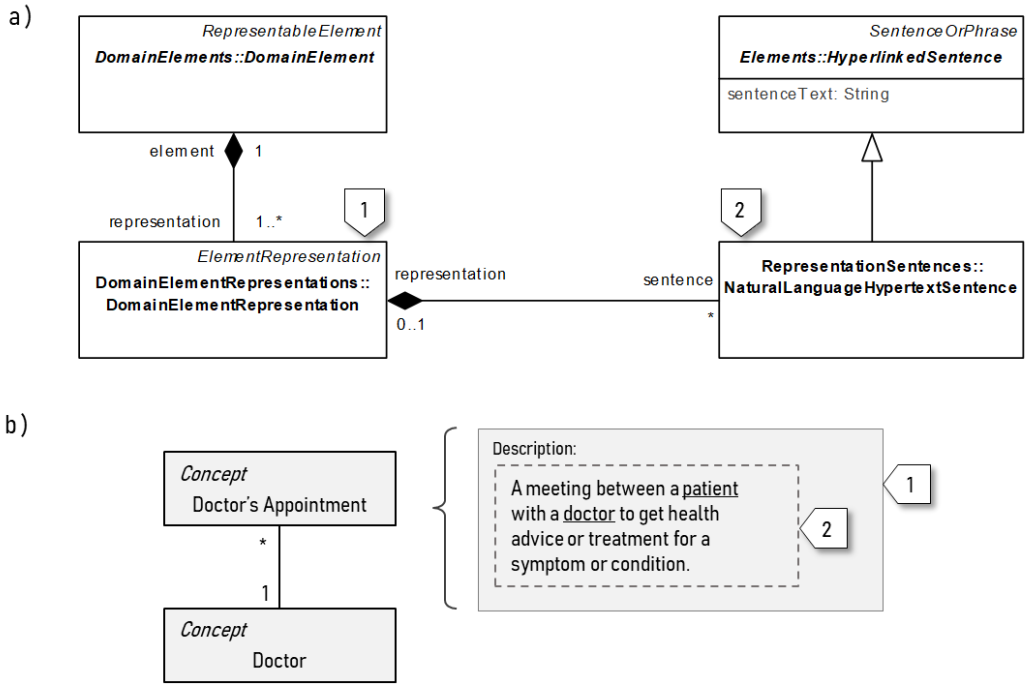
Relacja typu `DomainElementRelationship` łączy dwa elementy typu `DomainElement`, z których jeden pełni rolę źródła (`source`) a drugi – celu relacji (`target`). Relacja może być ukierunkowana lub nie – określa to wartość metaatrybutu `directed`. Pozostałe dwa atrybuty metaklasy `DomainElementRelationship` służą do określania krotności, pomiędzy powiązanymi elementami dziedziny. Są to `sourceMultiplicity` oraz `targetMultipplicity` – obydwa typu `String`. Umożliwia to stosowanie różnych wyrażeń określających krotność (m.in. zakresów liczbowych) bez nadmiernego komplikowania metamodelu. Przykłady składni konkretnej dla tego typu relacji oznaczone są numerem 2 na Rysunek 35b.

Drugim typem relacji jest specjalizacja (`NotionSpecialization`). Ma ona zastosowanie do pojęć (`Notion`) i pozwala modelować zależność specjalizacji (generalizacji) między nimi. Źródło relacji (atrybut `source`) wskazuje pojęcie ogólne, natomiast cel relacji (atrybut `target`) – pojęcie specjalizowane. W przykładzie składni konkretnej, przedstawionym na Rysunek 35b, pojęcie „Doctor's Appointment" jest specjalizacją bardziej ogólnego pojęcia „Event".

Każde pojęcie typu `Notion` może zawierać dowolną liczbę atrybutów. Atrybuty w języku RSL są reprezentowane przez tę samą metaklasę co pojęcia, czyli `Notion`. W odróżnieniu od pojęć, atrybuty muszą posiadać typ (metaatrybut `dataType`). Jest on określany poprzez metaklasę `PrimitiveDataType` zawierającą metaatrybut `typeName` typu wyliczeniowego `AttributeDataType`. Wszystkie podstawowe typy zdefiniowane przez `AttributeDataType` zostały pokazane na Rysunek 35a. Odpowiadają one typom atrybutów opisanych wcześniej.

Zawieranie atrybutu przez pojęcie zostało wyrażone w metamodelu poprzez relację kompozycji, która na Rysunek 35a oznaczona została numerem 4. W relacji tej, pojęcie reprezentujące atrybut występuje w roli `notionAttribute`, natomiast pojęcie zawierające atrybut – w roli `parentNotion`. Pojęcie zawierające atrybut przedstawiane jest na diagramie dziedzinowym w podobny sposób jak relacja między pojęciami (patrz przykład składni konkretnej na Rysunek 35b). Zawieranie atrybutu nie jest jednak relacją *sensu stricto* i nie posiada żadnych właściwości, dlatego nie posiada swojej reprezentacji w postaci metaklasy.

Istotnym uzupełnieniem przedstawionej składni abstrakcyjnej dla atrybutów pojęć są dwa ograniczenia. Pierwsze ograniczenie mówi, że pojęcie może zawierać inne pojęcie w roli `notionAttribute` tylko wtedy, gdy zawierane pojęcie jest atrybutem, tzn. posiada `dataType`. Drugie ograniczenie mówi, że pojęcie reprezentujące atrybut nie może zawierać innych atrybutów.

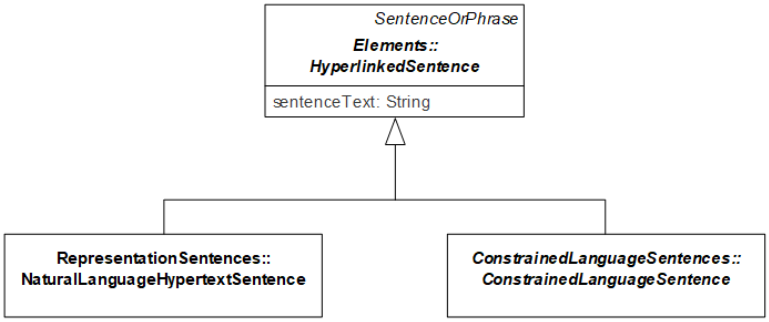


Rysunek 36 – Metamodel dla reprezentacji elementów dziedzinowych: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Wszystkie elementy typu `DomainElement` są podtypami metaklasy `RepresentableElement` i mogą posiadać swoje reprezentacje. W przypadku elementów dziedzinowych, każdy z nich posiada co najmniej jedną reprezentację w postaci `DomainElementRepresentation`, będącą specjalizacją metaklasy `ElementRepresentation` z pakietu `RSLKernel`. Reprezentacja taka może zawierać definicję elementu dziedzinowego wyrażoną w języku naturalnym w postaci hipertekstu. Stąd, jak widać w metamodelu na Rysunek 36a, reprezentacja elementu dziedzinowego jest sekwencją zdań typu `NaturalLanguageHypertextSentence`. Rysunek 36b przedstawia, z kolei, przykład takiej reprezentacji w postaci opisu z hiperłączami do pojęć powiązanych, co pomaga zapewnić wysoką spójność modelu.

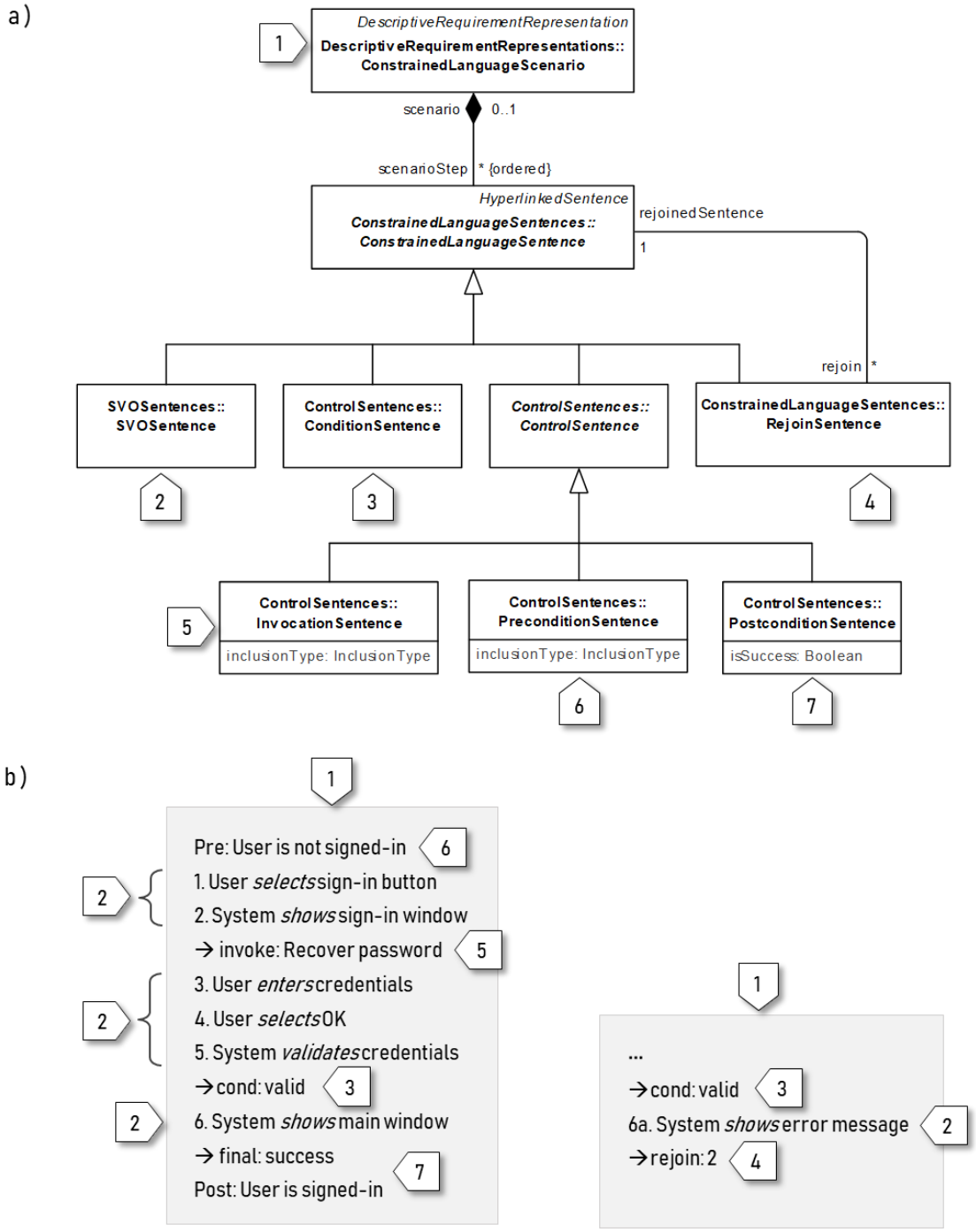
4.6 Zdania i scenariusze

Przedstawione powyżej elementy dziedzinowe wraz z powiązanymi frazami oraz terminologią, oprócz tego, że pozwalają tworzyć spójne modele dziedzinowe, służą również do reprezentowania wymagań funkcjonalnych. Wymagania funkcjonalne są szczególnie istotne z punktu widzenia semantyki translacyjnej, gdyż wyrażają logikę aplikacji, którą chcemy odzwierciedlić w automatycznie generowanym kodzie aplikacji. Aby umożliwić automatyczną transformację wymagań funkcjonalnych do kodu, język specyfikacji wymagań musi zapewnić nie tylko dobrze określona semantykę, ale również dobrze ustrukturyzowaną składnię abstrakcyjną. W języku RSL zostało to zrealizowane poprzez zastosowanie ograniczonego języka naturalnego (ang. constrained natural language) do reprezentowania przypadków użycia systemu.



Rysunek 37 – Metamodel dla zdań z hiperłączami

Rysunek 37 przedstawia specjalizacje zdań typu `HyperlinkedSentence`, czyli zdań mogących posiadać hiperłącza. Metaklasa `NaturalLanguageHypertextSentence` reprezentuje zdania w języku naturalnym o dowolnej strukturze, czego przykład pokazano na Rysunek 36. Abstrakcyjna metaklasa `ConstrainedLanguageSentence` reprezentuje, natomiast, zdania w ograniczonym języku naturalnym. Struktura tych zdań jest ściśle określona i zależy od konkretnego typu. Rysunek 38 przedstawia hierarchię zdań typu `ConstrainedLanguageSentence`. Doświadczenia praktyczne pokazują, że przedstawione typy zdań są wystarczające do opisywania logiki aplikacji w postaci tekstowych scenariuszy przypadków użycia.

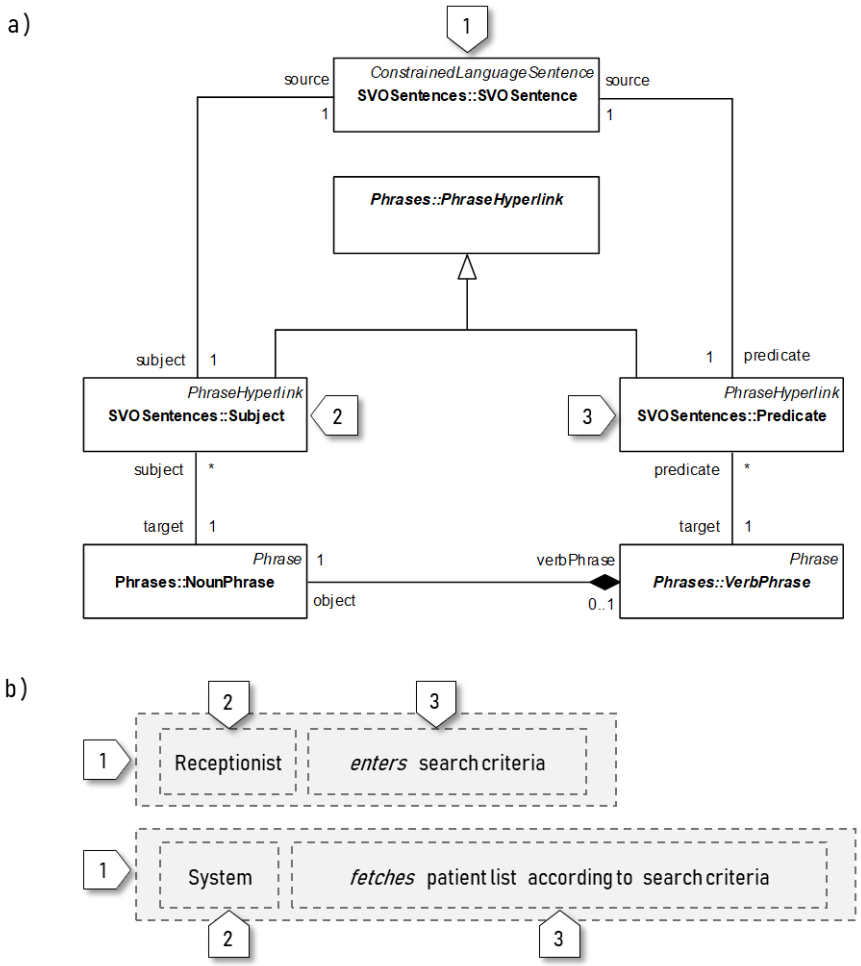


Rysunek 38 – Metamodel dla scenariuszy: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Scenariusze tekstowe w języku RSL reprezentowane są przez metaklasę `ConstarinedLanguageScenario` (Rysunek 38a). Każdy taki scenariusz składa się z dowolnej liczby uporządkowanych kroków (metaatrybut `scenarioStep`) wyrażonych zdaniami typu `ConstrainedLanguageSentence`. Przykłady składni konkretnej dla

scenariuszy (głównego i alternatywnego) wraz z odniesieniami do poszczególnych elementów metamodelu został pokazany na Rysunek 38b.

Język RSL nie narzuca określonej struktury scenariuszy, jednakże aby znacząco ułatwić automatyczną generację kodu należy przyjąć pewne ograniczenia w postaci reguł określających poprawne sekwencje zdań różnych typów w scenariuszu. Reguły te zostały opisane w rozdziale 3 i są niezbędne dla poprawnego działania transformacji „RSL to Java".



Rysunek 39 – Metamodel dla zdań typu SVO: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Najczęściej używanym typem zdań w scenariuszu jest `SVOSentence` (oznaczone numerem 2 na Rysunek 38). Jego składnię abstrakcyjną przedstawia Rysunek 39a. Zdania tego typu składają się z dwóch elementów: `Subject` oraz `Predicate`. Obie metaklasy są specjalizacją `PhraseHyperlink`, co oznacza, że pełnią rolę hiperlinków wskazujących na frazy. `Subject`, pełniąc rolę podmiotu w zdaniu, wskazuje na frazę rzeczownikową. `Predicate`, natomiast,

reprezentuje część zdania orzekającą o podmiocie, dlatego też wskazuje na frazę czasownikową. Może być to fraza czasownikowa prosta lub złożona, w zależności czy zdanie ma jedno czy dwa dopełnienia. Przykłady składni konkretnej dla obu przypadków przedstawia Rysunek 39b. Dzięki takiej budowie, zdania te odwołują się do elementów dziedzinowych a co za tym idzie, logika aplikacji wyrażona scenariuszami pozostaje całkowicie spójna z modelem dziedzinowym. Zdecydowanie ułatwia to zdefiniowanie semantyki translacyjnej oraz jej implementację w postaci transformacji „RSL to Java".

Pozostałe typy zdań używanych w scenariuszach przypadków użycia nie mają tak ściśle określonej struktury jak zdania `SVOSentence`, aczkolwiek są równie istotne dla wyrażenia pełnej logiki aplikacji. Wszystkie te zdania dziedziczą pośrednio metaatrybut `sentenceText` po metaklasie `HyperlinkedSentence` (Rysunek 37) a więc mogą zawierać dowolny tekst. Jest to wykorzystywane m.in. do wyrażania warunków początku i zakończenia scenariusza w zdaniach typu `PreconditionSentence` oraz `PostconditionSentence`, jak również do wyrażania warunku rozgałęzienia scenariusza za pomocą zdań typu `ConditionSentence`. Przykłady składni konkretnej dla tych typów zdań pokazano na Rysunek 38b.

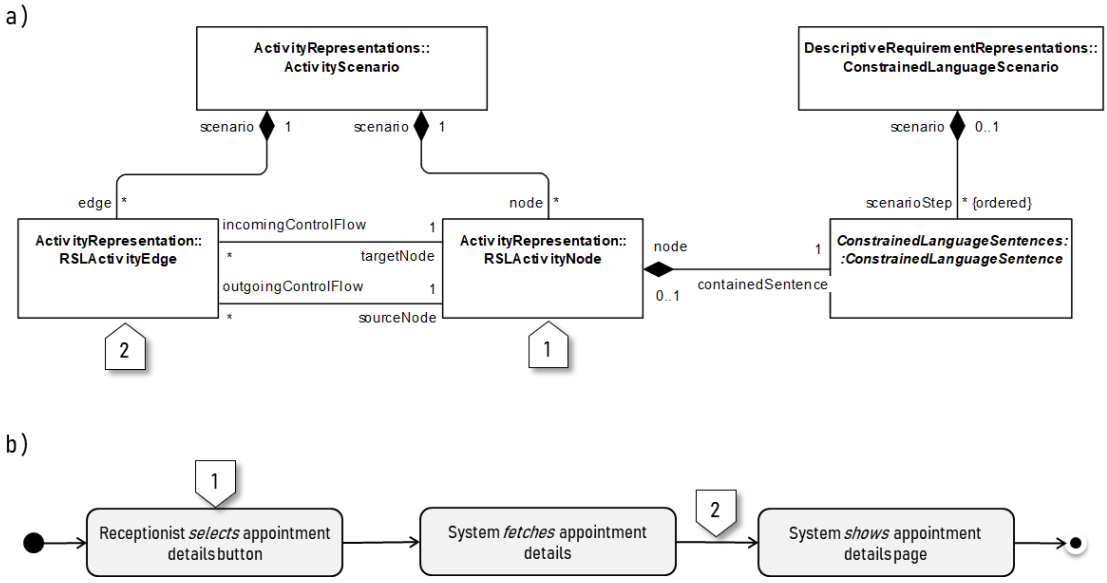
Możliwe jest zdefiniowanie dodatkowych reguł określających zawartość zdań używających `sentenceText`, co ułatwia ich parsowanie i wykorzystanie do generacji kodu. Na potrzeby transformacji „RSL to Java" takie reguły zostały określone a przykłady ich zastosowań są omówione na przykładach w rozdziale 7.

Zdania typu `RejoinSentence` oraz `InvocationSentence`, podobnie jak `SVOSentence`, nie muszą wykorzystują `sentenceText`, gdyż ich pozostała składnia definiuje wszystkie potrzebne elementy. W przypadku `RejoinSentence` wystarczającym elementem jest wskazanie na inne zdanie w scenariuszu alternatywnym, dzięki czemu zdanie `RejoinSentence` określa przepływ kontroli w przypadku łączenia scenariuszy (mechanizm ten został wyjaśniony w rozdziale 3). Zostało to wyrażone w metamodelu na Rysunek 38a za pomocą metaasocjacji łączącej metaklasę `RejoinSentence` oraz `ConstrainedLanguageSentence`.

Zdanie `InvocationSentence` ściśle łączy się z relacją typu „invoke" pomiędzy przypadkami użycia i zostanie omówione w rozdziale 4.8.

4.7 Notacja graficzna dla scenariuszy

Język RSL definiuje alternatywny sposób prezentacji scenariuszy przypadków użycia, tzn. w postaci grafów skierowanych, w których węzłami są zdania scenariusza a krawędzie określają ich kolejność. Składnia konkretna w przypadku tej reprezentacji zbliżona jest do diagramu aktywności w języku UML. Taki sposób reprezentowania scenariuszy ułatwia przedstawienie złożonych przypadków użycia (z wieloma scenariuszami alternatywnymi oraz zdaniami typu „rejoin"). Postać grafowa ułatwia także przetwarzanie scenariuszy przez transformacje, które muszą iterować po zdaniach scenariusza zgodnie z ich sekwencją, np. w przypadku transformacji „RSL to Java".

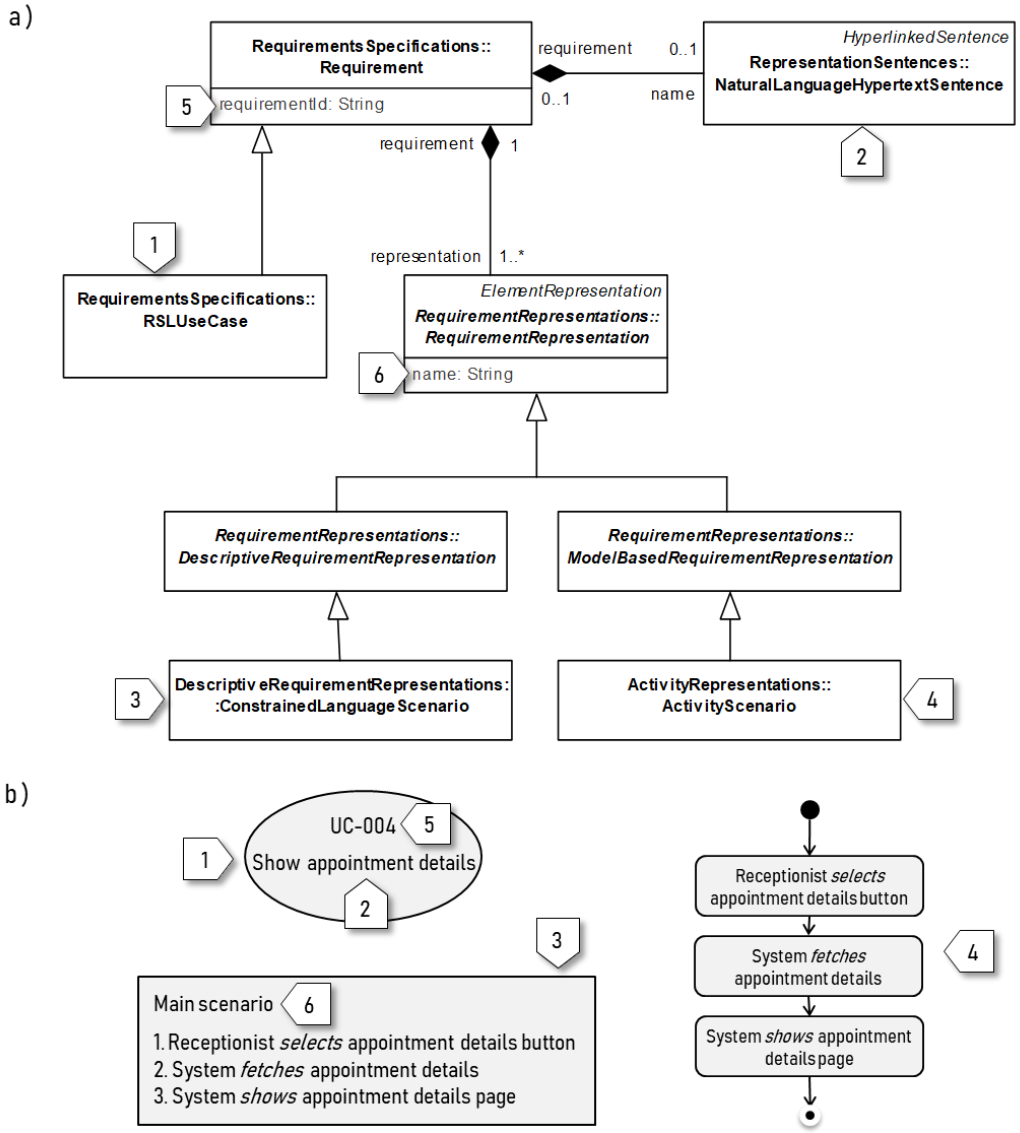


Rysunek 40 – Metamodel dla graficznej reprezentacji scenariuszy: składnia abstrakcyjna (a) oraz przykład składni konkretnej (b)

Rysunek 40a przedstawia fragment metamodelu dla graficznej reprezentacji scenariuszy. Węzły grafu reprezentowane są przez metaklasę `RSLActivityNode` a krawędzie przez `RSLActivityEdge`. Ponieważ węzły reprezentują zdania scenariusza, każdy z nich zawiera dokładnie jedno zdanie `ConstrainedNaturalLanguage`. Każdy węzeł może mieć, ponadto, dowolną liczbę krawędzi wychodzących (`outgoingControlFlow`) oraz wchodzących (`incomingControlFlow`), które określają sekwencję zdań w scenariuszu. Rysunek 40b przedstawia prosty przykład scenariusza dla opisywanej reprezentacji.

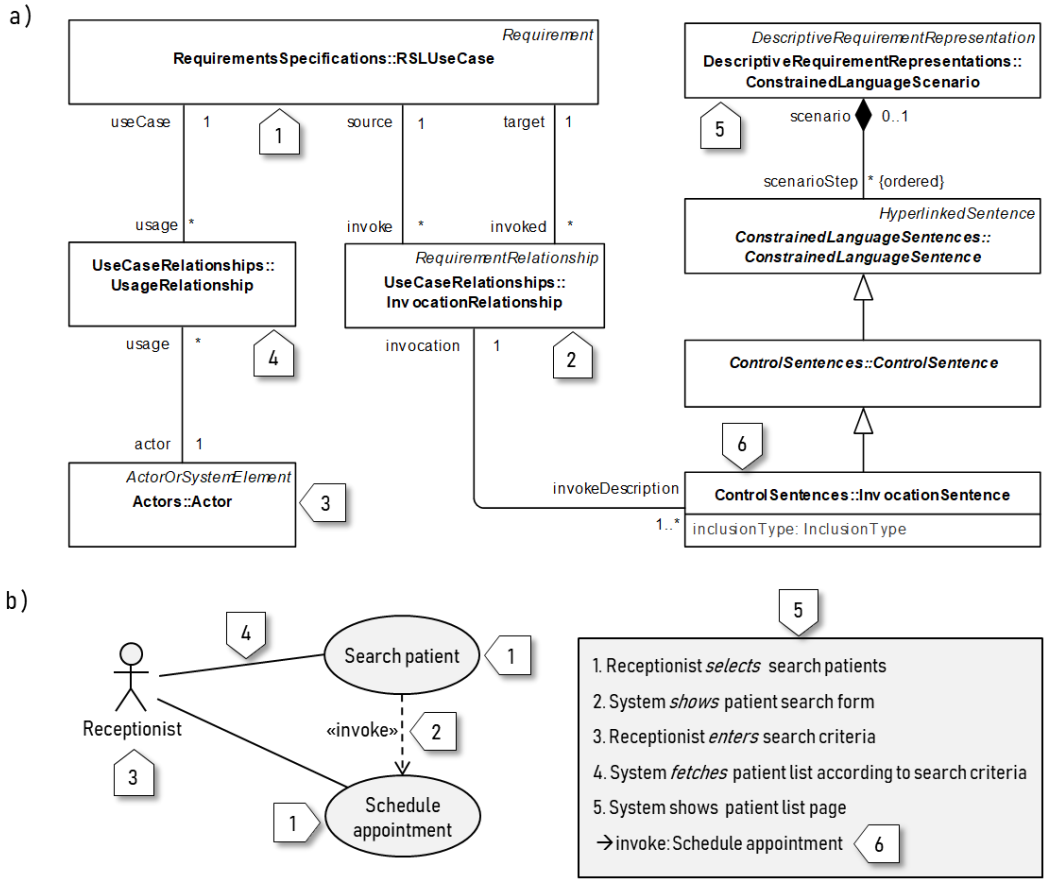
4.8 Przypadki użycia i ich zależności

Fragment metamodelu przedstawiony na Rysunek 41a definiuje wymagania (metaklasa `Requirement`) oraz ich reprezentacje (metaklasa `RequirementRepresentation`). Język RSL definiuje dwa rodzaje reprezentacji wymagań: opisowe (metakalsa `DescriptiveRequirementRepresentation`) oraz w postaci modelu (metaklasa `ModelBasedRequirementRepresentation`). Są to abstrakcyjne reprezentacje, które mają konkretne podtypy: `ConstrainedLanguageScenario` oraz `ActivityScenario`. Obie konkretne reprezentacje zostały przedstawione w powyższych rozdziałach. Wspólna nadklasa dla wszystkich rodzajów reprezentacji definiuje atrybut `name`, który przechowuje nazwę danej reprezentacji.



Rysunek 41 – Metamodel dla wymagań i przypadków użycia: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Każda reprezentacja musi być powiązana dokładnie z jednym wymaganiem a każde wymaganie musi mieć co najmniej jedną reprezentację. Właściwość ta została przedstawiona w metamodelu za pomocą relacji kompozycji pomiędzy metaklasą `Requirement` (kompozyt) oraz `RequirementRepresentation` (komponent). Co więcej, wymaganie może mieć jednocześnie reprezentacje różnych typów. Przykładowo, przypadek użycia, który jest szczególnym rodzajem wymagania, może mieć jednocześnie scenariusz w postaci tekstowej oraz w postaci modelu aktywności. Oprócz reprezentacji, wymaganie może mieć nazwę (metaatrybut `name`) oraz identyfikator (`requirementId`). Przykładowa notacja graficzna dla tak zdefiniowanej składni abstrakcyjnej jest przedstawiona na Rysunek 41b.



Rysunek 42 – Metamodel dla relacji przypadków użycia: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Z punktu widzenia semantyki translacyjnej opisanej w tej pracy, najważniejszym typem wymagania jest przypadek użycia reprezentowany przez metaklasę `RSLUseCase`. To właśnie

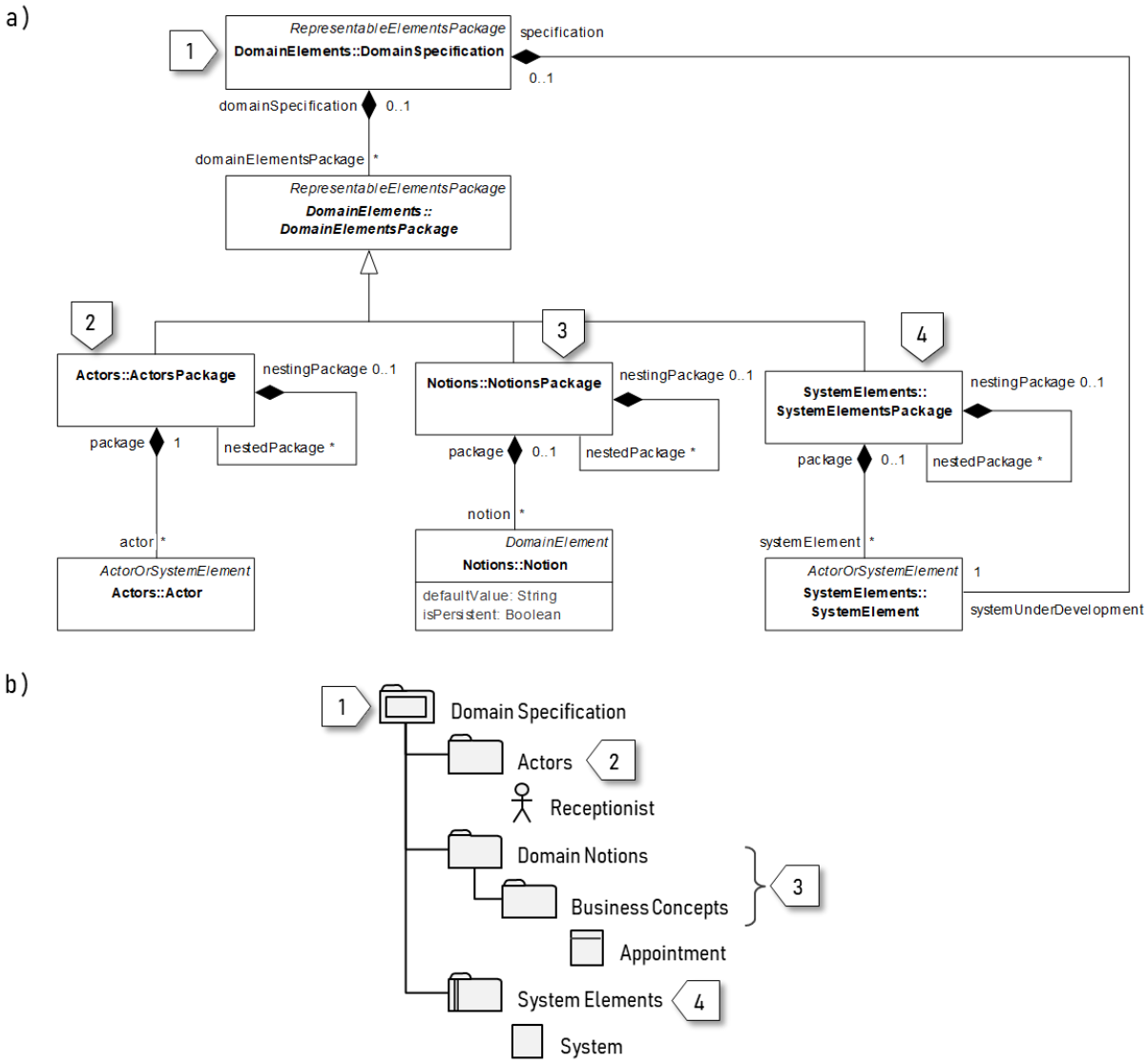
semantyka zdefiniowana dla przypadków użycia wraz z ich relacjami oraz reprezentacjami umożliwia automatyczną generację kodu logiki aplikacji.

Rysunek 42 przedstawia metamodel dla przypadków użycia. Oprócz samego przypadku użycia definiuje on dwa rodzaje relacji reprezentowane przez dwie metaklasy: `UsageRelationship` oraz `InvocationRelationship`. Pierwsza z nich jest relacją między aktorem a przypadkiem użycia i oznacza, że dany aktor bierze udział w wykonaniu danego przypadku użycia – w scenariuszach nazwa takiego aktora wyrażona za pomocą frazy rzeczownikowej pojawia się jako podmiot w zdaniach `SVOSentence`, które wyrażają akcje wykonywane przez aktora. Druga metaklasa reprezentuje relację „invoke" między przypadkami użycia. Każdy przypadek użycia może mieć dowolną liczbę przychodzących i wychodzących relacji „invoke".

Istotnym elementem relacji `InvocationRelationship` jest jej bezpośrednie powiązanie ze zdaniem `InvocationSentence` wyrażone w metamodelu metaasocjacją pomiędzy tymi metaklasami. Wynika z niej, że każde zdanie typu „invoke" wskazuje na dokładnie jedną relację typu „invoke", natomiast każda taka relacja może być powiązana z co najmniej jednym zdaniem typu „invoke". W przypadku powiązania relacji z więcej niż jednym zdaniem, zdania te mogą występować w jednym lub wielu scenariuszach przypadku użycia, z którego wychodzi dana relacja „invoke". Przedstawiony metamodel należy uzupełnić o ograniczenie, które mówi, że zdanie „invoke" powiązane z daną relacją „invoke" musi być częścią scenariusza będącego reprezentacją przypadku użycia, z którego ta relacja wychodzi. Tak zdefiniowana składnia abstrakcyjna zapewnia wysoką spójność pomiędzy przypadkami użycia i ich scenariuszami. Dzięki ściśle określonemu przepływowi sterowania, możliwa jest automatyczna i jednoznaczna generacja kodu logiki aplikacji.

4.9 Specyfikacja dziedziny i specyfikacja wymagań

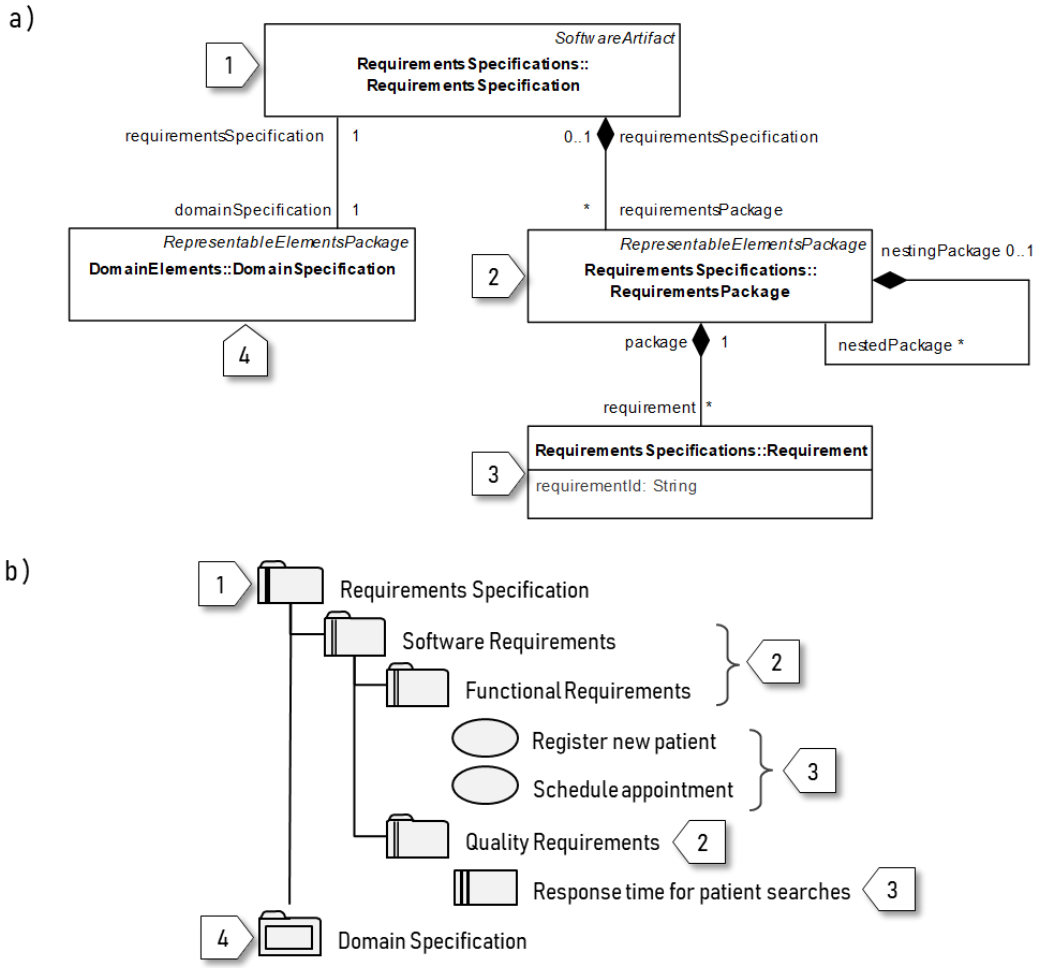
Wszystkie opisane powyżej elementy języka RSL są grupowane za pomocą pakietów. Język RSL definiuje kilka rodzajów pakietów – każdy przeznaczony do przechowywania elementów określonych typów. Dzięki temu specyfikacje w języku RSL mają precyzyjną i czytelną strukturę. Na najwyższym poziomie, cała specyfikacja w języku RSL jest podzielona na dwie części: część przechowującą elementy dziedzinowe oraz część przechowującą wymagania i ich reprezentacje.



Rysunek 43 – Metamodel dla specyfikacji dziedziny: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Rysunek 43 przedstawia metamodel oraz składnię konkretną dla pierwszej części. Metaklasa `DomainSpecification` definiuje pakiet najwyższego poziomu w drzewie specyfikacji dziedzinowej. Pakiet ten nie zawiera bezpośrednio elementów dziedzinowych. Elementy te są grupowane względem typu w odpowiednich pakietach na niższym poziomie: aktorzy w `ActorsPackage`, pojęcia w `NotionsPackage` oraz elementy systemowe w `SystemElementsPackage`. Wszystkie te trzy pakiety są specjalizacją abstrakcyjnej metaklasy `DomainElementsPackage`. Zarówno `DomainSpecification` jak i `DomainElementsPackage` dziedziczą po metaklasie `RepresentableElementsPackage` i pośrednio po `SCLElementsPackage` (patrz 4.3) a zatem mogą posiadać nazwę (metaatrybut `name`). Ponadto, cechą wszystkich specjalizacji `DomainElementsPackage` jest to, że mogą

posiadać dowolną strukturę podpakietów tego samego typu. Zostało to wyrażone w metamodelu za pomocą autoasocjacji, gdzie pakiet nadrzędny występuje w roli `nestingPackage` a pakiet podrzędny – `nestedPackage`. Syntaktyka języka RSL nie wprowadza ograniczenia na liczbę pakietów `DomainElementsPackage` zawartych w pakiecie nadrzędnym `DomainSpecification`. Ograniczenie takie może być jednak narzucone przez narzędzie implementujące język RSL: w przypadku narzędzia ReDSeeDS liczba ta jest ograniczona do jednego pakietu każdego typu. Zwiększa to przejrzystość specyfikacji i ułatwia implementację transformacji „RSL to Java".



Rysunek 44 – Metamodel dla specyfikacji wymagań: składnia abstrakcyjna (a) oraz przykłady składni konkretnej (b)

Specyfikacja dziedziny jest ściśle powiązana ze specyfikacją wymagań. Razem stanowią one pełną specyfikację systemu oprogramowania w języku RSL. W metamodelu przedstawionym na Rysunek 44a związek ten jest wyrażony asocjacją jeden do jednego pomiędzy metaklasą

`DomainSpecification` i `RequirementsSpecification`. Należy tutaj podkreślić, że faktyczne powiązanie elementów dziedzinowych oraz wymagań i ich reprezentacji odbywa się za pomocą fraz z hiperłączami, co zostało pokazane w powyższych podrozdziałach.

Wewnętrzna struktura specyfikacji wymagań jest zbliżona do struktury specyfikacji dziedzinowej. Wymagania (`Requirements`) grupowane są w pakiety wymagań (`RequirementsPackage`), które posiadają nazwy oraz mogą tworzyć hierarchiczne struktury (`nestingPackage` i `nestedPackage`). Rysunek 44b ilustruje przedstawioną składnię abstrakcyjną dla specyfikacji wymagań przykładami składni konkretnej.

5 Semantyka translacyjna dla języka RSL

Semantyka języka RSL została przedstawiona w rozdziale 3 w sposób nieformalny, poprzez bezpośrednie mapowanie elementów języka na dziedzinę semantyczną, którą w tym przypadku jest logika aplikacji oraz encje biznesowe i systemowe przetwarzane w ramach tej logiki. Przedstawienie semantyki w taki sposób jest wystarczające aby umożliwić zrozumienie języka przez twórców i odbiorców specyfikacji wymagań. W celu implementacji automatycznej transformacji „RSL to Java", konieczne jest formalne zdefiniowanie semantyki poszczególnych elementów języka.

W literaturze znaleźć można kilka ugruntowanych metod pozwalających w sposób formalny definiować semantykę języków używanych w inżynierii oprogramowania [149] [150] [151]. Wiele z tych metod posługuje się aparatem matematycznym – bardzo precyzyjnym ale jednocześnie złożonym, co może rodzić trudności podczas implementacji języka. Przykładem takiego podejścia jest semantyka denotacyjna [152] [153]. Istnieją również bardziej pragmatyczne metody inżynierskie [154]. Pozwalają one na definiowanie semantyki języka w sposób wystarczająco formalny a jednocześnie łatwiejszy w zrozumieniu oraz implementacji. W przypadku języków zdefiniowanych w oparciu o metamodele, istnieją dwie podstawowe metody definiowania semantyki: semantyka operacyjna (ang. operational semantics) oraz semantyka translacyjna (ang. translational semantics) [30] [155] [150]. W niniejszej pracy wykorzystane zostało to drugie podejście.

Definiowanie semantyki translacyjnej polega na definiowaniu semantyki języka poprzez tłumaczenie go na inny język o dobrze określonej semantyce. Składnię abstrakcyjną języka źródłowego mapuje się na składnię abstrakcyjną języka docelowego. Semantyka translacyjna jest więc zbiorem precyzyjnych jednoznacznych reguł mówiących jak przekształcać poszczególne konstrukcje gramatyczne języka źródłowego na konstrukcje języka docelowego. Istotne jest przy tym, aby język docelowy był w stanie wyrazić wszystkie konstrukcje języka źródłowego. Przyjmując, że język docelowy posiada dobrze zdefiniowaną semantykę (czy to za pomocą metod formalnych czy nieformalnie), semantyka języka źródłowego może być wyprowadzona z semantyki języka docelowego, gdyż każdy model lub program w języku źródłowym może być jednoznacznie przekształcony w model lub program w języku docelowym. W trakcie tego przekształcenia, zmienia się składnia abstrakcyjna natomiast semantyka nie ulega zmianie.

Dobrym przykładem zastosowania semantyki translacyjnej jest definiowanie wysokopoziomowego imperatywnego języka programowania poprzez jego translację na język assemblera. Każda instrukcja języka wysokopoziomowego jest tłumaczona na zestaw dobrze zdefiniowanych instrukcji niskopoziomowych. Tak zdefiniowana semantyka języka wysokopoziomowego ułatwia implementację jego kompilatora. Wynika to z łatwości przekształcenia języka docelowego użytego do zdefiniowania semantyki języka źródłowego, na język docelowy kompilatora, czyli język wykonywalny.

5.1 Środowisko translacyjne

Język RSL jest językiem wyższego poziomu niż języki programowania trzeciej generacji (ang. third-generation language, 3GL), np. C++, Java czy Python. Języki trzeciej generacji mają dobrze określoną semantykę a ich kompilatory czy interpretery dostępne są dla wszystkich popularnych platform, co czyni je najczęściej używanymi językami w programowaniu aplikacyjnym. Biorąc pod uwagę powyższe oraz fakt, że RSL jest językiem dedykowanym do specyfikowania funkcjonalności oprogramowania aplikacyjnego, naturalnym wydaje się zdefiniowanie semantyki języka RSL w oparciu o język trzeciej generacji. Podejście takie jest szczególnie praktyczne jeśli chcemy zaimplementować program do automatycznej translacji modelu wymagań do kodu modelowanej aplikacji w języku programowania i – w wyniku kompilacji uzyskanego kodu źródłowego – do kodu wykonywalnego.

Na potrzeby tej pracy, jako język docelowy wybrany został język Java – obecnie jeden z najpopularniejszych[22] obiektowych języków programowania ogólnego przeznaczenia. Należy tutaj zaznaczyć, że transformacja „RSL to Java” jako język docelowy wykorzystuje UML. Dzięki temu, reguły translacji zaimplementowane w języku MOLA posługują się dobrze zdefiniowanym metamodelem języka UML (w zakresie definiującym model klas) zamiast bardziej złożonego i nieustandaryzowanego metamodelu języka Java [156]. Kod metod jest przy tym osadzony w elementach modelu w postaci tekstowej. Wygenerowany w wyniku transformacji model klas jest następnie przekształcany w kompletny kod w języku Java. Takie przekształcenie jest powszechnie stosowane w wielu narzędziach typu CASE (ang. Computer Aided Software Engineering) i nie będzie tutaj omawiane. Dodatkową zaletą takiego podejścia

[22] Na podstawie indeksu TIOBE: https://www.tiobe.com/tiobe-index/

jest to, że wynikiem transformacji, oprócz finalnie wygenerowanego kodu, jest również jego dokumentacja w języku UML.

Przedstawione w tym rozdziale środowisko translacyjne oraz reguły translacji będą ilustrowane częściowo za pomocą języka UML a częściowo za pomocą języka Java.

Definiując sematykę translacyjną dla Języka RSL, oprócz wyboru docelowego języka programowania, istotny jest wybór platformy programistycznej dla tego języka (ang. software framework), która implementuje określony wzorzec architektoniczny, tj. strukturę aplikacji oraz ogólny mechanizm jej działania. Dzięki temu, dokonując translacji konstrukcji języka RSL na konstrukcje docelowe zgodne z konwencjami przyjętymi dla wybranej platformy programistycznej, będziemy mogli wygenerować gotowy do skompilowania i uruchomienia kod aplikacji (choć niekoniecznie kompletny, np. w zakresie logiki biznesowej) zgodny ze specyfikacją wymagań.

Praktycznie każda z dostępnych platform programistycznych przeznaczonych do tworzenia interaktywnych aplikacji biznesowych realizuje pewne ogólne wzorce architektoniczne [157] [158]. Najczęściej jest to Model-View-Controller [159] lub jego odmiana Model-View-Presenter [160]. W przypadku translacji języka RSL do kodu aplikacji, bardziej odpowiednim wzorcem wydaje się Model-View-Presenter (MVP), ze względu na lepiej określoną odpowiedzialność poszczególnych warstw oraz bardziej klarowną komunikację między nimi. W przypadku MVP, warstwa modelu (Model) pełni jedynie rolę interfejsu do danych: zapewnia dostęp do źródeł danych oraz wykonuje operacje na danych zgodnie ze zdefiniowaną logiką biznesową. W odróżnieniu od wzorca MVC, model nie komunikuje się bezpośrednio z warstwą widoku (View), dzięki czemu nie musi posiadać mechanizmów aktywnego powiadamiania widoku o zmianie stanu. Tę funkcję przejmuje warstwa prezentera (Presenter), która z jednej strony zleca warstwie modelu operacje na danych a z drugiej strony zarządza stanem widoku: przekazuje mu do wyświetlenia dane pobrane z modelu oraz implementuje logikę obsługi zdarzeń przechwyconych przez widok. Warstwa widoku stanowi element bezpośredniej interakcji z użytkownikiem: wyświetla dane oraz przechwytuje akcje użytkownika mające wyzwolić określone zachowanie systemu.

Widzimy, że prezenter we wzorcu MVP definiuje sekwencje interakcji pomiędzy aktorem i systemem, abstrahując od szczegółów związanych z modelem danych oraz sposobem wizualnej prezentacji tych danych. Taki aktywny prezenter jest dobrym odzwierciedleniem logiki aplikacji wyrażonej w scenariuszach przypadków użycia. Stąd, wzorzec MVP wydaje

się najbardziej naturalnym wyborem jako środowisko docelowe dla semantyki translacyjnej języka RSL. Dodatkową zaletą wzorca MVP jest możliwość wymiany warstwy widoku bez konieczności zmian w warstwie prezentera. Dzięki temu można w łatwy sposób stworzyć aplikacje działające według tej samej logiki na różne platformy, np. aplikację internetową, aplikację mobilną czy klasyczną aplikację desktopową.

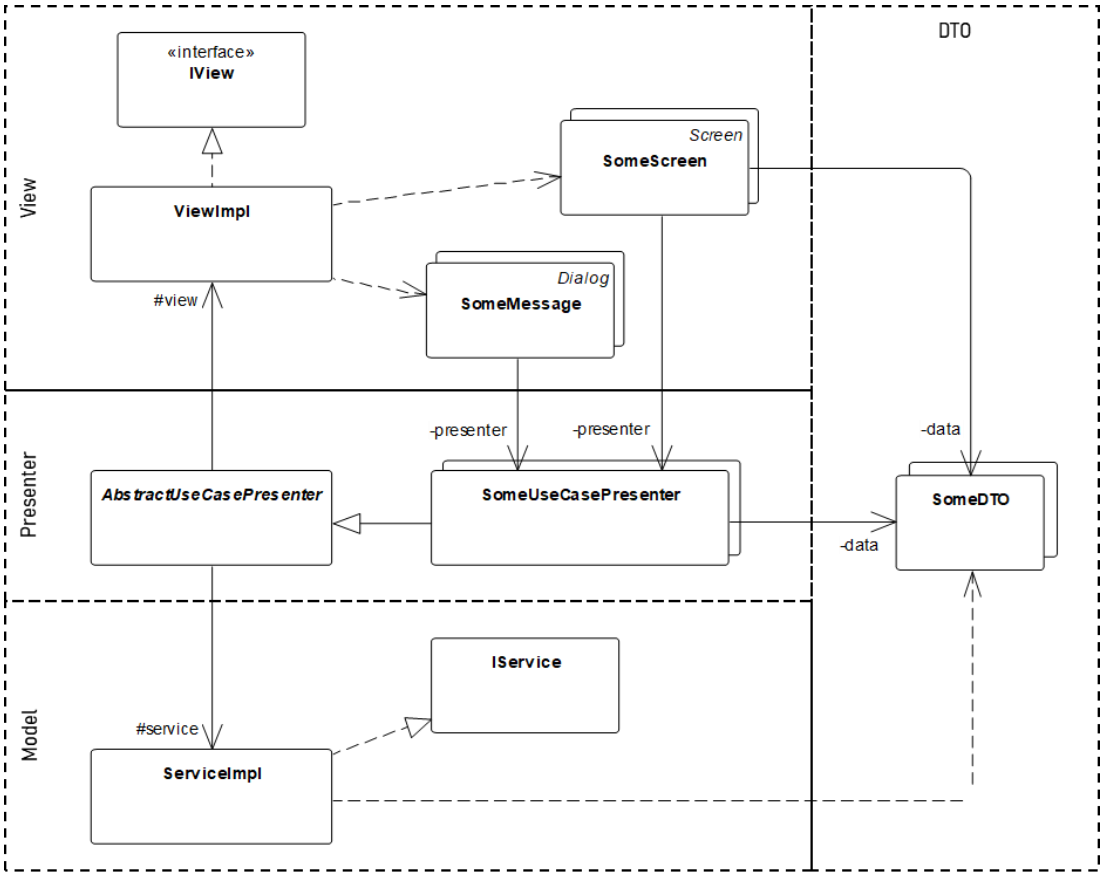
Dostępnych jest wiele platform programistycznych dla języka Java, które umożliwiają użycie wzorca MVP, m.in. JavaFX[23], Google Web Toolkit[24], Echo3[25] czy Vaadin[26]. Na potrzeby definicji semantyki translacyjnej języka RSL nie będziemy jednak odnosić się do żadnej konkretnej platformy i specyficznych dla niej konstrukcji czy konwencji. Zamiast arbitralnie wybranej platformy, posłużymy się pewnym uogólnionym środowiskiem. Pozwoli to zdefiniować reguły translacji na takim poziomie abstrakcji, aby można było je z łatwością zastosować w odniesieniu do konkretnej platformy wykorzystującej strukturę MVP. Przykładowa implementacja tych reguł dla platformy Echo3 zostanie przedstawiona w rozdziale 6.

[23] https://openjfx.io/

[24] http://www.gwtproject.org/

[25] http://echo.nextapp.com/

[26] https://vaadin.com/



Rysunek 45 – Środowisko translacyjne – struktura ogólna

Rysunek 45 przedstawia ogólną strukturę klas przyjętego środowiska translacyjnego, zgodnego ze wzorcem MVP. Klasy w warstwie widoku reprezentują ekrany oraz komunikaty prezentowane użytkownikowi. Są one specjalizacjami odpowiednich klas definiujących podstawowe właściwości i zachowanie tego typu elementów dla konkretnego środowiska programistycznego. W naszym abstrakcyjnym środowisku są to klasy `Screen` oraz `Dialog`, które w rzeczywistości będą mapowały się na klasy specyficzne dla użytego środowiska konkretnego, np. `JFrame` oraz `JDialog` w przypadku środowiska Java Swing czy `ContentPane` oraz `WindowPane` w przypadku Echo3. Dodatkowo, warstwa widoku zawiera klasę `ViewImpl` będącą realizacją interfejsu `IView`, który definiuje zachowanie widoku dostępne dla warstwy prezentera. Tym samym, prezentery są niezależne od technologii użytej w warstwie interfejsu użytkownika.

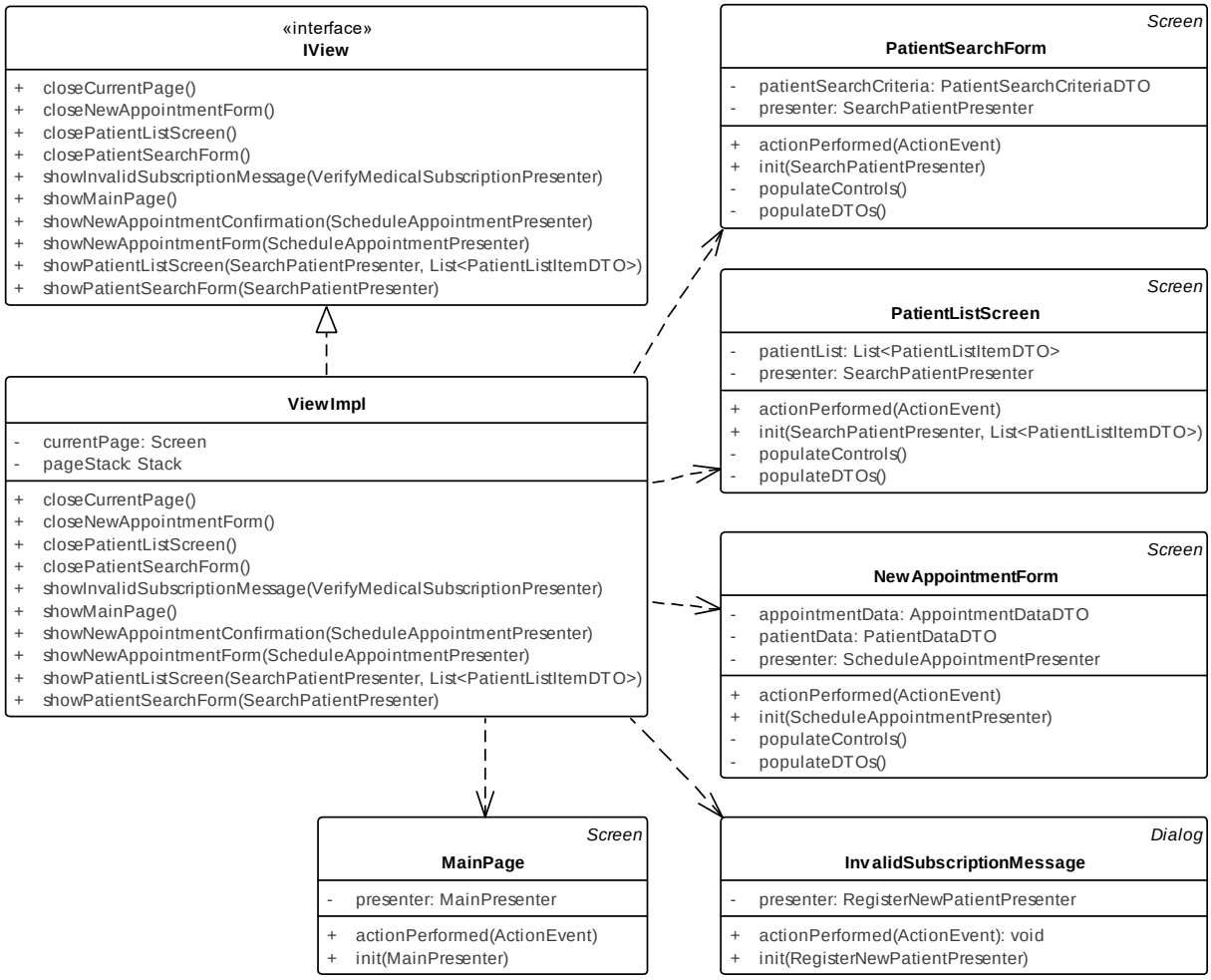
Klasy w warstwie prezentera implementują logikę aplikacji zdefiniowaną przez przypadki użycia. Każda klasa prezentera dziedziczy pewien zestaw wspólnych właściwości i operacji z abstrakcyjnej klasy `AbstractUseCasePresenter`, m.in. dostęp do implementacji interfejsów `IView` oraz `IService`. Ten ostatni udostępnia prezenterowi zestaw operacji

realizujących niezbędną logikę biznesową systemu oraz umożliwia przesyłanie danych z i do warstwy persystencji danych. Model jest pasywny i pełni rolę usługową w stosunku do warstwy prezentacji.

Dane pomiędzy warstwami przesyłane są w postaci obiektów transferu danych (ang. Data Transfer Object, DTO) [161]. Obiekty te mogą być tworzone w warstwie modelu na żądanie prezentera i przekazywane do warstwy widoku w celu ich prezentacji użytkownikowi. Mogą być też tworzone w warstwie widoku, na podstawie danych wprowadzonych przez użytkownika i przekazywane do modelu za pośrednictwem prezentera. Są to proste obiekty, które definiują zestaw atrybutów do przechowywania wartości danych a także metody do ustawiania (ang. setters) oraz odczytu (ang. getters) tych wartości. Mogą również zawierać metody do serializacji danych. Nie implementują, natomiast, żadnej logiki biznesowej. W odróżnieniu od obiektów dziedzinowych, obiekty transferu danych zawierają jedynie minimalny zestaw atrybutów, niezbędnych do wyświetlania lub pobierania danych w warstwie widoku.

Poniżej przedstawiona jest bardziej szczegółowa struktura klas w każdej warstwie modelu MVP. Dla większej czytelności przedstawione przykłady są realizacją prostego modelu przypadków użycia przedstawionego w rozdziale 3.3.3 na Rysunek 24.

Rysunek 46 przedstawia strukturę klas w warstwie widoku. Interfejs `IView` definiuje zestaw operacji do wyświetlania i ukrywania elementów interfejsu użytkownika: ekranów aplikacji oraz komunikatów i okien dialogowych, np. `showPatientSearchForm()` i `closePatientSearchForm()`. Implementacja tego interfejsu w postaci klasy `ViewImpl` jest wykorzystywana przez klasy prezenterów w celu realizacji logiki aplikacji, zgodnie z treścią scenariuszy przypadków użycia. Parametrem każdej metody `show…()` jest obiekt prezentera. Jest on przekazywany dalej do obiektu reprezentującego ekran lub komunikat, którego dotyczy dana metoda `show…()`, aby umożliwić informowanie prezentera o zdarzeniach na interfejsie użytkownika. Jeżeli jakiś ekran służy do prezentacji danych pobranych z modelu, to dane te muszą być przekazane do obiektu reprezentującego ten ekran w postaci obiektu DTO, np. jako dodatkowy parametr metody `show…()`. Oprócz realizacji operacji interfejsu `IView`, klasa `ViewImpl` przechowuje również stan widoku w postaci stosu aktualnie wyświetlanych ekranów aplikacji. Wywołanie metody `show…()` powoduje odłożenie na stos poprzednio wyświetlanego ekranu, natomiast wywołanie metody `close…()` zamyka bieżący ekran i przywraca ekran ostatnio odłożony na stos. Należy zaznaczyć, że mechanizm ten może być zależny od konkretnej technologii użytej do realizacji warstwy widoku.



Rysunek 46 – Środowisko translacyjne – struktura warstwy widoku

Klasy `PatientSearchForm`, `PatientListScreen`, `NewAppointmentForm` oraz `InvalidSubscriptionMessage` widoczne na Rysunek 46 są implementacją ekranów oraz komunikatów prezentowanych podczas pracy z aplikacją. Każda taka klasa zawiera zbiór pól i kontrolek służących do wyświetlania lub edycji danych oraz wyzwalania akcji systemu (np. pola tekstowe, listy rozwijane, przełączniki, tabele, przyciski, itp.), zgodnie ze specyfikacją danego ekranu w języku RSL. Pola i kontrolki stanowią zazwyczaj atrybuty klas reprezentujących ekrany. Ich typy i sposób inicjalizacji jest silnie uzależniony od konkretnego środowiska programistycznego i dostępnych w nim komponentów interfejsu użytkownika, stąd zostały one pominięte w prezentowanym przykładzie odnoszącym się do środowiska abstrakcyjnego.

Każda klasa dziedzicząca z nadklasy `Screen` oraz `Message` implementuje dwie publiczne operacje: `init()` oraz `actionPerformed()`. Pierwsza z nich zawiera kod inicjujący okno bądź

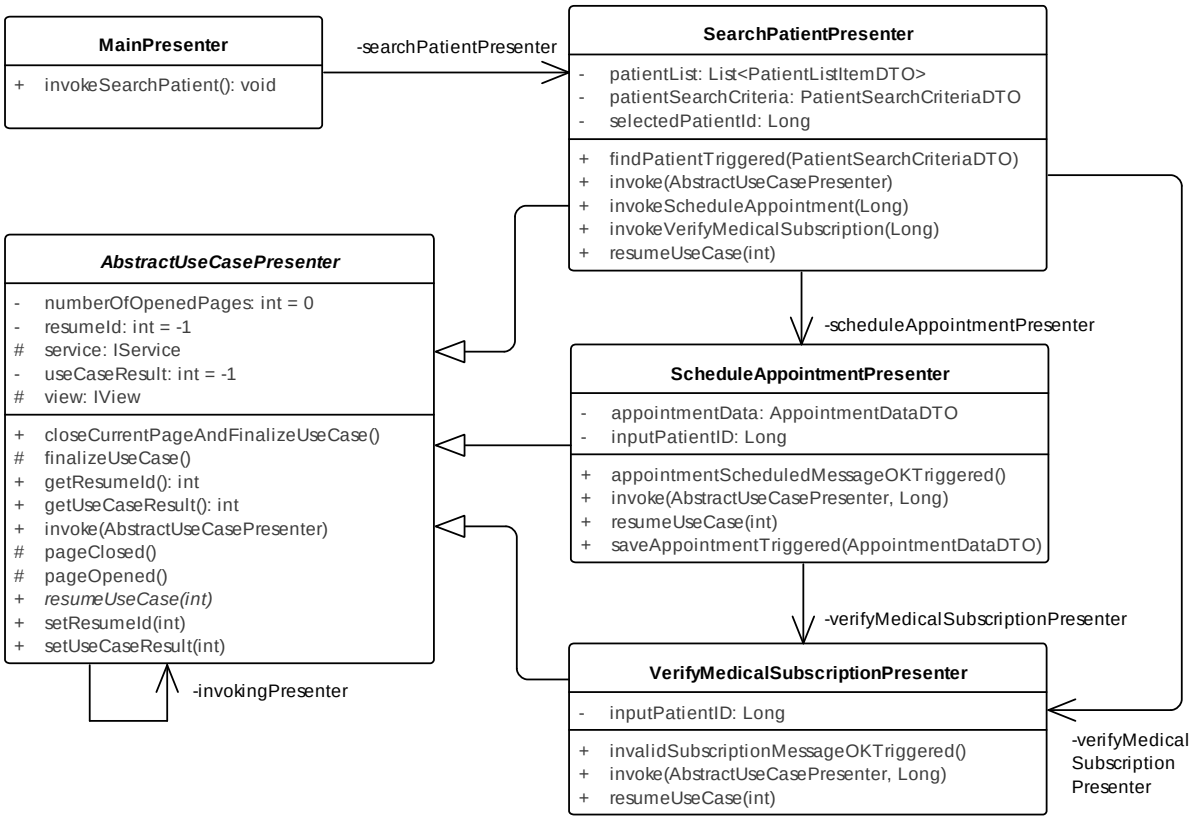
ekran poprzez utworzenia i rozmieszczenie na nim wszystkich pól i kontrolek. Warto tutaj zaznaczyć, że w obecnej wersji język RSL nie pozwala określać sposobu rozmieszczenia (ang. layout) kontrolek na ekranach, co oznacza, że wygenerowany kod powinien używać jednego z dostępnych w danym środowisku sposobów domyślnego rozmieszczania komponentów interfejsu użytkownika. W przypadku, gdy przeznaczeniem ekranu jest prezentacja danych użytkownikowi, dane te przekazywane są przez warstwę prezentera w postaci obiektu lub obiektów DTO. Przekazanymi danymi inicjalizowane są następnie odpowiednie pola i kontrolki ekranowe, za co odpowiada pomocnicza metoda `populateControls()`.

Metoda `actionPerformed()` zawiera kod obsługujący zdarzenia dotyczące elementów interfejsu użytkownika. Jest ona wywoływana zawsze wtedy, gdy na kontrolce związanej z danym ekranem zostanie wykonana akcja. Informacja o kontrolce oraz rodzaju akcji przekazywana jest jako parametr operacji. W przypadku akcji użytkownika mających na celu wyzwolenie reakcji systemu zgodnie ze zdefiniowaną logiką aplikacji, kod obsługujący to zdarzenie polega na przekazaniu sterowania do warstwy prezentera poprzez wywołanie odpowiedniej metody na powiązanym obiekcie prezentera zarządzającego danym ekranem (atrybut `presenter`). Jeżeli akcja wymaga przetworzenia danych wprowadzonych lub zmodyfikowanych przez użytkownika, wartości wprowadzone w polach edycji są odczytywane i mapowane na odpowiednie atrybuty obiektu lub obiektów DTO w celu przekazania do prezentera. Odpowiada za to pomocnicza metoda `populateDTOs()`.

Warstwa widoku zawiera również klasę `MainPage`, która reprezentuje główny ekran aplikacji, zawierający podstawowe menu. Z jego poziomu użytkownik ma możliwość wyzwolenia podstawowych funkcji system przewidzianych w modelu przypadków użycia. W omawianym przykładzie jest to funkcja wyszukiwania pacjentów „Search patient”.

Klasy tworzące warstwę prezentera przedstawione są na Rysunek 47. Każda konkretna klasa prezentera (oprócz `MainPresenter`) odpowiada jednemu przypadkowi użycia i implementuje zdefiniowaną dla niego logikę aplikacji. Wszystkie te klasy mają wspólną nadklasę `AbstractUseCasePresenter`, która zapewnia dostęp do implementacji interfejsu `IView` oraz `IService` (atrybuty `view` oraz `service`) a także ogólny mechanizm przepływu sterowania podczas wywołania jednego przypadku użycia przez inny, zgodnie z relacją „invoke”. Podstawową operacją w klasie `AbstractUseCasePresenter` jest `invoke()`, która inicjuje wykonanie każdego przypadku użycia. Parametrem tej operacji jest obiekt prezentera odpowiadający wywołującemu przypadkowi użycia. Referencja do tego obiektu

przechowywana jest w atrybucie `invokingPresenter`, aby po wykonaniu całej logiki zdefiniowanej dla danego przypadku użycia oddać kontrolę do prezentera przypadku wywołującego, jeśli taki istnieje. Dzieje się to w ramach metody `finalizeUseCase()`, w której na obiekcie `invokingPresenter` wywoływana jest metoda `resumeUseCase()`. Jej parametrem jest końcowy rezultat wykonania przypadku wywołanego (atrybut `useCaseResult`), gdyż może on mieć wpływ na dalszy przebieg przypadku wywołującego. Atrybut `resumeId` jest ściśle powiązany z metodą `resumeUseCase()`. Z punktu widzenia prezentera przypadku wywołującego, `resumeId` zapamiętuje kontekst wywołania innego przypadku użycia, aby po jego zakończeniu metoda `resumeUseCase()` wznowiła wykonywanie pierwotnego przypadku użycia we właściwym miejscu. Jest to istotne w sytuacji, gdy scenariusz danego przypadku użycia przewiduje wiele wywołań innych przypadków w różnych miejscach.



Rysunek 47 – Środowisko translacyjne – struktura warstwy prezentera

Klasa `AbstractUseCasePresenter` implementuje ponadto trzy metody związane z kontrolą liczby aktywnych ekranów wyświetlanych użytkownikowi w trakcie wykonania

danego przypadku użycia. Metody `pageOpened()` i `pageClosed()` wywoływane są lokalnie przez kod prezentera odpowiednio po każdym otwarciu i zamknięciu ekranu w celu inkrementacji bądź dekrementacji licznika otwartych ekranów `numberOfOpenedPages`. Licznik ten jest wykorzystywany podczas finalizowania przypadku użycia (metoda `finalizeUseCase()`). Jeżeli po wykonaniu całej logiki przypadku użycia przewidzianej w scenariuszu pozostają jakieś aktywne okna związane z danym przypadkiem użycia, to, pomimo osiągnięcia (bądź nie) celu przypadku użycia (atrybut `useCaseResult`), sterowanie nie zostanie zwrócone do prezentera przypadku wywołującego, dopóki okna te nie zostaną zamknięte przez użytkownika. Przykładowo, jest tak w sytuacji, w której ostatnio wyświetlony ekran aplikacji prezentuje dane, z którymi użytkownika chce się zapoznać zanim zakończy przypadek użycia i wróci do miejsca wywołania lub gdy z poziomu takiego ekranu użytkownik ma możliwość wywoływania innych, opcjonalnych przypadków użycia. Kod metody `finalizeUseCase()` przedstawia Rysunek 48a.

Trzecią wspomnianą metodą kontrolującą liczbą aktywnych okien jest `closeCurrentPageAndFinalizeUseCase()`. Jest ona wywoływana przez warstwę widoku zawsze wtedy, gdy użytkownik chce ręcznie zamknąć ekran aplikacji, niezależnie od tego czy końcowy rezultat przypadku użycia został osiągnięty czy nie. W takiej sytuacji, prezenter zleca widokowi zamknięcie aktywnego okna a następnie wywołuje metodę `finalizeUseCase()`, która może zakończyć przypadek użycia jeśli nie ma więcej aktywnych okien. Kod metody `closeCurrentPageAndFinalizeUseCase()` przedstawia Rysunek 48b.

a)
```
protected void finalizeUseCase() {
    if (numberOfOpenedPages == 0)
        if (invokingPresenter != null)
            invokingPresenter.resumeUseCase(useCaseResult);
}
```

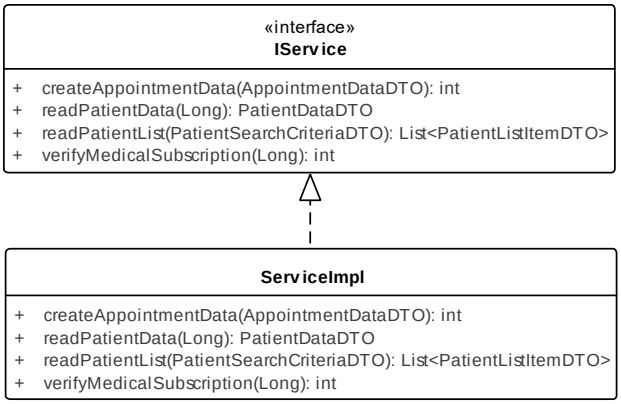
b)
```
public void closeCurrentPageAndFinalizeUseCase() {
    if (numberOfOpenedPages > 0) {
        view.closeCurrentPage();
        pageClosed();
    }
    finalizeUseCase();
}
```

Rysunek 48 – Kod metod finalizeUseCase() (a) oraz closeCurrentPageAndFinalizeUseCase() (b)

Oprócz powyższej logiki, każda klasa prezentera udostępnia operacje obsługujące zdarzenia przechwycone przez klasy widoku (patrz: metoda `actionPerformed()`) i przekazane do prezentera. Operacje te implementują logikę wykonywaną w odpowiedzi na akcje użytkownika (np. kliknięcie przycisku lub opcji w menu), stąd ich nazwy odzwierciedlają te akcje, np. `findPatientTriggered()` czy `invokeScheduleAppointment()`. Operacje te mogą posiadać parametry wejściowe w celu przekazania prezenterowi danych stanowiących kontekst wykonanej przez użytkownika akcji. Przykładami takich danych są np. kryteria wyszukiwania pacjentów lub identyfikator pacjenta wybranego z listy pacjentów. Dane te są tymczasowo przechowywane w atrybutach klasy prezentera i – zgodnie z logiką przypadku użycia – mogą być przekazane do warstwy modelu w celu wykonania określonej operacji dziedzinowej lub mogą służyć jako parametry wywołania innego przypadku użycia.

Jedyną klasą prezentera, która nie jest specjalizacją prezentera abstrakcyjnego, jest `MainPresenter`. Nie implementuje ona logiki żadnego przypadku użycia a jego rolą jest wywoływanie prezenterów dla tych przypadków użycia, które mogą być bezpośrednio uruchomione przez użytkownika z poziomu głównego ekranu lub głównego menu aplikacji (klasa `MainPage` w warstwie widoku).

Aby umożliwić wywołanie logiki prezentera z poziomu innego prezentera, prezenter wywołujący posiada atrybut wskazujący na obiekt prezentera wywoływanego. Na Rysunek 47 zostało to zilustrowane za pomocą asocjacji skierowanych, które odzwierciedlają relacje „invoke” pomiędzy przypadkami użycia.



Rysunek 49 – Środowisko translacyjne – struktura warstwy modelu

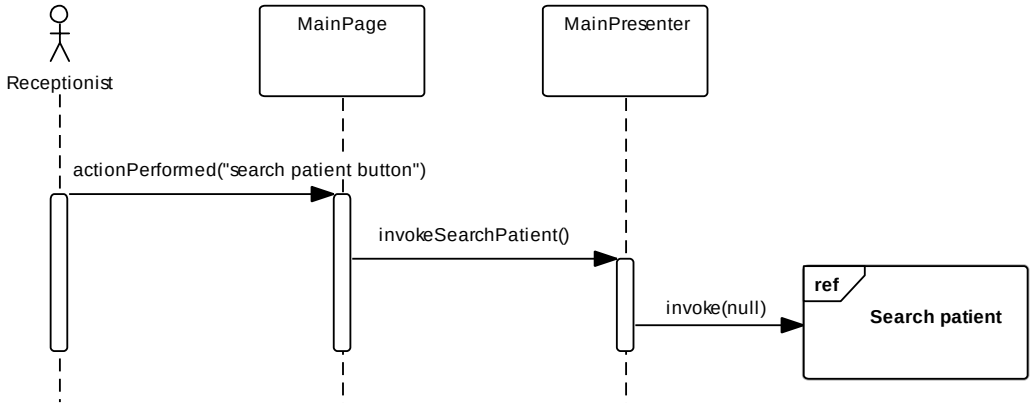
Na warstwę modelu składa się interfejs `ISvervice` oraz jego implementacja `ServiceImpl` (Rysunek 49). Interfejs deklaruje zestaw wszystkich operacji dziedzinowych niezbędnych do

realizacji przypadków użycia. Są to zarówno typowe operacje na danych dziedzinowych typu CRUD[27] [162] jak też operacje wykonujące bardziej zaawansowaną logikę biznesową jak np. walidacja danych, obliczenia na danych czy porównywanie danych. W przypadku przedstawionego na diagramie interfejsu `IService`, operacjami typu CRUD są `createAppointmentData()`, `readPatientData()` oraz `readPatientList()`, natomiast `verifyMedicalSubscription()` wymaga określenia konkretnych reguł biznesowych. Operacje te mogą posiadać parametry wejściowe w postaci obiektów DTO lub pojedynczych wartości (np. identyfikator obiektu biznesowego) w celu przekazania warstwie modelu danych niezbędnych do wykonania danej operacji. Wynikiem operacji mogą być dane dziedzinowe opakowane w obiekty DTO lub wartość określająca status wykonania operacji, jeżeli wymaga tego logika aplikacji. Ze względu na brak w języku RSL konstrukcji pozwalających wyrażać logikę biznesową, semantyka translacyjna nie uwzględnia reguł tworzenia kodu operacji w warstwie modelu. Reguły takie zostały opisane przez Rybiński [53] dla rozszerzonego języka RSL-DL (gdzie DL jest skrótem od Domain Logic).

Poniżej opisane zostały dynamiczne aspekty działania prezentowanego środowiska aplikacyjnego. Przedstawione diagramy sekwencji obrazują jak logika scenariuszy przypadków użycia przekłada się na komunikację pomiędzy elementami opisanej powyżej struktury MVP.

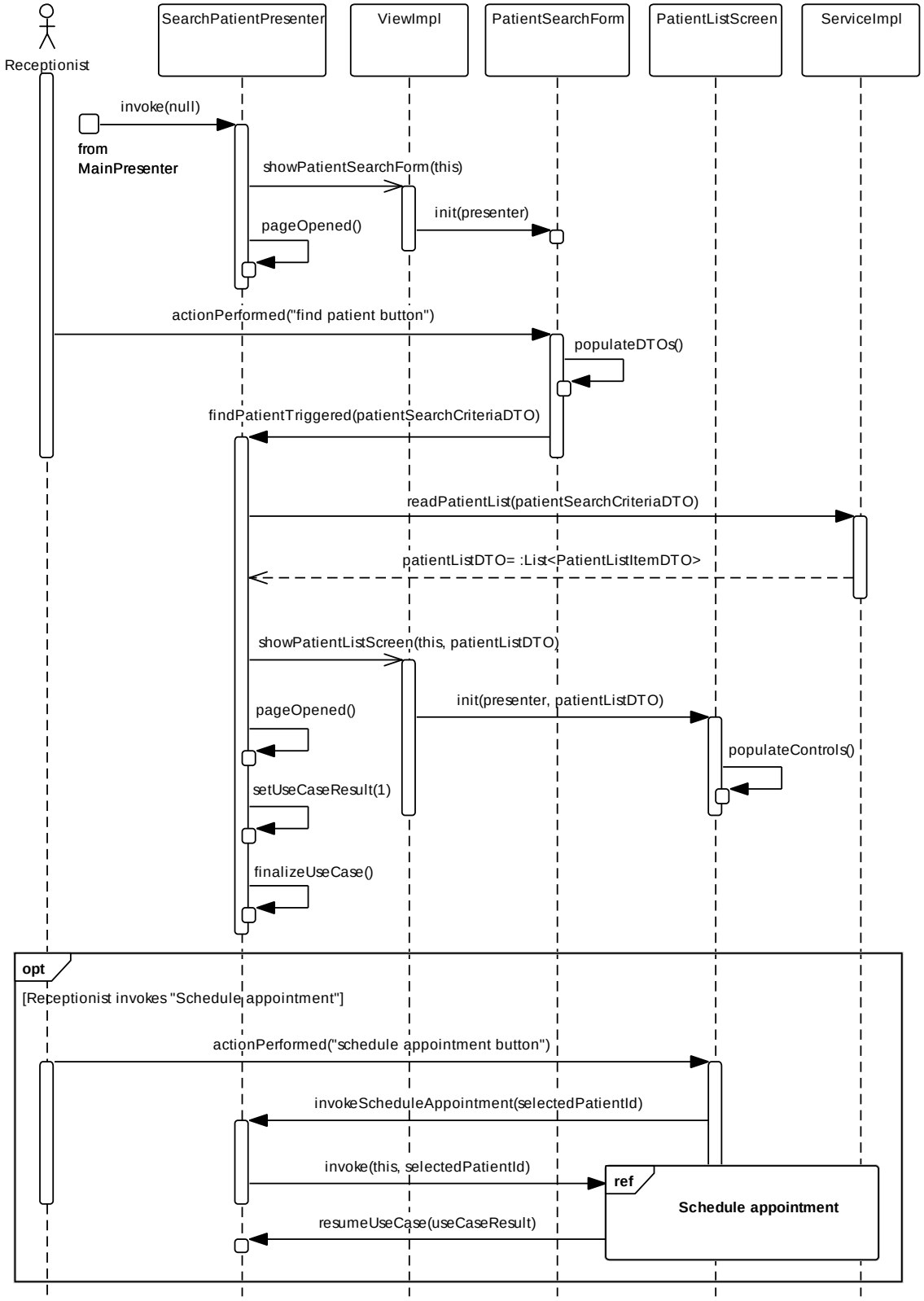
Diagram sekwencji na Rysunek 50 przedstawia logikę uruchomienia przez użytkownika przypadku użycia z poziomu głównego ekranu aplikacji. Operacja obsługi zdarzenia ekranu głównego (`actionPerformed()`) przekazuje sterowanie do prezentera głównego `MainPage` poprzez wywołanie metody adekwatnej do wybranej przez użytkownika opcji. Ta, z kolei uruchamia logikę żądanego przypadku użycia poprzez wywołanie metody `invoke()` na obiekcie prezentera tego przypadku. W prezentowanym przykładzie jest to przypadek „Search patient”. Dla większej czytelności, sekwencja komunikatów realizująca logikę tego przypadku została pokazana na oddzielnym diagramie (Rysunek 51).

[27] CRUD – od ang. create, read, update and delete (pol. utwórz, odczytaj, aktualizuj i usuń) – cztery podstawowe operacje pozwalające zarządzać warstwą przechowywania danych.



Rysunek 50 – Wywołanie przypadku użycia z poziomu głównego ekranu aplikacji

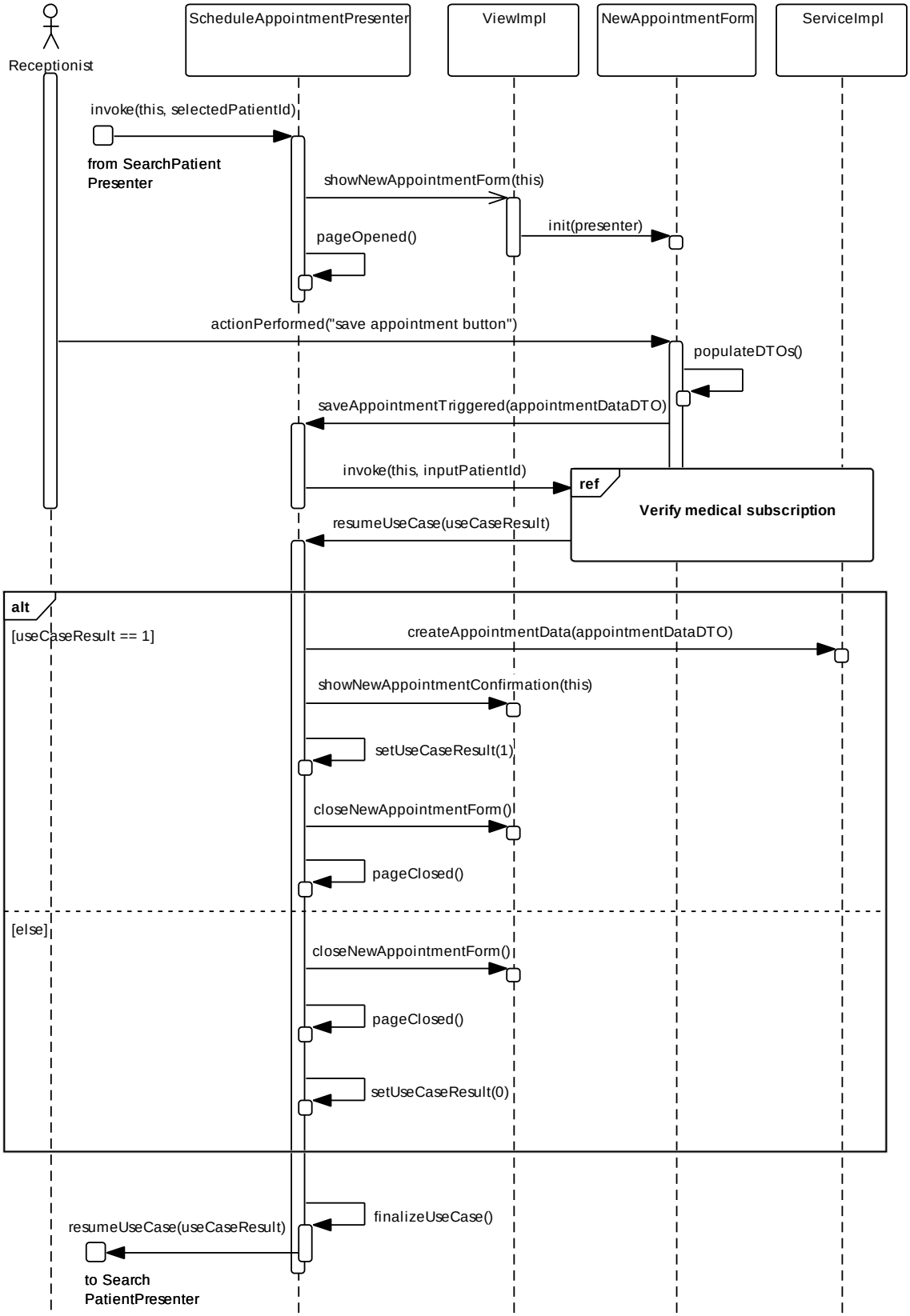
Po wywołaniu przypadku użycia „Search patient", obiekt klasy `SearchPatientPresenter` zarządza wykonaniem logiki przewidzianej dla tego przypadku. Zgodnie, ze scenariuszem, prezenter zleca widokowi wyświeltenie formularza wyszukiwania pacjenta, poprzez wywołanie metody `showPatientSearchForm()` na obiekcie `ViewImpl`. Metoda ta inicjalizuje i wyświetla wspomniany formularz (`PatientSearchForm`). Parametrem obu metod jest obiekt prezentera, który musi zostać przekazany do obiektu implementującego formularz, aby poinformować prezenter, gdy użytkownik wprowadzi kryteria wyszukwania pacjenta i kliknie przycisk „find patient". Dzieje się to w ramach wykonania metody `actionPerformed()`. Zanim, jednak, sterowanie wróci do prezentera, obiekt formularza wywołuje pomocniczą metodę `populateDTOs()`, która opakowuje wartości wprowadzone przez użytkownika w obiekt `PatientSearchCriteriaDTO`. Obiekt DTO przekazywany jest do prezentera jako parametr metody `findPatientTriggered()`. Po odzyskaniu sterowania, prezenter kontynuuje wykonywanie logiki przypadku użycia. W naszym przykładzie jest to wywołanie operacji dziedzinowej na warstwie modelu: `readPatientList()`. Operacja ta odczytuje ze źródła danych i zwraca listę pacjentów spełniających podane przez użytkownika kryteria wyszukiwania. Pobrana lista jest następnie przekazywana do warstwy widoku w celu jej zaprezentowania użytkownikowi w postaci ekranu `PatientListScreen`. Krok ten przekłada się na sekwencję komunikatów: `showPatientListScreen()`, `init()` oraz `populateControls()`. Zgodnie ze scenariuszem, wyświetlenie listy pacjentów kończy wykonanie przypadku użycia. Obiekt prezentera, wywołując własną metodę `setUseCaseResult()`, ustawia atrybut `useCaseResult` na wartość „1" oznaczającą sukces a następnie próbuje sfinalizować przypadek użycia wywołaniem metody `finalizeUseCase()`.



Rysunek 51 – Logika aplikacji dla przypadku użycia „Search patient”

Należy zauważyć, że pomimo pomyślnego osiągnięcia celu przypadku użycia, ekran `PatientListScreen` pozostaje widoczny dla użytkownika, który, zgodnie ze scenariuszem, ma możliwość opcjonalnego wywołania dodatkowego przypadku użycia (np. „Schedule appointment") dla pacjenta wybranego z prezentowanej listy. Oznacza to, że ekran listy pacjentów musi udostępniać odpowiednią kontrolkę wyzwalającą dodatkowy przypadek użycia (np. przycisk „search patient") oraz musi obsługiwać takie zdarzenie w ramach metody `actionPerformed()`. Obsługa zdarzenia po stronie widoku sprowadza się do poinformowania prezentera o akcji użytkownika poprzez wywołanie udostępnianej przez prezenter w tym celu metody `invokeScheduleAppointment()`. Ta, z kolei, przekazuje sterowanie do prezentera przypadku wywoływanego za pomocą komunikatu `invoke()`. Po zakończeniu wykonywania wywołanego przypadku użycia, sterowanie wraca do prezentera wywołującego za sprawą metody `resumeUseCase()`. Jej parametrem jest rezultat wywoływanego przypadku użycia (`useCaseResult`). W niektórych sytuacjach, rezultat ten może mieć znaczenie dla dalszego przebiegu przypadku wywołującego. W naszym przykładzie scenariusz nie definiuje jednak żadnych dalszych kroków. Sekwencja komunikatów dla wywołania przypadku użycia „Schedule appointment" jest przedstawiona na omawianym diagramie w postaci fragmentu włączonego „opt", natomiast sekwencja komunikatów realizujących logikę tego przypadku jest przedstawiona na osobnym diagramie na Rysunek 52. Widać na nim, że sekwencje komunikatów realizujących typowe kroki scenariusza (wyświetlenie ekranu, wyzwolenie akcji przez użytkownika czy wykonanie operacji dziedzinowej) są podobne jak w powyższym przykładzie. Należy jednak wyjaśnić różnice w wywołaniach przypadków użycia.

Po pierwsze, prezenter `ScheduleAppointmentPresenter` w komunikacie `invoke()` otrzymuje dwa parametry. Jeden z nich to obiekt prezentera wywołującego. Dzięki temu, po zakończeniu przypadku użycia, prezenter `ScheduleAppointmentPresenter` może zwrócić sterowanie wraz z rezultatem końcowym do miejsca wywołania, uruchamiając metodę `resumeUseCase()` na przekazanym obiekcie prezentera wywołującego (patrz: ostatni komunikat na przedstawionym diagramie sekwencji). Drugi parametr to identyfikator pacjenta, dla którego ma być umówiona wizyta. Przekazaną wartością inicjalizowany jest atrybut `inputPatientId`. Jest on wymagany do uruchomienia przypadku użycia, co określa warunek wstępny scenariusza wyrażony w zdaniu „precondition".

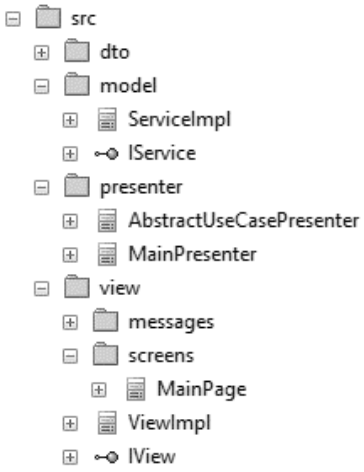


Rysunek 52 – Logika aplikacji dla przypadku użycia „Schedule appointment”

Po drugie, prezentowany przykład pokazuje mechanizm bezwarunkowego wywołania innego przypadku użycia, gdzie wywołanie nie jest efektem akcji użytkownika lecz dzieje się zawsze w określonym miejscu scenariusza. W naszym przykładzie, przypadek użycia „Verify medical subscription" jest wywoływany zawsze przed zapisaniem w systemie wizyty pacjenta, aby zweryfikować ważność abonamentu. Wywołanie tego przypadku jest częścią logiki wykonywanej przez prezenter w ramach metody `saveAppointmentTriggered()`, uruchamianej po tym jak użytkownik wprowadzi dane wizyty i wyzwoli akcję „save appointment". Po wykonaniu wywołanego przypadku i przekazaniu sterowania z powrotem do prezentera `ScheduleAppointmentPresenter`, dalszy jego przebieg zależy od zwróconego rezultatu (zmienna `useCaseResult`). Scenariusz, za pomocą zdania warunkowego, definiuje dwie alternatywne ścieżki, które zostały pokazane na diagramie w postaci fragment włączonego „alt". W przypadku, gdy wywoływany przypadek zakończył się sukcesem (`useCaseResult == 1`), informacje o wizycie są zapisywane przez warstwę modelu (`createAppointmentData()`), wyświetlany jest komunikat potwierdzający utworzenie wizyty (`showNewAppointmentConfirmation()`) po czym zamykany jest ekranu wizyty (`closeNewAppointmentForm()`) a cały przypadek użycia kończy się powodzeniem. W przeciwnym razie prezenter jedynie zleca widokowi zamknięcie ekranu wizyty a wykonanie przypadku kończy się porażką.

Dalsza część rozdziału przedstawia zestaw reguł translacyjnych określających zasady mapowania konkretnych konstrukcji języka RSL na modele implementacyjne wyrażone w języku UML i Java, zgodne z przedstawionym środowiskiem translacyjnym.

Przedstawione reguły zakładają, że istnieje wstępna struktura pakietów kodu dla środowiska translacyjnego a także podstawowe interfejsy i klasy zawierające kod (kompletny lub częściowy) implementujący ogólne mechanizmy, które nie zależą od źródłowych konstrukcji w języku RSL, np. kompletna klasa `AbstractUseCasePresenter` lub kalsa `ViewImpl` zawierająca jedynie podstawowe, niezależne od modelu wymagań, atrybuty i metody. Struktura taka przedstawiona jest na Rysunek 53.



Rysunek 53 – Wstępna struktura kodu dla uogólnionego środowiska translacyjnego

Pakiety kodu w przedstawionej strukturze grupują klasy tworzące poszczególne warstwy zdefiniowane przez wzorzec MVP. Dodatkowo mogą one zawierać dowolną strukturę podpakietów, np. pakiet `view` zawiera podpakiety `screens` oraz `messages`, które grupują klasy implementujące ekrany oraz komunikaty. Dalszy podział na podpakiety może wynikać z modelu wymagań lub może zależeć od konkretnego docelowego środowiska translacyjnego. W przypadku konkretnego środowiska należy dodatkowo założyć, że będą istniały wszystkie specyficzne dla danego środowiska klasy, interfejsy i pakiety, z których dziedziczą, realizują lub od których zależą klasy tworzone na podstawie przedstawionych reguł translacyjnych. Przykładowo, w środowisku Echo3, każda klasa implementująca ekran aplikacji musi dziedziczyć z klasy `ContentPane` zamiast z wirtualnej klasy `Screen`, jak w przypadku rozpatrywanego środowiska abstrakcyjnego.

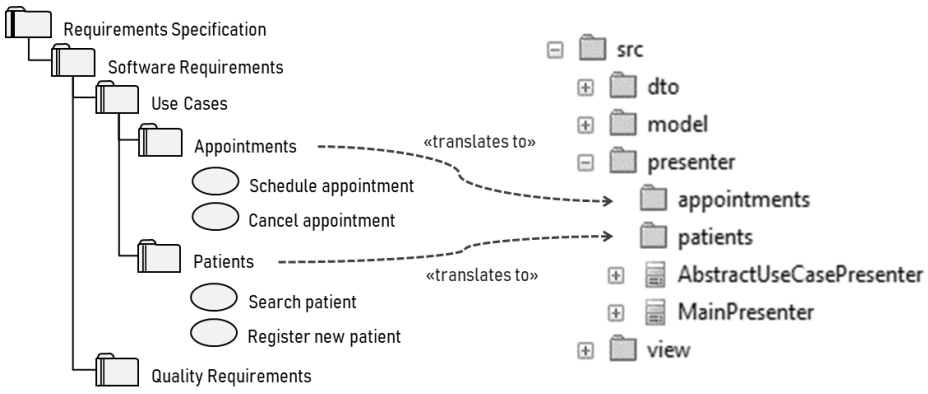
Wszystkie reguły składające się na definicję semantyki translacyjnej przedstawione są w kolejnych podrozdziałach. Reguły podzielone są na cztery części. Część pierwsza (rozdział 5.2) wprowadza reguły translacji podstawowych konstruktów języka RSL na ogólną strukturę środowiska docelowego. Część druga (rozdział 5.3) obejmuje reguły pozwalające wygenerować operacje dziedzinowe w warstwie modelu. Reguły przedstawione w części trzeciej (rozdział 5.4) opisują sposób translacji zdań scenariuszy przypadków użycia na kompletny kod klas prezenterów. Część czwarta (rozdział 5.5) zawiera reguły translacji elementów dziedzinowych reprezentujących interfejs użytkownika. Definicje reguł są zilustrowane konkretnymi przykładami.

5.2 Reguły translacji do ogólnej struktury kodu

Pierwsza część definicji semantyki translacyjnej przedstawia dziewięć reguł translacji podstawowych elementów języka RSL (m.in.: pakiety, przypadki użycia, pojęcia dziedzinowe) na inicjalną strukturę MVP środowiska docelowego. Uwzględnia ona pakiety kodu, podstawowy kod klas prezenterów, ekranów oraz komunikatów a także kompletny kod klas obiektów transferu danych.

Reguła G1. *Dla struktury pakietów wymagań wewnątrz pakietu „Use cases" utwórz analogiczną strukturę pakietów klas wewnątrz pakietu* `presenters` *z zachowaniem hierarchii pakietów źródłowych. Nazwa pakietu klas odpowiada nazwie pakietu wymagań zapisanej w notacji Camel case*[28].

Celem tej prostej reguły jest odzwierciedlenie pakietów wymagań grupujących przypadki użycia w strukturze pakietów klas warstwy prezentera tak, aby klasy reprezentujące poszczególne przypadki użycia były pogrupowane zgodnie z zamysłem twórcy specyfikacji wymagań. Ma to znaczenie głównie w przypadku modeli wymagań definiujących dużą liczbę przypadków użycia – zwiększa czytelność i łatwość poruszania się po złożonym modelu. Reguła ta zilustrowana jest na Rysunek 54.

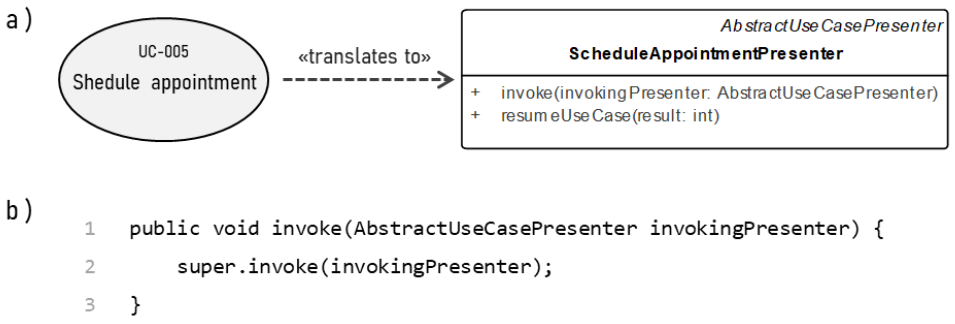


Rysunek 54 – Reguła G1: translacja pakietów przypadków użycia na pakiety klas

[28] Camel case – system notacji ciągów tekstowych, w którym kolejne wyrazy pisane są łącznie, rozpoczynając każdy z nich (oprócz pierwszego) wielką literą, np. *useCaseResult*. Notacja ta jest powszechnie przyjętą w wielu językach programowania (m.in. Java), gdzie służy do zapisu nazw zmiennych, atrybutów, metod, pakietów, itp.

Reguła G2. *Dla każdego przypadku użycia utwórz klasę prezentera i umieść ją w pakiecie klas odpowiadającym pakietowi przypadku użycia (patrz: Reguła G1). Nazwa klasy jest nazwą przypadku użycia zapisaną w notacji Pascal case*[29] *z sufiksem „Presenter". Klasa prezentera rozszerza klasę* `AbstractUseCasePresenter` *i nadpisuje dwie metody swojej nadklasy:* `invoke(AbstractUseCasePresenter invokingPresenter)` *oraz* `resumeUseCase(int useCaseResult)`. *Kod pierwszej metody zawiera wywołanie metody* `invoke()` *z nadklasy.*

Reguła ta zilustrowana jest na Rysunek 55. Każdemu przypadkowi użycia odpowiada jedna klasa w warstwie prezentera, której zadaniem jest implementacja logiki aplikacji zdefiniowanej w ramach tego przypadku użycia. Wszystkie klasy w warstwie prezentera realizują generyczny mechanizm wywoływania logiki innych przypadków użycia (odpowiadający relacji „invoke"), dziedziczony z abstrakcyjnej klasy `AbstractUseCasePresenter`, będącej częścią docelowego środowiska translacyjnego (patrz: rozdział 5.1). Niniejsza reguła odnosi się jedynie do metod dziedziczonych z nadklasy. Dalszy kod wynikać będzie z innych reguł.



Rysunek 55 – Reguła G2: translacja przypadku użycia na klasę prezentera (a); kod metody invoke() (b)

Reguła G3. *Dla każdego pojęcia dziedzinowego typu Screen utwórz klasę implementującą ekran aplikacji i umieść ją w pakiecie* `src/view/screens`. *Nazwa klasy jest nazwą pojęcia źródłowego zapisaną w notacji Pascal case. Klasa ekranu rozszerza klasę* `Screen`, *definiującą podstawowe właściwości i zachowanie ekranów aplikacji. Klasa posiada dwie*

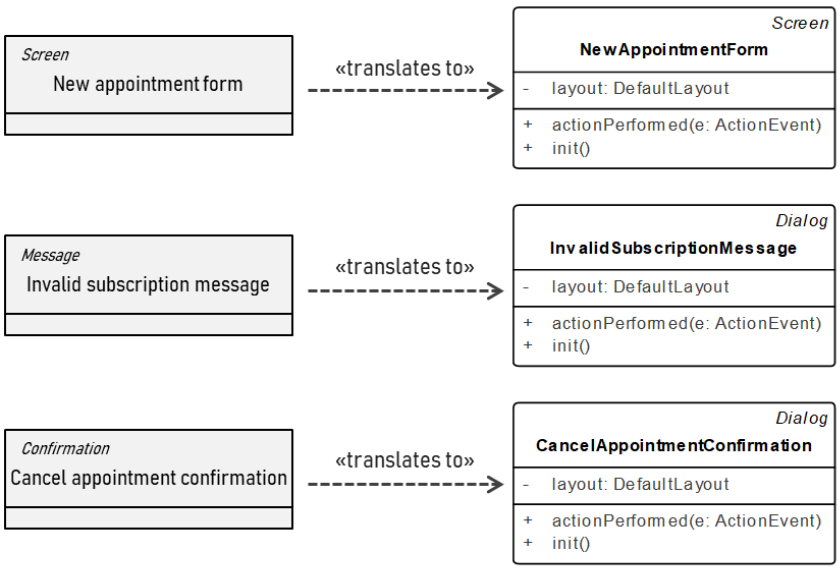
[29] Pascal case (inna powszechna nazwa: Upper camel case) – system notacji ciągów tekstowych, w którym kolejne wyrazy pisane są łącznie, rozpoczynając każdy z nich wielką literą, np. *MainPresenter*. Notacja ta jest powszechnie przyjęta w wielu językach programowania (m.in. Java), gdzie służy do zapisu m.in. nazw klas czy konstruktorów.

publiczne metody: `actionPerformed(ActionEvent e)` *oraz* `init()` *(bez parametrów). Klasa zawiera atrybuty definiujące układ pól i kontrolek na ekranie (ang. layout) oraz kod inicjalizujący te atrybuty w metodzie* `init()`.

Reguła G4. *Dla każdego pojęcia dziedzinowego typu Message lub Confirmation utwórz klasę implementującą komunikat i umieść ją w pakiecie* `src/view/messages`. *Nazwa klasy jest nazwą pojęcia źródłowego zapisaną w notacji Pascal case. Klasa komunikatu rozszerza klasę* `Dialog`, *definiującą podstawowe właściwości i zachowanie ekranów aplikacji. Klasa posiada dwie publiczne metody:* `actionPerformed(ActionEvent e)` *oraz* `init()` *(bez parametrów). Klasa zawiera atrybuty definiujące układ pól i kontrolek na ekranie (ang. layout) oraz kod inicjalizujący te atrybuty w metodzie* `init()`.

Reguły G3 i G4 zilustrowane są na Rysunek 56. Każdemu pojęciu typu Screen, Message i Confirmation odpowiadają klasy w warstwie widoku, które dziedziczą ogólne właściwości i zachowanie elementów interfejsu użytkownika w danym środowisku programistycznym. Klasy te muszą zatem implementować wymagane motody z nadklasy: `actionPerformed()` – do obsługi zdarzeń oraz `init()` – inicjującą i wyświetlającą ekran lub komunikat. Podobnie, jak w przypadku reguły G2, dalszy kod klasy wynika z innych konstrukcji modelu wymagań, co będzie uwzględnione w innych regułach. Należy tu podkreślić, że kod tych klas jest całkowicie zależny od konkretnej platformy, co należy uwzględnić podczas implementacji tych reguł.

a)



b)

```
1  public void init() {
2      // Set layout data
3      layout = new DefaultLayout();
4      layout.setInsets(10);
5      // ...
6  }
```

Rysunek 56 – Reguła G3 i G4: translacja pojęć typu Screen, Message i Confirmation na klasy widoku (a); przykładowy kod metody init() (b)

Reguła G5. *Dla każdego przypadku użycia powiązanego z aktorem relacją typu „use" utwórz w klasie* `MainPresenter`:

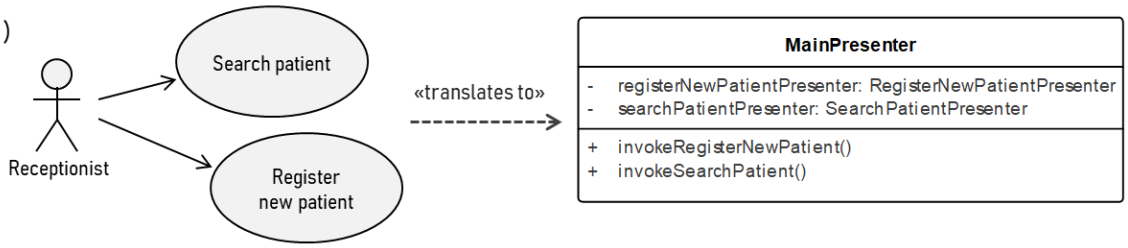
a) prywatny atrybut, którego typem jest klasa prezentera odpowiadająca przypadkowi użycia a nazwa jest nazwą klasy prezentera zapisaną w notacji Camel case;

b) publiczną metodę o nazwie odpowiadającej nazwie przypadku użycia zapisanej w notacji Pascal case z prefiksem „invoke", której kod stanowi wywołanie metody `invoke(null)` *obiektu wskazywanego przez powyższy atrybut.*

Reguła G5 zilustrowana jest na Rysunek 57. Jak wspomniano w poprzednim podrozdziale, z poziomu ekranu głównego użytkownik ma możliwość bezpośredniego uruchomienia przypadków użycia, które w modelu RSL są wskazywane przez relacje „use" (patrz: diagram sekwencji na Rysunek 50). Klasa `MainPresenter` musi, więc, posiadać metody których zadaniem jest uruchomienie na żądanie użytkownika logiki prezenterów odpowiadających tym przypadkom użycia. Metody te są wywoływane z poziomu kodu klasy `MainPage` (patrz: reguła

G6). Ponieważ `MainPresenter` nie implementuje logiki, która zależałaby od rezultatu wykonania wywołanego przypadku użycia, nie ma potrzeby aby wywoływany prezenter zwracał rezultat poprzez zwrotne wywołanie metody `resumeUseCase()`. Stąd, jako parametr metody `invoke()` przekazywana jest wartość `null`, zamiast obiektu prezentera wywołującego (`invokingUseCase`).

a)



b)

```
1  public void invokeSearchPatient() {
2      searchPatientPresenter = new SearchPatientPresenter();
3      searchPatientPresenter.invoke(null);
4  }
5
6  public void invokeRegisterNewPatient() {
7      registerNewPatientPresenter = new SearchPatientPresenter();
8      newPatientPresenter.invoke(null);
9  }
```

Rysunek 57 – Reguła G5: translacja przypadków użycia powiązanych z aktorem na klasę MainPresenter (a); kod tworzonych metod (b)

Reguła G6. *Dla każdego przypadku użycia powiązanego z aktorem relacją typu „use" utwórz w klasie* `MainPage`:

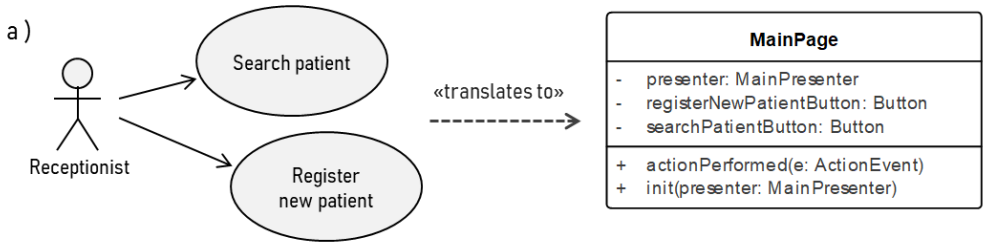
a) *prywatny atrybut reprezentujący kontrolkę ekranową (przycisk, opcja menu, itp.) wyzwalającą przypadek użycia; nazwa atrybutu jest nazwą przypadku użycia zapisaną w notacji Camel case z sufiksem odpowiadającym typowi kontrolki („Button", „MenuOption", itp.);*

b) *kod w metodzie* `init()` *inicjujący kontrolkę w celu jej prezentacji na ekranie głównym; nazwa kontrolki widoczna na ekranie odpowiada nazwie przypadku użycia;*

c) *kod w metodzie* `actionPerformed()` *przechwytujący zdarzenie użycia kontrolki przez użytkownika i wywołujący na obiekcie klasy* `MainPresenter` *metodę uruchamiającą logikę danego przypadku użycia określoną w regule G5.*

Powyższa reguła zilustrowana jest na Rysunek 60Rysunek 58. W momencie, kiedy użytkownik uruchamia z poziomu głównego ekranu aplikacji przypadek użycia poprzez wybór odpowiedniego wyzwalacza, wykonywany jest kod obsługi tego zdarzenia polegający na wywołaniu odpowiedniej metody głównego prezentera (`MainPresenter`), który z kolei uruchamia właściwy przypadek użycia (patrz: reguła G5). Zastosowanie reguły G6 dla całego modelu przypadków użycia, pozwala wygenerować kompletne menu główne aplikacji, dostępne na ekranie startowym.

a)



b)

```
1  public void init() {
2
3      // Initialize searchPatientButton
4      searchPatientButton = new Button(”Search patient”);
5      searchPatientButton.setActionCommand(”searchPatientButton”);
6      content.add(searchPatientButton);
7
8      // Initialize registerNewPatientButton
9      registerNewPatientButton = new Button(”Register new patient”);
10     // ...
11 }
12
13 public void actionPerformed(ActionEvent e) {
14
15     if (e.getActionCommand().equals(”searchPatientButton”))
16         presenter.invokeSearchPatient();
17
18     if (e.getActionCommand().equals(”registerNewPatientButton”))
19         presenter.invokeRegisterNewPatient();
20 }
```

Rysunek 58 – Reguła G6: translacja przypadków użycia powiązanych z aktorem na klasę MainPage (a); kod metod (b)

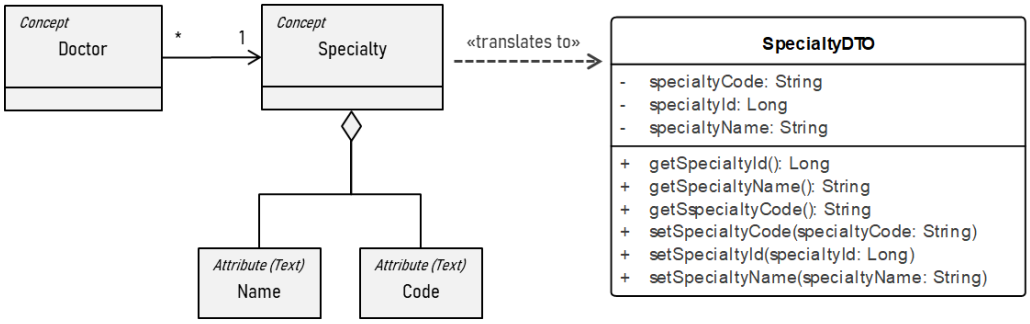
Reguła G7. *Dla każdego pojęcia dziedzinowego typu Concept utwórz klasę DTO w pakiecie* `src/dto`. *Nazwa klasy jest nazwą pojęcia źródłowego zapisaną w notacji Pascal case z sufiksem „DTO”. Dla każdego atrybutu pojęcia utwórz prywatny atrybut w klasie DTO. Nazwa atrybutu klasy jest konkatenacją nazwy pojęcia oraz nazwy jego atrybutu*

zapisanych w notacji Camel case. Typ atrybutu klasy jest tworzny na podstawie typu atrybutu pojęcia według następujących reguł mapowania:

- *Text, Description, Secret text → String,*
- *Number → Integer,*
- *Floating point number → Float,*
- *True or false → Boolean,*
- *Date, Time → java.util.Date.*

Dodatkowo, utwórz prywatny atrybut typu Long o nazwie składającej się z nazwy pojęcia zapisanej w notacji Camel case z sufiksem „Id".

Dla każdego atrybutu klasy DTO utwórz parę publicznych metod służących do pobierania i modyfikowania wartości atrybutu: akcesor i mutator. Nazwy tych metod odpowiadają nazwie atrybutu zapisanej w notacji PascalCase z prefiksem „get" – w przypadku akcesora oraz „set" – w przypadku mutatora.



Rysunek 59 – Reguła G7: translacja pojęcia typu Concept na klasę transferu danych

Reguła G7 zilustrowana jest na Rysunek 59. Klasy transferu danych odpowiadające pojęciom typu Concept używane są w celu przekazywania danych o obiektach biznesowych określonego typu, m.in. na potrzeby słownikowych list wyboru prezentowanych użytkownikom w interfejsie użytkownika.

Reguła G8. *Dla każdego pojęcia dziedzinowego typu Simple View utwórz klasę DTO i umieść ją w pakiecie* `src/dto`. *Nazwa klasy jest nazwą pojęcia źródłowego zapisaną w notacji Pascal case z sufiksem „DTO". Dla każdego atrybutu wskazywanego przez pojęcie Simple View utwórz prywatny atrybut w klasie DTO. Nazwa atrybutu klasy jest konkatenacją nazwy pojęcia typu Concept, do którego należy atrybut oraz nazwy tego atrybutu*

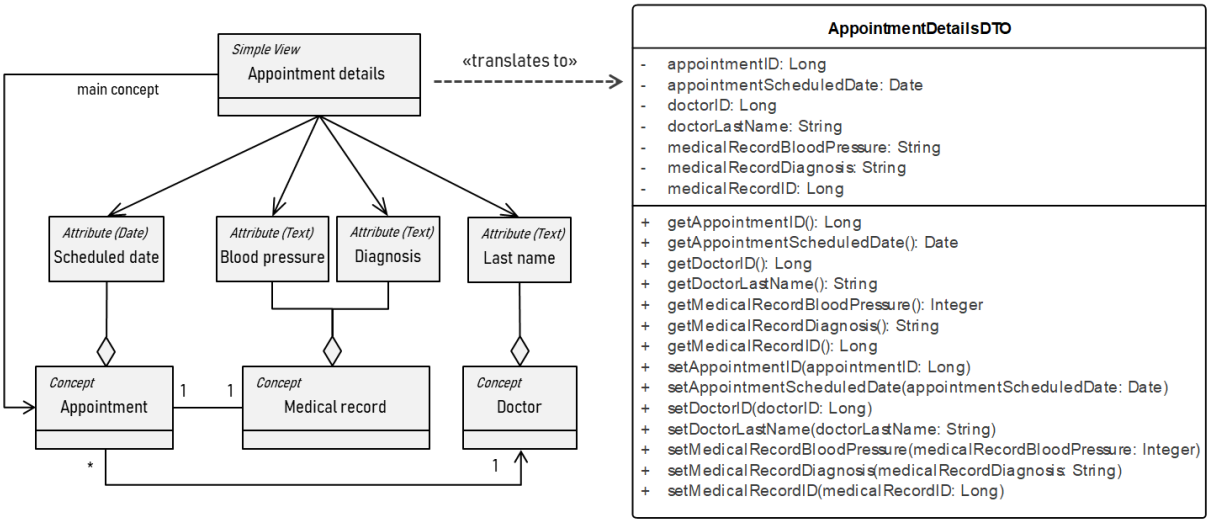
zapisanych w notacji Camel case. Typ atrybutu klasy jest tworzony na podstawie typu atrybutu pojęcia według następujących reguł:

a) *Jeżeli atrybut należy do pojęcia „main concept” – użyj reguł mapowania przedstawionych w regule G7;*
b) *Jeżeli atrybut należy do pojęcia typu Concept powiązanego z pojęciem „main concept” a krotność po stronie tego pierwszego jest równa 1 – użyj reguł mapowania przedstawionych w regule G7;*
c) *Jeżeli atrybut należy do pojęcia typu Concept powiązanego z pojęciem „main concept” a krotność po stronie tego pierwszego jest większa niż 1 – atrybut klasy ma typ java.util.List<T>, gdzie parametr typu listy T jest określany według reguł mapowania przedstawionych w regule G7.*

Dodatkowo, dla każdego pojęcia typu Concept, którego atrybut jest wskazywany przez pojęcie SimpleView, utwórz prywatny atrybut typu Long lub List<Long> o nazwie składającej się z nazwy pojęcia, do którego należy atrybut, zapisanej w notacji Camel case z sufiksem „Id”.

Dla każdego atrybutu klasy DTO utwórz parę publicznych metod służących do pobierania i modyfikowania wartości atrybutu: akcesor i mutator. Nazwy tych metod odpowiadają nazwie atrybutu zapisanej w notacji Pascal case z prefiksem „get” – w przypadku akcesora oraz „set” – w przypadku mutatora.

Reguła G8 zilustrowana jest na Rysunek 60. Obiekty klas DTO odpowiadających pojęciom Simple View służą do przesyłania danych do i z warstwy widoku. Zakres danych przesyłanych w obiektach klasy DTO jest ograniczony tylko do danych niezbędnych w przypadku określonego ekranu aplikacji. Obiekt taki przechowuje wybrane atrybuty pojedynczej instancji obiektu biznesowego (odpowiadającemu pojęciu „main concept”) oraz, ewentualnie, jednego lub wielu instancji obiektów z nim powiązanych, w zależności od krotności tego powiązania. W przedstawionym przykładzie, pojęciem „main concept” jest Appointment, który jest powiązany z pojęciami Medical record oraz Doctor z kotnością 1, stąd atrybuty klasy AppointmentDetailsDTO reprezentują pojedyncze wartości. Takie proste podejście do konstrukcji obiektów transferu danych jest wystarczające na potrzeby niniejszej pracy, jednak w przypadku zastosowań bardziej praktycznych niezbędne mogłoby się okazać roszerzenie języka RSL i reguł translacji.



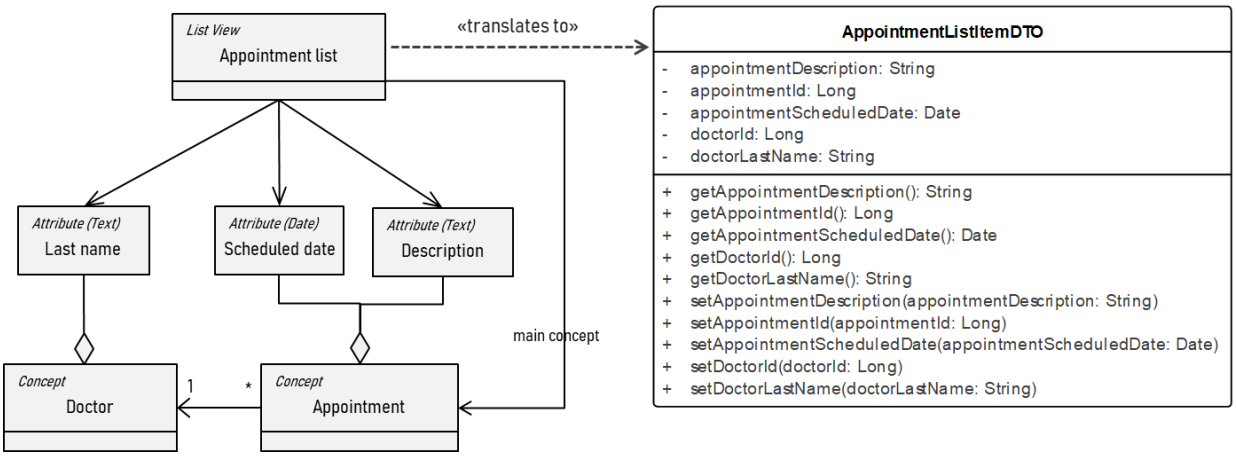
Rysunek 60 – Reguła G8: translacja pojęcia typu Simple View na klasę transferu danych

Reguła G9. *Dla każdego pojęcia dziedzinowego typu List View utwórz klasę DTO i umieść ją w pakiecie* `src/dto`. *Nazwa klasy jest nazwą pojęcia źródłowego zapisaną w notacji Pascal case z sufiksem „ItemDTO”. Dla każdego atrybutu wskazywanego przez pojęcie List View utwórz prywatny atrybut w klasie DTO. Nazwa atrybutu klasy jest konkatenacją nazwy pojęcia typu Concept, do którego należy atrybut oraz nazwy tego atrybutu zapisanych w notacji Camel case. Typ atrybutu klasy jest tworzony na podstawie typu atrybutu pojęcia według reguł mapowania przedstawionych w regule G8.*

Dodatkowo, dla każdego pojęcia typu Concept, którego atrybut jest wskazywany przez pojęcie ListView, utwórz prywatny atrybut typu Long o nazwie składającej się z nazwy pojęcia, do którego należy atrybut, zapisanej w notacji Camel case z sufiksem „Id”.

Dla każdego atrybutu klasy DTO utwórz parę publicznych metod służących do pobierania i modyfikowania wartości atrybutu: akcesor i mutator. Nazwy tych metod odpowiadają nazwie atrybutu zapisanej w notacji Pascal case z prefiksem „get” – w przypadku akcesora oraz „set” – w przypadku mutatora.

Reguła G9 zilustrowana jest na Rysunek 61. Jest ona bardzo podobna do reguły G8, z tą różnicą, że obiekty klas DTO odpowiadające pojęciom List View wykorzystywane są jako elementy list. W tej postaci przesyłane są dane wielu instancji obiektów biznesowych odpowiadających pojęciu „main concept” oraz obiektów powiązanych.



Rysunek 61 – Reguła G9: translacja pojęcia typu List View na klasę transferu danych

5.3 Reguły translacji do warstwy modelu

Trzy reguły przedstawione w tej części, dotyczą translacji stwierdzeń dziedzinowych, do których odnoszą się zdania scenariuszy przypadków użycia, na operacje dziedzinowe udostępniane przez warstwę modelu. Ze względów wspomnianych w podrozdziale 5.1, semantyka translacyjna opisana w tej pracy nie uwzględnia kodu implementującego operacje dziedzinowe w klasie `ServiceImpl`.

Reguła M1. *Dla każdego zdania typu System-to-SimpleView, System-to-ListView oraz System-to-Concept utwórz w interfejsie* `IService` *oraz w klasie* `ServiceImpl` *operację dziedzinową odpowiadającą stwierdzeniu dziedzinowemu do którego odnosi się predykat zdania. Nazwa operacji odpowiada prostej frazie czasownikowej zawartej w stwierdzeniu dziedzinowym, przy czym, jeżeli czasownik mapuje się na operację standardową przewidzianą dla pojęcia danego typu, zamiast czasownika używana jest nazwa operacji standardowej.*

Ponadto, jeżeli czasownik mapuje się na akcję „read”, to:

a) typem zwracanym przez operację jest klasa DTO odpowiadająca dopełnieniu bliższemu (patrz: reguła G7 oraz G8); w przypadku, gdy dopełnieniem bliższym jest pojęcie List View, typem zwracanym jest `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G9);*

b) *jeżeli zdanie ma dopełnienie dalsze typu Concept, utwórz parametr wywołania operacji, którego typem jest* `Long` *a nazwa jest nazwą pojęcia typu Concept zapisaną w notacji Camel case z sufiksem „Id”;*

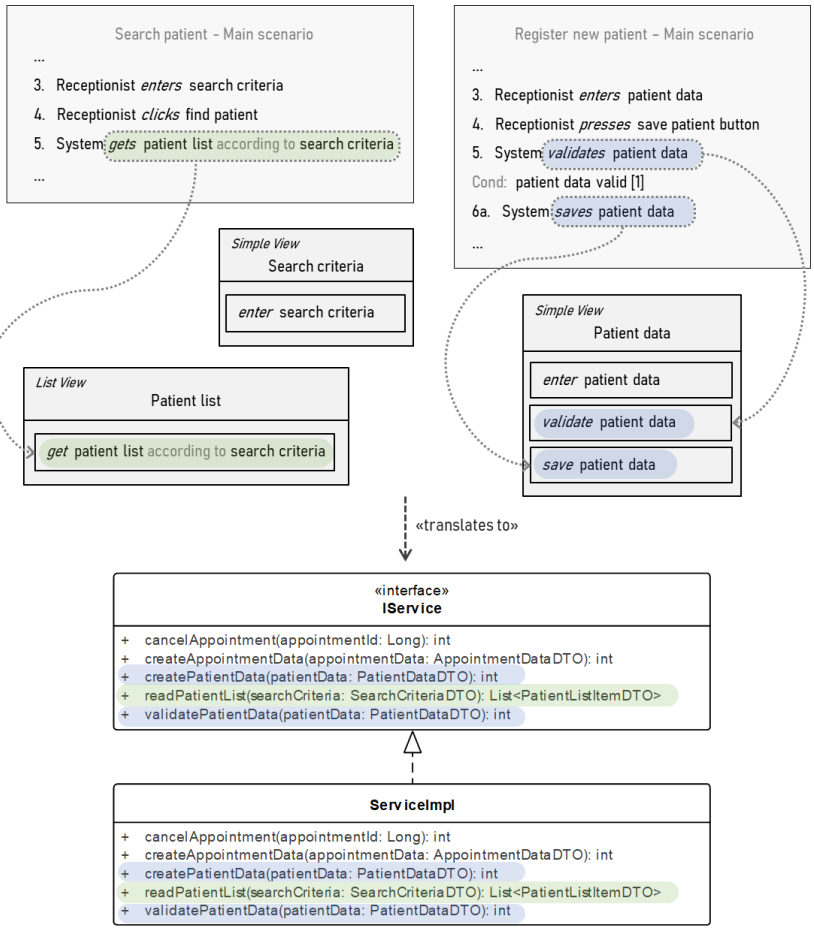
c) *jeżeli zdanie ma dopełnienie dalsze typu Simple View lub List View, utwórz parametr wywołania operacji, którego typem jest klasa DTO odpowiadająca dopełnieniu dalszemu typ Simple View (patrz: reguła G8) lub* `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca dopełnieniu dalszemu typu List View (patrz: reguła G9).*

Jeżeli czasownik mapuje się na akcję inną niż „read”, to:

a) *utwórz parametr wywołania operacji, którego typem jest klasa DTO odpowiadająca dopełnieniu bliższemu (patrz: reguła G8);*

b) *jeżeli zdanie ma dopełnienie dalsze typu Concept, utwórz parametr wywołania operacji, którego typem jest* `Long` *a nazwa jest nazwą pojęcia typu Concept zapisaną w notacji Camel case z sufiksem „Id”;*

c) *operacja zwraca wartość typu* `int`.



Rysunek 62 – Reguła M1: translacja zdań System-to-SimpleView oraz System-to-ListViewna operacje dziedzinowe

Reguła M1 zilustrowana jest na Rysunek 62. Użycie w scenariuszu zdania System-to-SimpleView, System-to-ListView lub System-to-Concept oznacza wykonanie przez system operacji dziedzinowej na danych reprezentowanych przez dopełnienie zdania (bliższe i, ewentualnie, dalsze, jeśli występuje). Operacje takie definiowane są w warstwie modelu na podstawie stwierdzeń dziedzinowych, do których odnoszą się wspomniane zdania.

Nazwa operacji zależy od czasownika użytego w stwierdzeniu dziedzinowym (orzeczenie zdania). Jeżeli czasownik mapuje się na standardową operację dziedzinową (jak przedstawiono w Tabela 1 w rozdziale 3.2.3), w nazwie operacji używana jest nazwa standardowej operacji. Prowadzi to do ujednolicenia nazw operacji dziedzinowych i zwiększa czytelność generowanego kodu. W przeciwnym razie używany jest czasownik w jego oryginalnym brzmieniu. Przykład na Rysunek 62 przedstawia trzy stwierdzenia dziedzinowe użyte w zdaniach scenariuszy, których czasowniki „get”, „validate” oraz „save” mapują się na

standardowe operacje, odpowiednio: „read”, „validate” oraz „create”, co znajduje odzwierciedlenie w nazwach operacji w interfejsie `IService` oraz klasie go implementującej.

Każda operacja może mieć parametry wejściowe oraz parametr wyjściowy. Zależą one od typu frazy czasownikowej użytej w stwierdzeniu dziedzinowym (może to być prosta lub złożona fraza czasownikowa) oraz od samego czasownika tworzącego frazę.

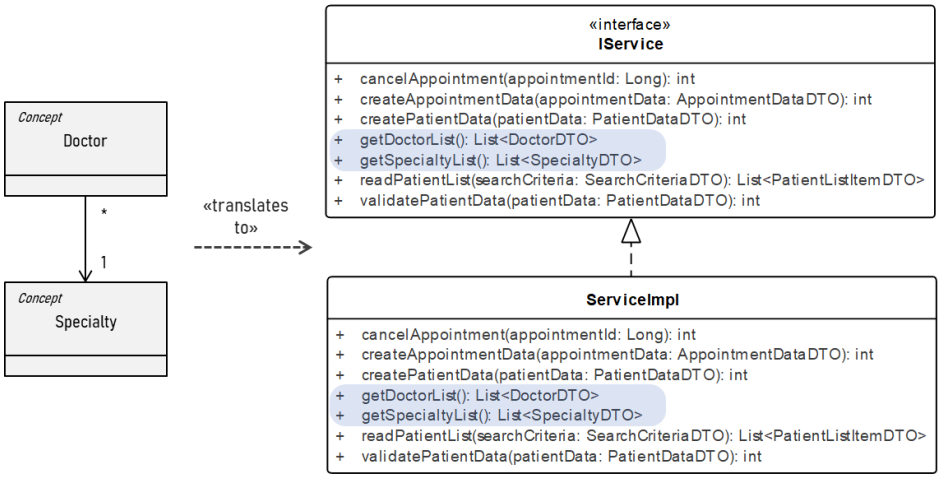
Jeżeli czasownik mapuje się na operację „read”, typem wartości zwracanej przez operację jest klasa obiektu transferu danych odpowiadająca pojęciu pełniącemu rolę dopełnienia bliższego lub lista takich obiektów w przypadku, gdy dopełnieniem jest pojęcie typu ListView. Jest to oczywiste, gdyż rolą tego typu operacji jest pobranie danych z warstwy danych systemu i przekazanie ich do obiektu wywołującego, tj. prezentera. Przykładem takiej operacji na Rysunek 62 jest `readPatientList()`.

Ponadto, zdania którym odpowiada operacja „read” powinny posiadać dopełnienie dalsze, które określa parametr wywołania operacji. Dopełnienie dalsze typu Concept przekłada się na parametr operacji w postaci identyfikatora obiektu biznesowego typu `Long`, który jednoznacznie określa dla jakiego obiektu mają być pobrane dane. W przeciwnym wypadku, niezbędne byłoby określenie reguł biznesowych, według których warstwa modelu zwracałaby dane pojedynczego obiektu, np. ostatnio utworzony lub wybrany losowo, co raczej nie jest typowe w systemach biznesowych. W przypadku zdań typu System-to-ListView, dopełnienie dalsze może być pojęciem typu Concept (niekoniecznie stanowiącym „main concept” dla dopełnienia bliższego) lub pojęciem typu SimpleView. Z tym drugim wariantem mamy do czynienia w przedstawionym przykładzie, gdzie parametrem operacji jest obiekt DTO, stanowiący kryteria wyszukiwania pacjentów. Można sobie wyobrazić, że w takim przypadku dane przekazywane jako parametr używane są w klauzuli WHERE zapytania bazodanowego, która zwraca listę obiektów spełniających zadane warunki.

Jeżeli czasownik w stwierdzeniu dziedzinowym mapuje się na jakąkolwiek operację inną niż „read”, operacja dziedzinowa zwraca zawsze status jej wykonania w postaci liczby typu `int`. Rysunek 62 prezentuje dwa przykłady takich operacji: `validatePatientData()` oraz `createPatientData()`. Zakres i znaczenie zwracanych wartości określają zdania warunkowe, które mogą występować po zdaniach System-to-*. Semantyka translacyjna dla zdań warunkowych przedstawiona jest w kolejnym podrozdziale. Parametrem wywołania takich operacji jest zawsze obiekt DTO odpowiadający dopełnieniu bliższemu. W obu

przykładowych operacjach jest to obiekt klasy `PatientDataDTO` odpowiadający pojęciu „patient data”. Semantyka języka RSL nie określa roli dopełnienia dalszego w tego typu zdaniach.

Reguła M2. *Dla każdego pojęcia typu Concept utwórz w interfejsie* `IService` *oraz w klasie* `ServiceImpl` *operację dziedzinową, bez parametrów wejściowych, której nazwa jest nazwą pojęcia Concept zapisaną w notacji Pascal case z prefiksem „read” oraz sufiksem „List”. Typem wartości zwracanej przez operację jest* `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca pojęciu Concept (patrz: reguła G7).*



Rysunek 63 – Reguła M2: translacja pojęć typu Concept na operacje dziedzinowe

Reguła M2 zilustrowana jest na Rysunek 63. Operacje generowane w ten sposób, nie wynikają bezpośrednio z logiki aplikacji wyrażonej w scenariuszach. Są to operacje pomocnicze używane na potrzeby prezentowania list wyboru (słowników) w interfejsie użytkownika. Operacje te zwracają listy obiektów DTO odpowiadających pojęciom typu Concept (patrz: reguła G7). Przykładowo, dodanie do system nowego lekarza (Doctor) wymaga określenia jego specjalizacji (Specialty). Na formatce dodawania lekarza musi być, zatem, możliwość wyboru jednej ze zdefiniowanych specjalizacji. W takiej sytuacji, klasa reprezentująca formatkę powinna otrzymać od prezentera listę obiektów `SpecialtyDTO` pobranych w wyniku wywołania operacji `getSpecialtyList()`.

5.4 Reguły translacji do warstwy prezentera

Reguły przedstawione w tej części dotyczą scenariuszy przypadków użycia. Definiują one sposób translacji poszczególnych typów zdań scenariuszy na elementy środowiska docelowego: głównie klasy prezenterów oraz – w przypadku niektórych reguł – podstawowe elementy klas warstwy widoku. Implementacja tych reguł pozwala wygenerować kompletny kod logiki aplikacji implementowany przez warstwę prezentera.

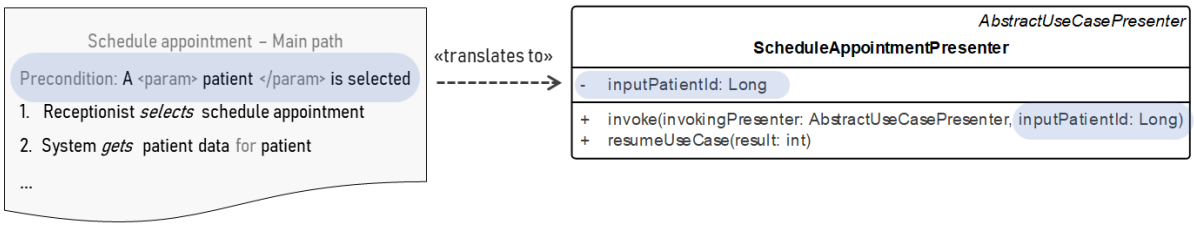
Reguła P1. *Dla każdego zdania typu Precondition określającego parameter wejściowy (nazwa pojęcia typu Concept ujęta w znaczniki <param></param>) warunkujący wykonanie przypadku użycia, utwórz w klasie prezentera odpowiadającej danemu przypadkowi użycia:*

a) prywatny atrybut typu `Long`*, przechowujący wartość parametru wejściowego przypadku użycia, tj. identyfikator obiektu biznesowego; nazwa atrybutu jest nazwą parametru (zawartość znaczników <param></param>) zapisaną w notacji Pascal case z prefiksem „input” i sufiksem „Id”;*

b) drugi parametr wywołania metody `invoke()`*, gdzie typ i nazwa parametru są takie same jak dla wcześniej utworzonego atrybutu;*

c) kod w metodzie `invoke()`*, który inicjalizuje atrybut wartością parametru wywołania.*

a)



b)

```
1  public void invoke(AbstractUseCasePresenter invokingPresenter, Long inputPatientId) {
2      // ...
3      this.inputPatientId = inputPatientId;
4  }
```

Rysunek 64 – Reguła P1: translacja zdania Precondition na elementy klasy prezentera (a); kod metody invoke() (b)

Reguła P1 zilustrowana jest na Rysunek 64. W przedstawionym przykładzie, zdanie Precondition stanowi, że przypadek użycia „Schedule appointment" może być wykonany pod warunkiem określenia konkretnego pacjenta, dla którego ma być umówiona wizyta. Przekłada się to na atrybut `inputPatientId`, który jest unikalnym identyfikatorem pacjenta w systemie. Jest on inicjalizowany wartością parametru o tej samej nazwie w metodzie `invoke()`. Prezenter wywołujący musi, zatem, przekazać wartość tego parametru przy uruchamianiu logiki prezentera `ScheduleAppointmentPresenter`.

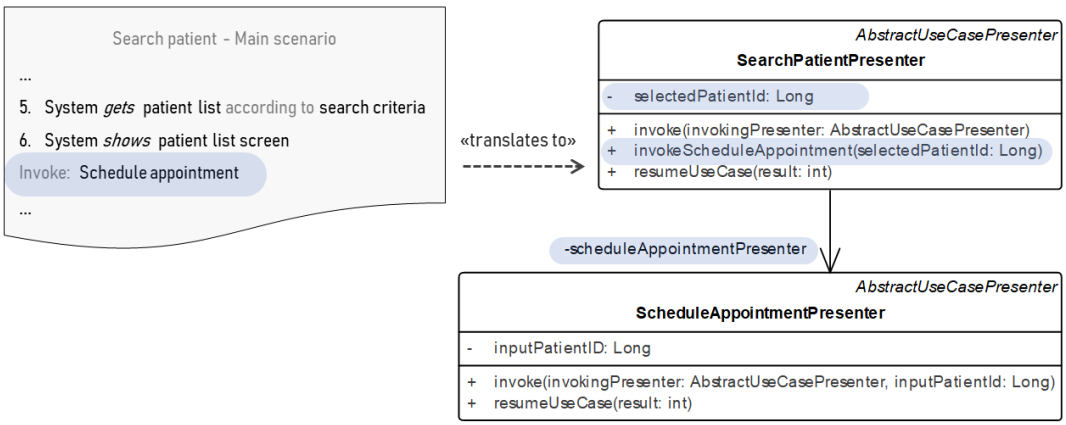
Reguła P2. *Dla każdego zdania typu Invoke, dla którego stan dialogu wskazuje na aktora:*

a) *w klasie prezentera wywołującego przypadku użycia utwórz prywatny atrybut, którego typem jest klasa prezentera wywoływanego przypadku użycia; nazwa atrybutu jest nazwą klasy prezentera przypadku wywoływanego zapisaną w notacji Camel case;*

b) *jeżeli wywoływany przypadek użycia wymaga parametru wejściowego (patrz: reguła P1), to w klasie prezentera przypadku wywołującego utwórz prywatny atrybut typu* `Long` *przechowujący identyfikator obiektu biznesowego, który ma być przekazany do prezentera przypadku wywoływanego; nazwa atrybutu jest nazwą parametru wejściowego wywoływanego przypadku użycia, zapisaną w notacji Pascal case z prefiksem „selected" i sufiksem „Id";*

c) *w klasie prezentera wywołującego przypadku użycia utwórz metodę uruchamiającą logikę przypadku wywoływanego na żądanie aktora; nazwa metody jest nazwą wywoływanego przypadku użycia zapisaną w notacji Camel case z prefiksem „invoke"; kod metody zawiera wywołanie metody* `invoke()` *na obiekcie prezentera przypadku wywoływanego; jeżeli wywoływany przypadek użycia wymaga parametru wejściowego, jako parametr wywołania przekazywana jest wartość atrybutu opisanego w części b) reguły.*

a)



b)

```
1  public void invokeScheduleAppointment(Long selectedPatientId) {
2      // ...
3      scheduleAppointmentPresenter.invoke(this, selectedPatientId);
4  }
```

Rysunek 65 – Reguła P2: translacja zdania Invoke na element klasy prezentera (a), kod metody uruchamiającej wywoływany przypadek użycia (b)

Reguła P2 zilustrowana jest na Rysunek 65. Jeżeli w scenariuszu przypadku użycia występuje zdanie typu Invoke w miejscu, w którym stan dialogu wskazuje na aktora, zdaniu takiemu odpowiada kod warunkowego wywołania przypadku użycia, do którego odnosi się to zdanie. W prezentowanym przykładzie, przypadkiem wywoływanym jest „Schedule appointment". Warunkowe wywołanie oznacza, że funkcjonalność przypadku wywoływanego jest uruchamiana na żądanie użytkownika. Stąd, w prezenterze przypadku wywołującego `SearchPatientPresenter` musi istnieć metoda, za pomocą której warstwa widoku przekazuje to żądanie do klasy prezentera. W naszym przykładzie jest to `invokeScheduleAppointment()`. Ponieważ przypadek użycia „Schedule appointment" wymaga wskazania konkretnego pacjenta, dla którego ma być umówiona wizyta, wspomniana metoda posiada parametr wywołania `selectedPatientId`. Kod metody sprowadza się do wywołania metody `invoke()` na obiekcie prezentera docelowego. Odwołanie do tego obiektu możliwe jest poprzez atrybut `scheduleAppointmentPresenter`, który na rysunku zilustrowany jest w postaci asocjacji skierowanej. Dynamikę tego wywołania prezentuje fragment włączony na diagramie sekwencji na Rysunek 51.

Zdanie Invoke wykonywane warunkowo przez aktora, w oczywisty sposób przekłada się na wyzwalacz dostępny na ekranie aplikacji, z poziomu którego użytkownik może wywołać

żądaną funkcjonalność. Translację zdania Invoke na kod klasy ekranu opisuje reguła V9 w rozdziale 5.5.

Reguła P3. *Dla każdego zdania typu Invoke, dla którego stan dialogu wskazuje na system:*

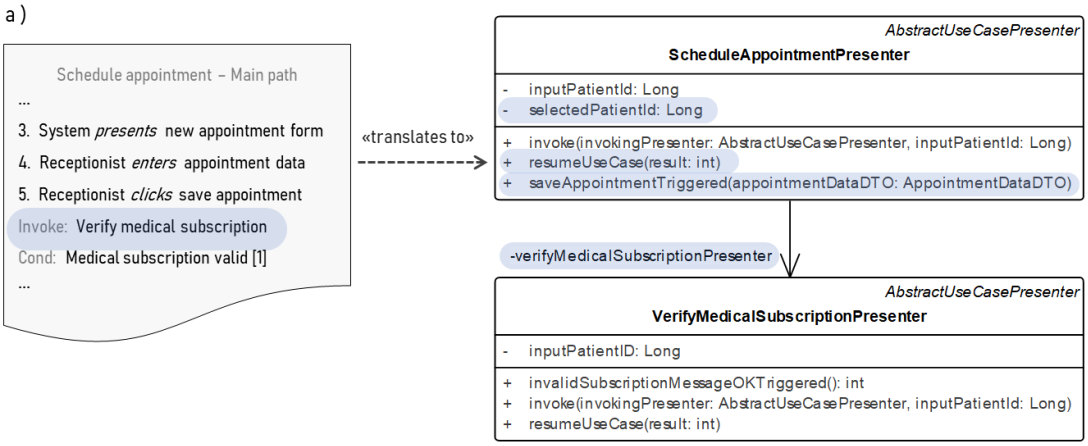
a) w klasie prezentera wywołującego przypadku użycia utwórz prywatny atrybut, którego typem jest klasa prezentera wywoływanego przypadku użycia; nazwa atrybutu jest nazwą klasy prezentera przypadku wywoływanego zapisaną w notacji Camel case;

b) jeżeli wywoływany przypadek użycia wymaga parametru wejściowego (patrz: reguła P1), to w klasie prezentera przypadku wywołującego utwórz prywatny atrybut typu `Long` *przechowujący identyfikator obiektu biznesowego, który ma być przekazany do prezentera przypadku wywoływanego; nazwa atrybutu jest nazwą parametru wejściowego wywoływanego przypadku użycia, zapisaną w notacji Pascal case z prefiksem „selected" i sufiksem „Id";*

c) w klasie prezentera wywołującego przypadku użycia utwórz kod, który (1) atrybutowi `resumeId` *przypisuje wartość będącą identyfikatorem zdania Invoke a następnie (2) wywołuje metodę* `invoke()` *na obiekcie prezentera przypadku wywoływanego; jeżeli wywoływany przypadek użycia wymaga parametru wejściowego, jako parametr wywołania przekazywana jest wartość atrybutu opisanego w części b) reguły; powyższy kod umieść w metodzie odpowiadającej zdaniu, które przed wykonaniem zdania Invoke zmieniło stan dialogu na „system" lub w metodzie* `resumeUseCase()` *jeżeli zdanie Invoke jest bezpośrednio poprzedzone innym zdaniem Invoke wykonywanym przez system;*

d) w metodzie `resumeUseCase()` *utwórz pusty blok warunkowy* `if`*, który jest wykonywany gdy wartość* `resumeId` *jest zgodna z identyfikatorem zdania Invoke.*

a)



b)

```
1  public void saveAppointmentTriggered(AppointmentDataDTO appointmentDataDTO) {
2      // ...
3      resumeId = 7;
4      verifyMedicalSubscriptionPresenter.invoke(this, selectedPatientId);
5  }
```

c)

```
1  public void resumeUseCase(int result) {
2      // ...
3      if (resumeId == 7) {
4          // Application logic to be executed when "Verify medical subscription" is finished
5      }
6  }
```

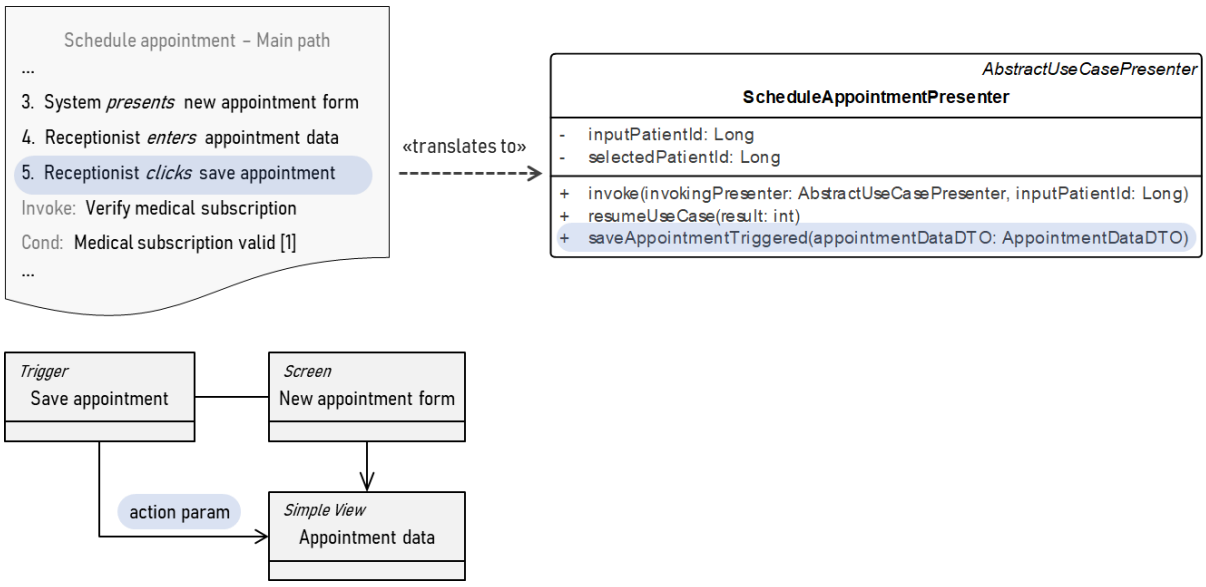
Rysunek 66 – Reguła P3: translacja zdania Invoke na element klasy prezentera (a), kod wywołania docelowego przypadku użycia (b), kod metody resumeUseCase() (c)

Reguła P3 zilustrowana jest na Rysunek 66. Jeżeli w scenariuszu przypadku użycia występuje zdanie typu Invoke w miejscu, w którym stan dialogu wskazuje na system, zdaniu takiemu odpowiada kod bezwarunkowego wywołania przypadku użycia, do którego odnosi się to zdanie. W prezentowanym przykładzie, przypadkiem wywoływanym jest „Verify medical subscription". Bezwarunkowe wywołanie oznacza, że funkcjonalność przypadku wywoływanego jest uruchamiana niezależnie od decyzji użytkownika. W odróżnieniu od sytuacji przedstawionej w poprzedniej regule, nie potrzebna jest tutaj metoda prezentera odpowiadająca na żądanie użytkownika. Wywołanie metody `invoke()` na obiekcie prezentera docelowego jest częścią sekwencji kroków wykonywanych przez system. Sekwencja taka zawiera kod odpowiadający wszystkim zdaniom System-to-system znajdującym się w scenariuszu pomiędzy zdaniem Invoke i wcześniejszym zdaniem Actor-to-Trigger. Instrukcja wywołania docelowego przypadku użycia znajduje się zawsze na końcu takiej sekwencji. W przedstawionym przykładzie wywołanie logiki przypadku „Verify medical subscription" jest częścią sekwencji kodu wykonywanej przez prezenter „Schedule

appointment" w ramach metody `saveAppointmentTriggered()`. Dynamika tego wywołania zaprezentowana jest na diagramie sekwencji na Rysunek 52.

Po zakończeniu bezwarunkowego wywołania docelowego przypadku użycia, musi zostać wznowione wykonywanie wywołującego przypadku użycia. W tym celu tworzona jest metoda `resumeUseCase()`, której kod implementuje logikę wynikającą z części scenariusza następującej po każdym zdaniu Invoke. Aby poprawnie zidentyfikować zdanie Invoke, po którym następuje wznowienie, w metodzie `resumeUseCase()` tworzone są bloki warunkowe `if` odpowiadające poszczególnym zdaniom. Warunek bloku `if` sprawdza wartość zmiennej `resumeId`, która jest inicjalizowana tuż przed wywołaniem przypadku użycia odpowiadającego zdaniu Invoke.

Reguła P4. *Dla każdego zdania typu Actor-to-Trigger, które nie następuje bezpośrednio po zdaniu typu Precondition, utwórz w klasie prezentera publiczną metodę wywoływaną przez warstwę widoku w odpowiedzi na akcję użytkownika na danym wyzwalaczu. Nazwą metody jest nazwa wyzwalacza zapisana w notacji Camel case z sufiksem „Triggered". Jeżeli pojęcie Trigger użyte w zdaniu jest powiązane relacją „action param" z pojęciem reprezentującym widok danych, powiązanemu widokowi danych odpowiada parametr wywołania metody. Typ parametru określa klasa DTO odpowiadająca wskazywanemu widokowi danych a nazwa jest nazwą klasy DTO zapisaną w notacji Camel case.*



Rysunek 67 – Reguła P4: translacja zdania Actor-to-Trigger na elementy klasy prezentera

Reguła P4 zilustrowana jest na Rysunek 67. Zdanie Actor-to-Trigger zmienia stan dialogu z „aktor” na „system”. Dzieje się to za sprawą akcji wykonanej przez użytkownika na wyzwalaczu, przy czym dotyczy to tylko akcji, które mapują się na operację „Select” (patrz: rozdział 3.2.3). W odpowiedzi użytkownik spodziewa się reakcji systemu polegającej na wykonaniu logiki zdefiniowanej przez dalszą część scenariusza. Konstrukcji takiej odpowiada operacja w klasie prezentera, która zawiera kod implementujący wspomnianą logikę. Jest ona wywoływana przez warstwę prezentera w ramach obsługi zdarzenia na wyzwalaczu, gdyż widok w przyjętym środowisku docelowym nie powinien implementować logiki aplikacji. W przedstawionym przykładzie, jest to metoda `saveAppointmentTriggered()`. Działanie tej metody, polega na wykonaniu operacji biznesowych na danych reprezentowanych przez widok danych „Appointment data” (relacja „action param”). Dane te są przekazywane w postaci obiektu DTO reprezentującego ten widok (patrz: reguły G7–G9). Sam kod metody `saveAppointmentTriggered()` nie wynika z tej reguły, ale jest wynikiem translacji sekwencji zdań występujących pomiędzy omawianym zdaniem Actor-to-Trigger a zdaniem zmieniającym stan dialogu na „aktor”.

Reguła P5. *Dla każdego zdania typu System-to-Screen:*

a) w klasie prezentera utwórz kod wywołania operacji interfejsu `IView` *odpowiadającej akcji wykonanej przez system na danym ekranie; Nazwa wywoływanej metody jest nazwą stwierdzenia dziedzinowego zawartego w zdaniu System-to-Screen zapisaną w notacji Camel case, przy czym, jeżeli orzeczenie mapuje się na standardową operację „show”, „close” lub „refresh”, zamiast czasownika, używana jest nazwa operacji; Kod wywołania umieść w metodzie odpowiadającej zdaniu, które przed wykonaniem zdania System-to-Screen zmieniło stan dialogu na „system” lub w metodzie* `resumeUseCase()` *jeżeli zdanie System-to-Screen jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system;*

b) po wywołaniu operacji interfejsu `IView` *typu „open” lub „close” umieść wywołanie metody odpowiednio* `pageOpened()` *lub* `pageClosed()`*;*

c) w klasie odpowiadającej pojęciu typu Screen (patrz: reguła G3) utwórz (jeśli jeszcze nie został utworzony) prywatny atrybut o nazwie `presenter`*, którego typem jest klasa prezentera; W metodzie* `init()` *utwórz parametr wywołania, którego typ*

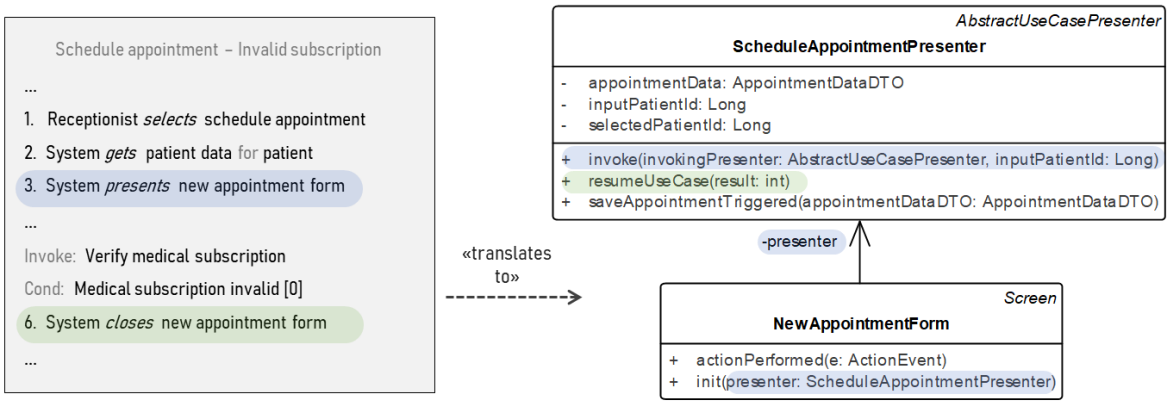
i nazwa są takie same jak dla wcześniej utworzonego atrybutu; Do metody `init()` *dodaj kod inicjalizujący atrybut wartością parametru wywołania.*

Reguła P5 zilustrowana jest na Rysunek 68. Jak wynika z reguły G3, każdemu pojęciu typu Screen odpowiada klasa w warstwie widoku. Stwierdzeniom dziedzinowym zawartym w tych pojęciach odpowiadają, natomiast, operacje na klasach, np. wyświetlenie lub zamknięcie ekranu (patrz: rozdział 3.2.3). Operacje te zgrupowane są w interfejsie `IView` – szczegóły ich tworzenia opisują reguła V1 (patrz: rozdział 5.5). Pojawienie się w scenariuszu zdania System-to-Screen jest jednoznaczne z wywołaniem metody implementującej taką operację z poziomu kodu prezentera.

W scenariuszu przedstawionym w przykładzie występują dwa zdania, które przekładają się na wywołania odpowiednich metod. W przypadku zdania numer 3, orzeczenie mapuje się na operację wyświetlenia ekranu `showNewAppointmentForm(…)`. Parametrem wywołania metody `show…()` jest zawsze obiekt wywołującego prezentera (`this`). Umożliwia to tworzonemu ekranowi komunikację zwrotną z warstwą prezentacji. W tym celu w klasie ekranu tworzony jest atrybut `presenter`, który przechowuje referencję do odpowiedniego obiektu przekazywaną jako parametr metody `init()`. W przypadku ekranów, których zadaniem jest wyświetlenie danych przekazanych przez obiekt prezentera, dane takie przekazywane są w postaci obiektów transferu danych, jako dodatkowy parametr wywołania operacji `show…()`.

W przyapadku zdania numer 6, orzeczenie mapuje się na operację zamknięcia ekranu `closeNewAppointmentForm()`. Wywołanie tej operacji umieszczone jest w metodzie `resumeUseCase()`, gdyż – zgodnie ze scenariuszem – ta część logiki aplikacji wykonywana jest wtedy, gdy wywołany wcześniej przypadek „Verify medical subscription” zwróci rezultat równy 0 (translację zdań warunkowych opisuje reguła P12).

a)



b)

```
1  public void invoke(AbstractUseCasePresenter invokingPresenter, Long inputPatientId) {
2      // ...
3      view.showNewAppointmentForm(this);
4      pageOpened();
5  }
6
7  public void resumeUseCase(int result) {
8      if (resumeId == 7) {
9          // Application logic to be executed when "Verify medical subscription" is finished
10         if (result == 0) { // Medical subscription invalid [0]
11             view.closeNewAppointmentForm();
12             pageClosed();
13         } else if (result == 1) { // Medical subscription valid [1]
14             // ...
15         }
16 }
```

c)

```
1  public void init(ScheduleAppointmentPresenter presenter) {
2      // ...
3      this.pressenter = presenter;
4  }
```

Rysunek 68 – Reguła P5: translacja zdania System-to-Screen na elementy klasy prezentera i widoku (a), kod wywołania w klasie prezentera (b), kod metody init() w klasie ekranu (c)

Reguła P6. *Dla każdego zdania typu System-to-Message oraz System-to-Confirmation, w którym orzeczenie mapuje się na operację „show”:*

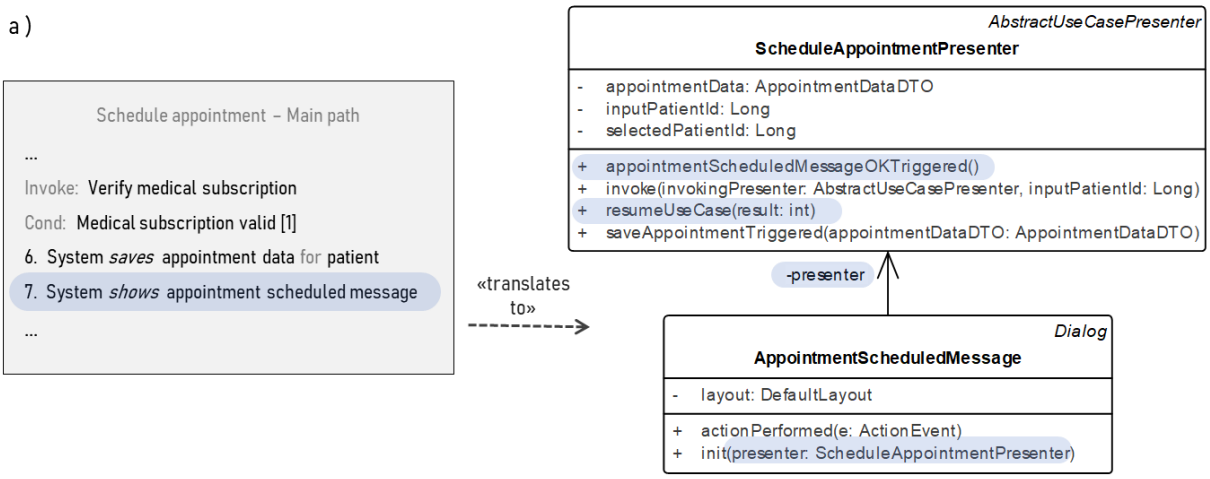
a) *w klasie prezentera utwórz kod wywołania operacji interfejsu* `IView` *wyświetlającej dany komunikat; Nazwa wywoływanej metody jest nazwą pojęcia Message lub Confirmation zapisaną w notacji Camel case z przedrostkiem „show”; Kod wywołania umieść w metodzie odpowiadającej zdaniu, które przed wykonaniem zdania System-to-Message lub System-to-Confirmation zmieniło stan dialogu na „system” lub w metodzie* `resumeUseCase()` *jeżeli zdanie System-to-Message lub System-to-Confirmation jest bezpośrednio poprzedzone zdaniem Invoke;*

b) *w klasie odpowiadającej pojęciu typu Message lub Confirmation (patrz: reguła G4) utwórz (jeśli jeszcze nie został utworzony) prywatny atrybut o nazwie* `presenter`*, którego typem jest klasa prezentera; W metodzie* `init()` *utwórz parametr wywołania, którego typ i nazwa są takie same jak dla wcześniej utworzonego atrybutu; Do metody* `init()` *dodaj kod inicjalizujący atrybut wartością parametru wywołania.*

Reguła P7. *Dla każdego zdania typu System-to-Message, w którym orzeczenie mapuje się na operację „show" utwórz w klasie prezentera publiczną metodę wywoływaną przez warstwę widoku w odpowiedzi na akcję użytkownika polegającą na zamknięciu komunikatu. Nazwą metody jest nazwa pojęcia typu Message zapisana w notacji Camel case z sufiksem „OKTriggered".*

Reguła P6 oraz P7 są zilustrowane na Rysunek 69. Pierwsza z nich jest niemal identyczna jak reguła P5 z tą różnicą, że ma zastosowanie tylko w przypadku zdań, których orzeczenie mapuje się na operację typu „show". Operacja „close" w przypadku okien dialogowych nie jest potrzebna – zamknięcie okna następuje po użyciu wyzwalacza prezentowanego razem z komunikatem. W przypadku komunikatu typu „Message" jest to standardowo przycisk „OK" a w przypadku komunikatów typu „Confirmation" może to być szereg opcji wynikających ze zdań warunkowych następujących bezpośrednio po zdaniu System-to-Confirmation. W tym pierwszym przypadku zastosowanie ma dodatkowa reguła P7. W momencie zamknięcia komunikatu przyciskiem „OK", obsługa zdarzenia w obiekcie komunikatu informuje o tym fakcie obiekt prezentera poprzez wywołanie odpowiedniej metody. W przedstawionym przykładzie jest to metoda `appointmentScheduledMessageOKTriggered()`. W przypadku zdania System-to-Conformation, musi istnieć analogiczna metoda dla każdej opcji dostępnej dla użytkownika w oknie dialogowym. Jak wspomniano powyżej, opcje te wynikają ze zdań warunkowych – sposób ich tworzenia definiuje reguła P11.

a)



b)

```
1  public void resumeUseCase(int result) {
2      if (resumeId == 7) {
3          // Application logic to be executed when "Verify medical subscription" is finished
4          if (result == 0) { // Medical subscription invalid [0]
5              view.closeNewAppointmentForm();
6          } else if (result == 1) { // Medical subscription valid [1]
7              // ...
8              view.showAppointmentScheduledMessage(this);
9          }
10 }
```

c)

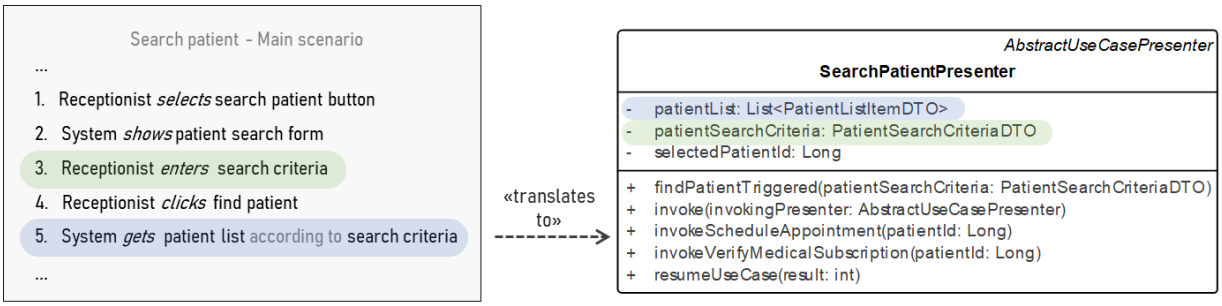
```
1  public void init(ScheduleAppointmentPresenter presenter) {
2      // ...
3      this.presenter = presenter;
4  }
```

Rysunek 69 – Reguła P6 i P7: translacja zdania System-to-Message na elementy klasy prezentera i widoku (a), kod wywołania w klasie prezentera (b), kod metody init() w klasie komunikatu (c)

Reguła P8. *Dla każdego zdania typu System-to-SimpleView, Actor-to-SimpleView, System-to-ListView oraz Actor-to-ListView utwórz w klasie prezentera prywatny atrybut, którego nazwa jest nazwą pojęcia zapisaną w notacji Camel case. Jeżeli dopełnienie zdania odnosi się do pojęcia SimpleView, typem atrybutu jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G8). Jeżeli dopełnienie zdania odnosi się do pojęcia ListView, typem atrybutu jest java.util.List<T>, gdzie parametrem typu listy T jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G9).*

Reguła P8 zilustrowana jest na Rysunek 70. Jeżeli w scenariuszu występuje odwołanie do pojęcia typu Simple View lub List View oznacza to, że w klasie odpowiedniego prezentera istnieje atrybut pozwalający tymczasowo przechowywać dane reprezentowane przez to pojęcie.

Prezenter przechowuje te dane w obiekcie klasy DTO odpowiadającej danemu pojęciu i w odpowiednim momencie (zgodnie z logiką zapisaną w scenariuszu) przekazuje je do warstwy widoku bądź do warstwy modelu. W przedstawionym przykładzie, mamy dwa zdania, których dopełnienia bliższe odnoszą się do pojęć typu Simple View (zdanie nr 3) oraz List View (zdanie nr 5). Przekłada się to na dwa atrybuty w klasie prezentera implementującego logikę scenariusza. Typ atrybutu określa klasa DTO odpowiadająca danemu pojęciu. W przypadku Simple View atrybut przechowuje pojedynczy obiekt odpowiedniej klasy DTO a w przypadku List View – listę obiektów, których typ określa odpowiednia klasa DTO. Struktura klas DTO dla pojęć typu Simple View oraz List View opisują reguły G8 i G9.



Rysunek 70 – Reguła P8: translacja zdania, którego dopełnienie odnosi się do pojęcia typu Simple View lub List View na atrybut klasy prezentera

Reguła P9. *Dla każdego zdania typu System-to-SimpleView lub System-to-ListView utwórz w klasie prezentera kod wywołania operacji dziedzinowej, która odpowiada stwierdzeniu dziedzinowemu pojęcia Simple View lub List View, do którego odnosi się zdanie (patrz: reguła M1).*

W przypadku, gdy orzeczenie w zdaniu mapuje się na akcję „read”, to:

a) wartość zwracana przez wywoływaną operację przypisywana jest do atrybutu odpowiadającego pojęciu Simple View lub List View (patrz: reguła P8);

b) jeżeli dopełnieniem dalszym w zdaniu jest pojęcie typu Concept, które odpowiada parametrowi wejściowemu w zdaniu Precondition, to parametrem wywołania operacji dziedzinowej jest atrybut odpowiadający parametrowi wejściowemu (patrz: reguła P1);

c) *jeżeli dopełnieniem dalszym w zdaniu jest inne pojęcie typu Simple View, to parametrem wywołania operacji dziedzinowej jest atrybut typu DTO odpowiadający temu pojęciu (patrz: reguła P8).*

W przypadku, gdy orzeczenie w zdaniu mapuje się na akcję inną niż „read", to:

a) *parametrem wywołania operacji dziedzinowej jest atrybut odpowiadający pojęciu Simple View lub List View pełniącego w zdaniu rolę dopełnienia bliższego (patrz: reguła P8);*

b) *jeżeli dopełnieniem dalszym w zdaniu jest pojęcie typu Concept, które odpowiada parametrowi wejściowemu w zdaniu Precondition, to parametrem wywołania operacji dziedzinowej jest atrybut odpowiadający parametrowi wejściowemu (patrz: reguła P1);*

c) *jeżeli po zdaniu System-to-SimpleView lub System-to-ListView występuje zdanie warunkowe, to wartość zwracana przez wywoływaną operację przypisywana jest do lokalnej zmiennej o nazwie* `result`.

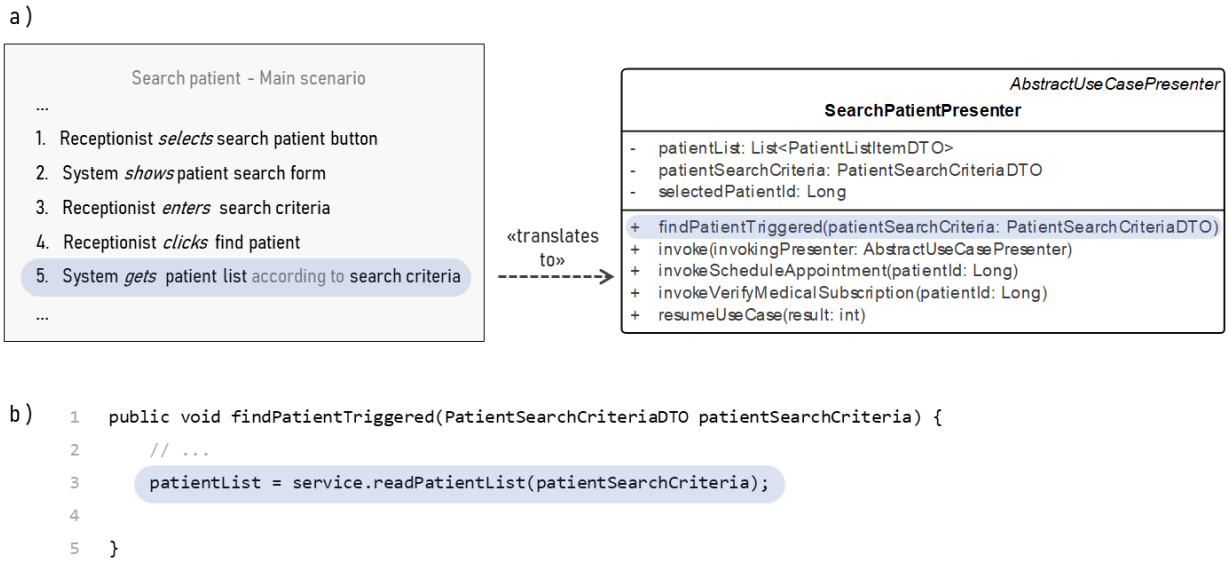
Kod wywołania umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system" lub w metodzie `resumeUseCase()` *jeżeli zdanie System-to-SimpleView lub System-to-ListView jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.*

Reguła P9 zilustrowana jest na Rysunek 71. Każde zdanie System-to-SimpleView oraz System-to-ListView oznacza wykonanie przez system operacji dziedzinowej na danych reprezentowanych odpowiednio przez pojęcie Simple View lub List View. Operacje takie definiowane są w warstwie modelu na podstawie stwierdzeń dziedzinowych odpowiednich pojęć (patrz: reguła M1), natomiast ich wywołania umiejscowione są w kodzie prezentera. Kod wywołania oraz parametry wywołania zależą od typu wywoływanej operacji.

W przypadku operacji typu „read" wartość zwracana przez wywoływaną operacje dziedzinową w postaci obiektu lub listy obiektów DTO przypisywana jest zawsze do atrybutu utworzonego dla pojęcia Simple View lub List View użytego w zdaniu w roli dopełnienia bliższego (patrz: reguła P8). Parametr wywołania zależy natomiast od dopełniania dalszego jak opisuje to omawiana reguła. W przedstawionym przykładzie orzeczenie w zdaniu nr 5 mapuje się na operację typu „read", dopełnienie bliższe odnosi się do pojęcia typu List View a dopełnienie dalsze – do pojęcia typu Simple View. Przekłada się to na wywołanie operacji

`readPatientList()`, gdzie parametrem wywołania jest obiekt DTO odpowiadający dopełnieniu dalszemu „search criteria” a wynik zwracany przez operację przypisywany jest do atrybutu odpowiadającego dopełnieniu bliższemu „patient list”.

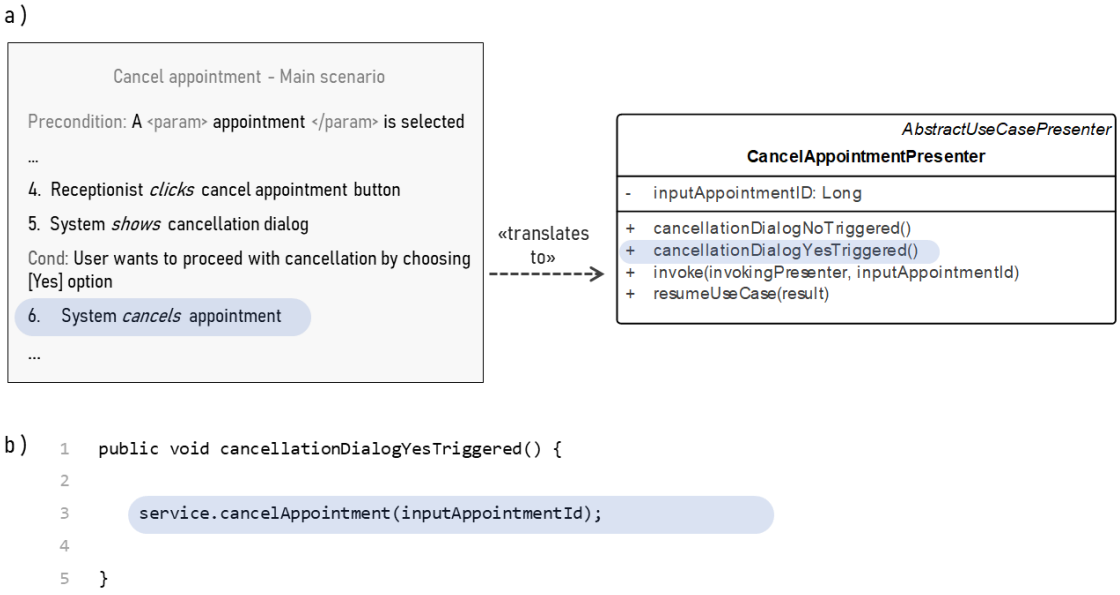
W przypadku operacji innych niż „read”, parametrem wywołania operacji dziedzinowej jest atrybut odpowiadający dopełnieniu bliższemu a status wykonania operacji przypisywany jest do lokalnej zmiennej `result` tylko wtedy, gdy po omawianym zdaniu występuje zdanie warunkowe odnoszące się do rezultatu wykonania operacji. W obecnej wersji semantyka translacyjna nie określa reguł translacji dla dopełnienia dalszego w tego typu zdaniach.



Rysunek 71 – Reguła P9: translacja zdania System-to-ListView na kod wywołania operacji dziedzinowej

Reguła P10. *Dla każdego zdania typu System-to-Concept, w którym dopełnienie odpowiada parametrowi wejściowemu w zdaniu Precondition utwórz w klasie prezentera kod wywołania operacji dziedzinowej, która odpowiada stwierdzeniu dziedzinowemu pojęcia Concept, do którego odnosi się zdanie (patrz: reguła M1). Parametrem wywołania operacji jest atrybut odpowiadający parametrowi wejściowemu (patrz: reguła P1). Jeżeli po zdaniu System-to-Concept występuje zdanie warunkowe, to wartość zwracana przez wywoływaną operację przypisywana jest do lokalnej zmiennej o nazwie* `result`*. Kod wywołania umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system” lub w metodzie* `resumeUseCase()` *jeżeli zdanie System-to-Concept jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.*

Reguła P10 zilustrowana jest na Rysunek 72. Reguła ta jest podobna do reguły P9 z tą różnicą, że zastosowanie zdania System-to-Concept jest ograniczone do operacji, dla których parametrem wywołania jest parametr wejściowy przypadku użycia. Są to operacje, których wykonanie wymaga jednoznacznej identyfikacji obiektu biznesowego, na którym operacja ma być wykonana, np. usuwanie, anulowanie, itp. Przykład na Rysunek 72 przedstawia fragment scenariusza dla przypadku użycia „Cancel appointment". Dopełanienie w zdaniu System-to-Concept oraz parametr wejściowy w zdaniu Precondition odnoszą się do tego samego pojęcia. Konstrukcji takiej odpowiada kod wywołania operacji dziedzinowej `cancelAppointment()`. Parametrem jej wywołania jest atrybut `inputAppointmentId` utworzony dla parametru wejściowego, zgodnie z regułą P1.

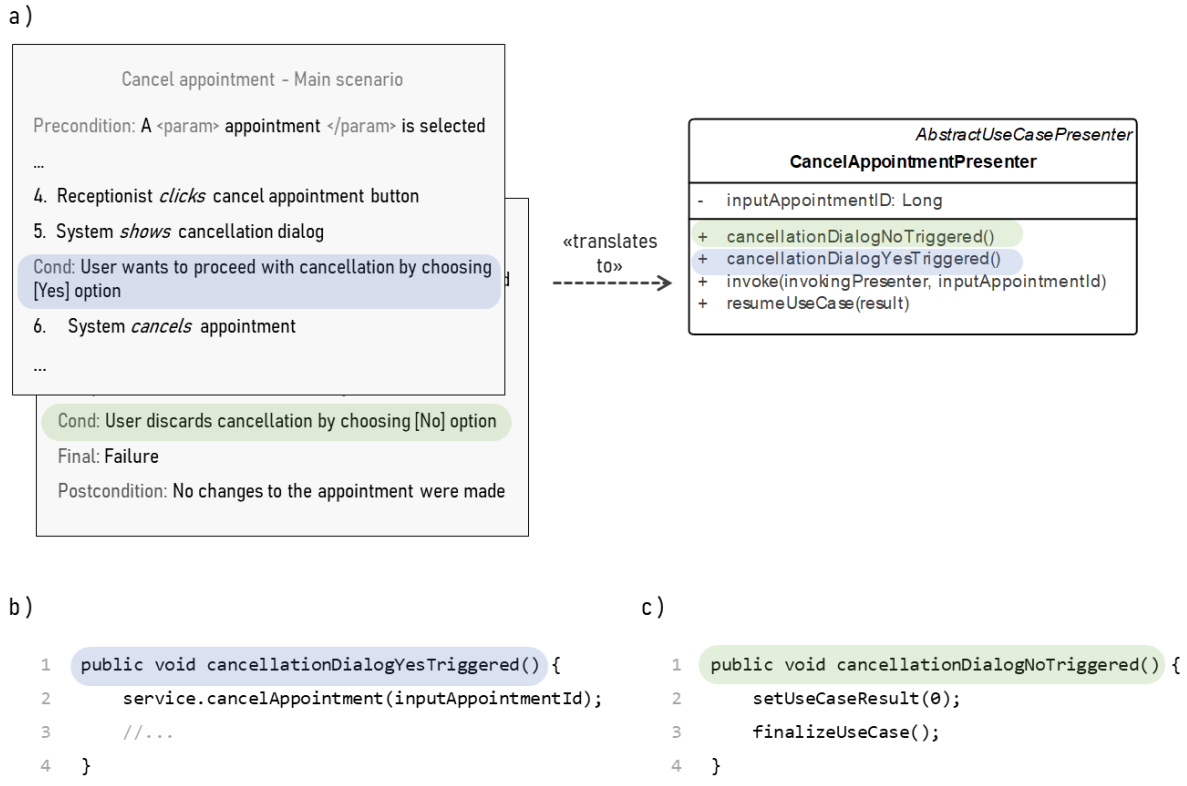


Rysunek 72 – Reguła P10: translacja zdania System-to-Concept na kod wywołania operacji dziedzinowej

Reguła P11. *Dla każdego zdania typu Condition następującego bezpośrednio po zdaniu System-to-Confirmation utwórz w klasie prezentera publiczną metodę wywoływaną przez warstwę widoku w momencie wyboru przez użytkownika wyzwalacza, do którego odnosi się zdanie warunkowe (nazwa wyzwalacza ujęta w nawiasy kwadratowe w tekście zdania warunkowego) i który jest prezentowany razem z komunikatem. Nazwa metody jest konkatenacją nazwy pojęcia typu Conformation oraz nazwy wyzwalacza z sufiksem „Triggered", zapisaną w notacji Camel case.*

Reguła P11 zilustrowana jest na Rysunek 73. Zdania warunkowe występujące po zdaniu typu System-to-Confirmation definiują wyzwalacze powiązane z komunikatem. Wybór przez

użytkownika każdego z tych wyzwalaczy powoduje wykonanie logiki aplikacji przewidzianej dla wybranej opcji. Logika ta zdefiniowana jest w odpowiednich metodach klasy prezentera, które są wywoływane z poziomu klasy implementującej komunikat, w ramach obsługi zdarzenia na danym wyzwalaczu. Przedstawiony przykład dotyczy przypadku użycia polegającego na anulowaniu wcześniej umówionej wizyty. Przed anulowaniem wizyty, system prosi użytkownika o potwierdzenie tej operacji poprzez wyświetlenie komunikatu z dwoma wyzwalaczami („Yes" oraz „No") określonymi w zdaniach warunkowych – jedno w scenariuszu głównym, drugie – w alternatywnym. Zdania te mapują się na dwie metody w klasie prezentera: `cancellationDialogYesTriggered()` oraz `cancellationDialogNoTriggered()`. Kod logiki dalszej części każdego scenariusza umieszczony jest w odpowiedniej metodzie.



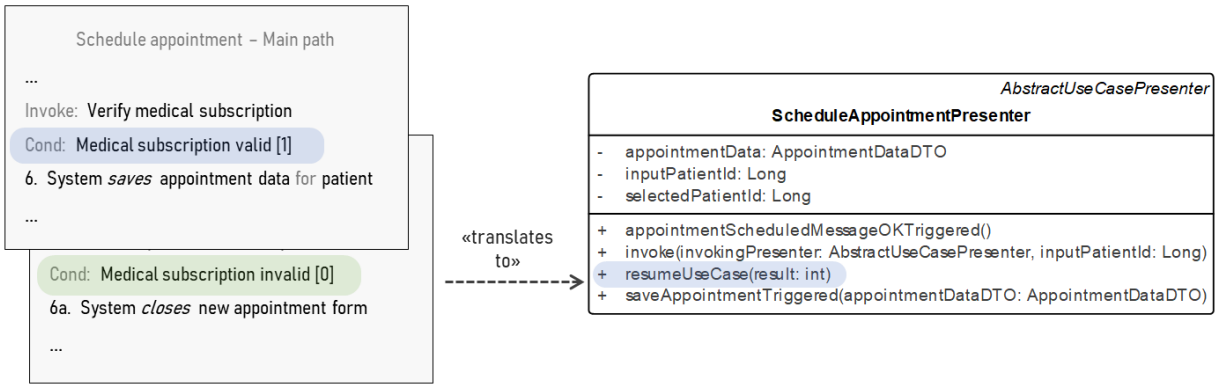
Rysunek 73 – Reguła P11: translacja zdania warunkowego na metody klasy prezentera

Reguła P12. *Dla każdego zdania typu Condition następującego bezpośrednio po zdaniu Invoke, dla którego stan dialogu wskazuje na system utwórz w metodzie* `resumeUseCase()` *w klasie prezentera instrukcję warunkową* `if…else if`*, w którym wyrażenia logiczne sprawdzają wartość zmiennej* `result`*, zawierającej rezultat wykonania wywołanego*

wcześniej przypadku użycia. Jeden z bloków warunkowych jest wykonywany gdy wywołany przypadek użycia zakończył się sukcesem (`result == 1`*), drugi – gdy wywołany przypadek użycia zakończył się niepowodzeniem lub jego wykonanie zostało przerwane (*`result == 0` *lub* `result == -1`*). Instrukcja warunkowa powinna znajdować się w bloku warunkowym* `if`*, którego wyrażenie logiczne odnosi się do atrybutu* `resumeId` *odpowiadającego zdaniu Invoke poprzedzającemu zdanie warunkowe.*

Reguła P12 zilustrowana jest na Rysunek 74. W przypadku, kiedy przebieg scenariusza zależy od rezultatu wykonania innego przypadku użycia (wywołanego przez system), po zdaniu Invoke muszą wystąpić dwa zdania warunkowe: jedno definiujące scenariusz wykonywany w przypadku pomyślnego rezultatu (musi zawierać znacznik „[1]”), drugie – w przypadku przeciwnym. Konstrukcji takiej odpowiada instrukcja warunkowa umieszczona w metodzie `resumeUseCase()`. Instrukcja ta zawiera dwa wyrażenia logiczne, które sprawdzają wartość parametru `result`, zawierającego wynik wykonania wywoływanego przypadku użycia (1 – w przypadku powodzenia, 0 lub -1 – w przypadku niepowodzenia bądź przerwania wykonywania przypadku wywoływanego). Przykład takiej instrukcji warunkowej przedstawia Rysunek 74. Należy zauważyć, że blok warunkowy zagnieżdżony jest w innym bloku warunkowym, który określa warunek powrotu dla tego konkretnego zdania Invoke (patrz: reguła P3).

a)



b)

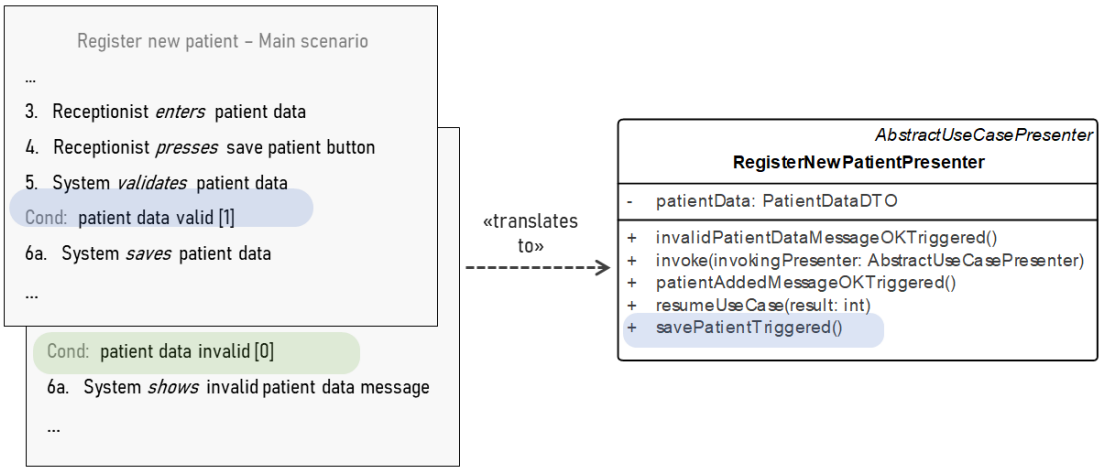
```
1  public void resumeUseCase(int result) {
2      if (resumeId == 7) {
3          // Application logic to be executed when "Verify medical subscription" is finished
4          if (result == 1) {    // Medical subscription valid [1]
5              // ...
6          } else if (result == 0 || result == -1) {    // Medical subscription invalid [0]
7              // ...
8          }
9      }
10 }
```

Rysunek 74 – Reguła P12: translacja zdania warunkowego na kod klasy prezentera

Reguła P13. *Dla każdego zdania typu Condition następującego bezpośrednio po zdaniu System-to-SimleView lub System-to-Concept utwórz w klasie prezentera instrukcję warunkową* `if…else if`*, w którym wyrażenia logiczne sprawdzają rezultat wywołania metody odpowiadającej zdaniu System-to-SimpleView. Operacji typu „read” odpowiada jedno wyrażenie logiczne w bloku* `if`*, który jest wykonywany jeśli zwrócony obiekt jest różny od null, co odpowiada wartości 1 ujętej w nawiasy kwadratowe w tekście zdania warunkowego. W przeciwnym razie wykonywany jest blok* `else`*, co odpowiada wartości 0 w tekście zdania warunkowego. W przypadku operacji innej niż „read”, każdemu zdaniu warunkowemu odpowiada jedno wyrażenie logiczne, które sprawdza czy wartość zmiennej* `result` *jest równa wartości ujętej w nawiasy kwadratowe w tekście odpowiedniego zdania warunkowego. Instrukcję warunkową* `if…else if` *umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system” lub w metodzie* `resumeUseCase()` *jeżeli zdanie System-to-SimpleView jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.*

Reguła P13 zilustrowana jest na Rysunek 75. Zdania warunkowe, które w scenariuszu poprzedzone są zdaniem typu System-to-SimpleView, przekładają się na instrukcję warunkową, w której każde wyrażenie logiczne (a zatem każdy warunkowy blok kodu) odpowiada jednemu zdaniu warunkowemu. Wyrażenia te odnoszą się do rezultatu wywołania operacji dziedzinowej odpowiadającej poprzedzającemu zdaniu System-to-SimpleView (patrz: reguła 9). Wyrażenie jest prawdziwe, jeżeli rezultat jest zgodny z wartością podaną w danym zdaniu warunkowym, przy czym wyrażenie zależy od typu operacji dziedzinowej. Dla operacji typu ”read” wyrażenie sprawdza czy operacja zwróciła oczekiwany obiekt z danymi (wartość musi być inna niż „null”) czy nie. W takich sytuacjach, po zdaniu System-to-SimpleView mogą wystąpić maksymalnie dwa zdania warunkowe – jednemu odpowiada blok warunkowy `if` a drugiemu blok warunkowy `else`. Dla pozostałych typów operacji, wyrażenie sprawdza kod statusu operacji zapisany w zmiennej `result`. Sytuacja taka ma miejsce w prezentowanym przykładzie. Operacja `validatePatientData()` może zwrócić wartość 1 lub 0. Dla każdej z tych wartości przewidziany jest blok warunkowy z wyrażeniem logicznym. Zdań warunkowych dla tego typu operacji może być więcej niż dwa.

a)



b)

```
1   public void savePatientTriggered() {
2       int result;
3       result = service.validatePatientData(patientData);
4       if (result == 1) {    // patient data valid [1]
5           service.savePatientData(patientData);
6           // ...
7       } else if (result == 0) {    // patient data invalid [0]
8           view.showIvalidPatientDataMessage(this);
9           // ...
10      }
11  }
```

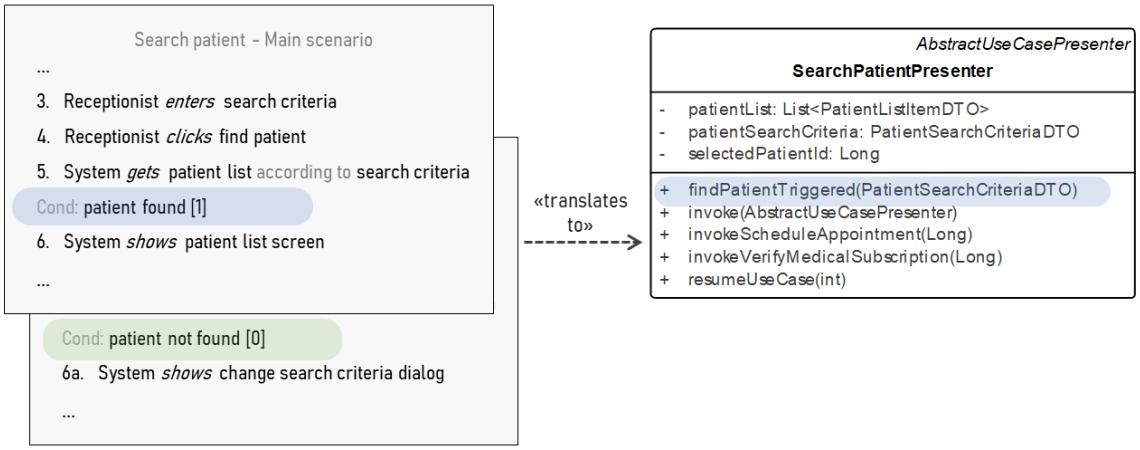
Rysunek 75 – Reguła P13: translacja zdania warunkowego na kod klasy prezentera

Reguła P14. *Dla każdego zdania typu Condition następującego bezpośrednio po zdaniu System-to-ListView utwórz w klasie prezentera instrukcję warunkową* `if-else if`*, w którym wyrażenia logiczne sprawdzają rezultat wywołania metody odpowiadającej zdaniu System-to-ListView. Operacji typu „read" odpowiada jedno wyrażenie logiczne w bloku* `if`*, który jest wykonywany jeśli zwrócona lista obiektów nie jest pusta, co odpowiada wartości 1 ujętej w nawiasy kwadratowe w tekście zdania warunkowego. W przeciwnym razie wykonywany jest blok* `else`*, co odpowiada wartości 0. W przypadku operacji innej niż „read", każdemu zdaniu warunkowemu odpowiada jedno wyrażenie logiczne, które sprawdza czy wartość zmiennej* `result` *jest równa wartości ujętej w nawiasy kwadratowe w tekście odpowiedniego zdania warunkowego. Instrukcję warunkową* `if…else if` *umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system" lub w metodzie* `resumeUseCase()` *jeżeli zdanie*

System-to-ListView jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.

Reguła P14 zilustrowana jest na Rysunek 76. Jest ona niemalże identyczna z regułą P13. Jedyna różnica występuje w przypadku gdy zdanie typu System-to-ListView mapuje się na operację typu „read". Wyrażenie logiczne dla bloku warunkowego `if` jest prawdziwe, jeżeli lista zwrócona jako wynik operacji dziedzinowej nie jest pusta. Sytuacja taka ma miejsce w zaprezentowanym przykładzie. Jeżeli lista pacjentów zwrócona przez operację `readPatientList()` nie jest pusta, wykonywany jest blok warunkowy `if`, implementujący dalszą część scenariusza głównego. W przeciwnym razie wykonywany jest kod scenariusza alternatywnego zawarty w bloku `else`.

a)



b)

```
1  public void findPatientTriggered(PatientSearchCriteriaDTO patientSearchCriteria) {
2      patientList = service.readPatientList(patientSearchCriteria);
3      if (!patientList.isEmpty()) {    // patient found [1]
4          view.showPatientListScreen(this, patientList);
5          // ...
6      } else {    // patient not found [0]
           view.showChangeSearchCriteriaDialog(this);
7          // ...
8      }
9  }
```

Rysunek 76 – Reguła P14: translacja zdania warunkowego na kod klasy prezentera

Reguła P15. *Dla każdego zdania typu Rejoin:*

a) *Utwórz (zgodnie z regułami: P3, P5, P6, P9, P10, P13 i P14) kod odpowiadający sekwencji zdań pomiędzy zdaniem wskazywanym przez zdanie Rejoin (włącznie)*

a zdaniem typu Acor-to-Trigger, System-to-Message, System-to-Confirmation lub zdaniem Invoke wykonywanym przez system;

b) *Jeżeli zdanie Rejoin wskazuje na zdanie wcześniejsze i pomiędzy zdaniem wskazywanym a zdaniem Rejoin nie występuje zdanie typu Acor-to-Trigger, System-to-Message, System-to-Confirmation lub zdanie Invoke wykonywane przez system, utwórz instrukcję pętli* `do-while` *obejmującą kod odpowiadający sekwencji zdań pomiędzy zdaniem wskazywanym przez zdanie Rejoin (włącznie) a zdaniem Rejoin. Wyrażenie logiczne instrukcji* `do-while` *jest tożsame z wyrażeniem logicznym w instrukcji warunkowej wygenerowanej dla ostatniego zdania warunkowego znajdującego się pomiędzy zdaniem wskazywanym przez zdanie Rejoin a zdaniem Rejoin. Jeżeli takie zdanie warunkowe nie występuje, wyrażenie logiczne instrukcji* `do-while` *ma wartość* `true`.

Kod umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system" lub w metodzie `resumeUseCase()` *jeżeli zdanie Final/Postcondition jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.*

Reguła P15 zilustrowana jest na Rysunek 77. Przedstawia on dwa przypadki opisane w treści reguły. W pierwszym przypadku (a), scenariusz alternatywny dla przypadku użycia „Register new patient" zawiera zdanie Rejoin, które pozwala użytkownikowi poprawić dane pacjenta, które system w wyniku walidacji uznał za niepoprawne. Ponieważ pomiędzy zdaniem nr 3 stanowiącym punkt połączenia a najbliższym zdaniem Actor-to-Trigger (zdanie nr 4) nie występują żadne zdania, dla których zastosowanie miałyby reguły P3, P5, P6, P9, P10, P13 i P14, nie ma potrzeby generowania jakiegokolwiek kodu wynikającego ze zdania Rejoin, w miejscu do tego przewidzianym w metodzie `savePatientTriggered()`. W przypadku wprowadzenia niepoprawnych danych pacjenta, system informuje o tym użytkownika poprzez wyświetlenie odpowiedniego komunikatu. Po zamknięciu okna komunikatu, użytkownik wraca do wyświetlonego na początku scenariusza ekranu wprowadzania danych pacjenta, który nie został zamknięty w międzyczasie przez system.

Drugi przypadek (b) przedstawia nieco zmodyfikowane scenariusze dla tego samego przypadku użycia. W tej wersji, w przypadku negatywnej walidacji danych pacjenta, system próbuje je automatycznie naprawić (zdanie nr 6a). Następnie, za sprawą zdania Rejoin, walidacja danych jest powtarzana aż do skutku. Taka konstrukcja scenariusza odpowiada części b) reguły P15. W efekcie generowana jest petla `do-while` obejmująca całą instrukcję warunkową `if-else`

odpowiadającą zdaniu warunkowemu oraz zdaniom po nim następującym. Warunkiem przerwania pętli `do-while` jest pomyślna walidacja danych (`result == 0`).

Rysunek 77 – Reguła P15: translacja zdania Rejoin na kod klasy prezentera

Reguła P16. *Dla każdego zdania typu Final/Postcondition utwórz w klasie prezentera kod, który:*

a) *ustawia rezultat wykonania przypadku użycia poprzez wywołanie metody* `setUseCaseResult()`*; parametr wywołania metody zależy od słowa kluczowego w zdaniu Final/PostCondition i przyjmuje wartość 1 dla „success” oraz 0 dla „failure”;*

b) *finalizuje wykonanie przypadku użycia poprzez wywołanie metody* `finalizeUseCase()`.

Kod umieść w metodzie odpowiadającej zdaniu, które ostatnio zmieniło stan dialogu na „system” lub w metodzie `resumeUseCase()` *jeżeli zdanie Final/Postcondition jest bezpośrednio poprzedzone zdaniem Invoke wykonywanym przez system.*

Reguła P16 zilustrowana jest na Rysunek 78. Każde zdanie Final/Postcondition przekłada się na kod kończący wykonywanie logiki przewidzianej dla danego przypadku użycia. Po pierwsze, rezultat wykonania zapisywany jest w atrybucie `useCaseResult` poprzez wywołanie metody `setUseCaseResult()`, która zmienia domyślną wartość tego atrybutu (-1) na wartość, która zależy od zdania Final/Postcondition. W przedstawionym przykładzie jest to wartość 0, co zgodnie z regułą oznacza, że cel przypadku użycia nie został osiągnięty (failure). Po drugie, sterowanie musi wrócić do miejsca wywołania przypadku użycia, aby możliwe było wywołanie innego przypadku lub ponowne wywołanie tego samego przypadku. Odpowiada za to metoda `finalizeUseCase()`, która – jak wyjaśniono w rozdziale 5.1 – zwraca sterowanie do prezentera wywołującego przypadku użycia pod warunkiem, że użytkownik zakończył wszelkie interakcje poprzez zamknięcie ekranów uaktywnionych w ramach wykonania danego przypadku użycia.

Rysunek 78 – Reguła P16: translacja zdania Final/Postcondition na kod prezentera

5.5 Reguły translacji do warstwy widoku

Reguła V1. *Dla każdego stwierdzenia dziedzinowego należącego do pojęcia typu Screen utwórz operację w interfejsie* `IView` *oraz w klasie* `ViewImpl`*. Nazwa operacji odpowiada prostej frazie czasownikowej zawartej w stwierdzeniu dziedzinowym, przy czym, jeżeli czasownik mapuje się na operację standardową przewidzianą dla pojęcia typu Screen, zamiast czasownika używana jest nazwa operacji standardowej.*

Jeżeli stwierdzenie dziedzinowe mapuje się na akcję typu „show", to:

a) *utwórz parametr wywołania operacji, którego typem jest klasa prezentera odpowiadająca przypadkowi użycia, którego scenariusz odnosi się do danego pojęcia typu Screen;*

b) *jeżeli dane pojęcie typu Screen jest powiązane relacją z pojęciem typu Simple View lub List View i relacja ta wskazuje na pojęcie typu Screen lub jest dwukierunkowa, utwórz parametr wywołania operacji, którego typem jest klasa DTO odpowiadająca pojęciu Simple View (patrz: reguła G8) lub w przypadku pojęcia List View –* `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G9);*

c) *utwórz w klasie* `ViewImpl` *kod operacji, który wyświetla ekran odpowiadający pojęciu Screen (patrz: reguła G3).*

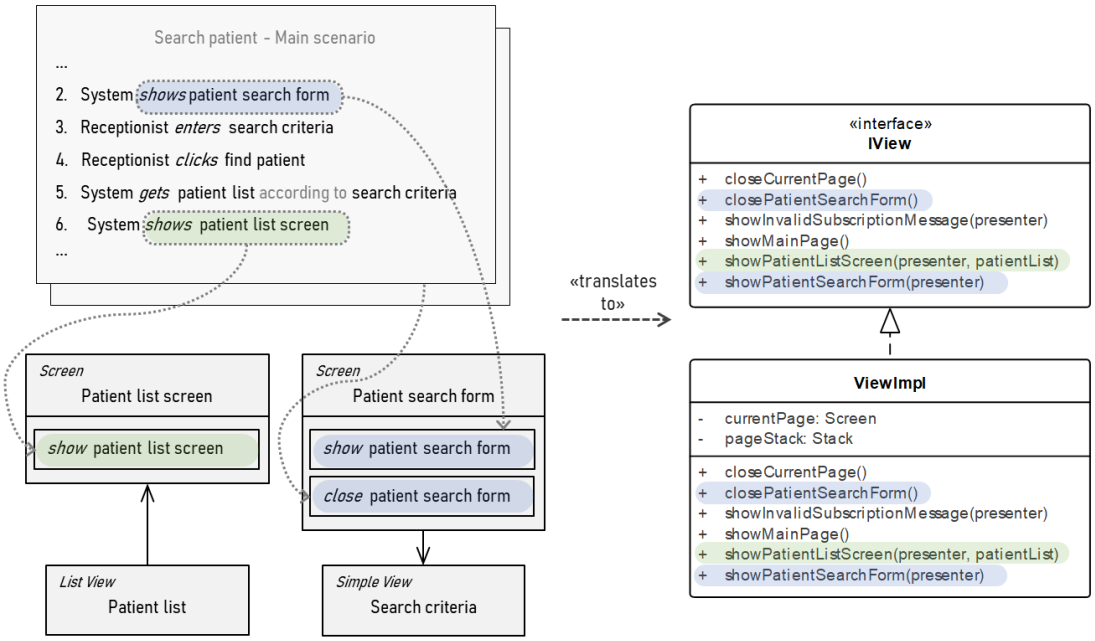
Jeżeli stwierdzenie dziedzinowe mapuje się na akcję typu „refresh", to:

a) *jeżeli dane pojęcie typu Screen jest powiązane relacją z pojęciem typu Simple View lub List View i relacja ta wskazuje na pojęcie typu Screen lub jest dwukierunkowa, utwórz parameter wywołania operacji, którego typem jest klasa DTO odpowiadająca pojęciu Simple View (patrz: reguła G8) lub w przypadku pojęcia List View –* `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G9);*

b) *utwórz w klasie* `ViewImpl` *kod operacji, który odświeża dane prezentowane na ekranie odpowiadającym pojęciu Screen (patrz: reguła G3), na podstawie przekazanego parametru.*

Jeżeli stwierdzenie dziedzinowe mapuje się na akcję typu „close", utwórz w klasie ViewImpl kod operacji, który zamyka ekran odpowiadający pojęciu Screen.

a)



b)

```
1  public void showPatientListScreen(SearchPatientPresenter presenter, List<PatientListItemDTO> patientList) {
2      if (currentPage != null)
3          pageStack.push(currentPage);
4      PatientListScreen page = new PatientListScreen();
5      page.init(presenter, patientList);
6  }
7
8  public void showPatientSearchForm(SearchPatientPresenter presenter) {
9      if (currentPage != null)
10         pageStack.push(currentPage);
11     PatientSearchForm page = new PatientSearchForm();
12     page.init(presenter);
13 }
14
15 public void closePatientSearchForm() {
16     if (currentPage instanceof PatientSearchForm)
17         closeCurrentPage();
18     else
19         for (int i = pageStack.size()-1; i >= 0; i--) {
20             if (pageStack.get(i) instanceof PatientSearchForm) {
21                 pageStack.remove(i);
22                 break;
23             }
24         }
25 }
```

Rysunek 79 – Reguła V1: translacja stwierdzeń dziedzinowych pojęcia typu Screen na operacje interfejsu IView

Reguła V1 zilustrowana jest na Rysunek 79. Przedstawia on przykłady trzech operacji w interfejsie `IView`, które odpowiadają stwierdzeniom dziedzinowym dwóch pojęć typu

Screen. W obu pojęciach występują stwierdzenia dziedzinowe, które mapują się na operacje typu ”show”: `showPatientSearchForm()` oraz `showPatientListScreen()`. Parametrem wejściowym każdej z nich jest obiekt klasy `SearchPatientPresenter` – prezenter odpowiadający przypadkowi użycia „Search patient”, którego scenariusze odnoszą się do odpowiednich stwierdzeń dziedzinowych (w zdaniu nr 2 i 6). Operacja `showPatientListScreen()` ma dodatkowy parametr wejściowy w postaci listy pacjentów do wyświetlenia. Wynika to z faktu, że pojęcie ekranowe „Patient list screen” jest celem relacji łączącej go z widokiem danych „Patient list”.

Pojęcie ekranowe „Patient search form” zawiera również stwierdzenie dziedzinowe, które mapuje się na operację typu „close”: `closePatientSearchForm()`. Tego typu operacje nie definiują żadnych parametrów wejściowych.

Kod implementujący wszystkie operacje interfejsu `IView` generowany jest w klasie `ViewImpl`. W przyjętym środowisku translacyjnym, klasa ta, oprócz realizacji operacji interfejsu `IView`, przechowuje również stan widoku w postaci stosu aktualnie wyświetlanych ekranów aplikacji. Kod operacji `show…()` przed wyświetleniem ekranu, odkłada na stos ekran wyświetlany do tej pory (`currentPage`), aby do niego wrócić po zamknięciu ekranu, którego dotyczy operacja. Kod metody `close…()`, z kolei, zamyka aktualnie prezentowany ekran i przywraca ekran ostatnio odłożony na stos. Należy podkreślić, że w konkretnym środowisku translacyjnym, używającym konkretnej technologii w warstwie GUI, może nie być potrzeby przechowywania stanu widoku w postaci stosu.

Reguła V2. *Dla każdego stwierdzenia dziedzinowego należącego do pojęcia typu Message lub Confirmation utwórz operację w interfejsie* `IView` *oraz w klasie* `ViewImpl`. *Nazwa operacji odpowiada prostej frazie czasownikowej zawartej w stwierdzeniu dziedzinowym, przy czym, jeżeli czasownik mapuje się na operację standardową przewidzianą dla pojęcia typu Message lub Screen, zamiast czasownika używana jest nazwa operacji standardowej.*

Jeżeli stwierdzenie dziedzinowe mapuje się na akcję typu „show”, utwórz w klasie `ViewImpl` *kod operacji, który wyświetla okno komunikatu odpowiadające pojęciu Message lub Confirmation (patrz: reguła G4).*

Reguła V2 jest analogiczna względem reguły V1, przy czym dotyczy operacji na komunikatach (Message) i oknach dialogowych (Confirmation). Zasada generowania operacji w interfejsie

`IView` jest niemal identyczna jak w przypadku pojęć typu Screen, z tą różnicą, że jedyną operacją standardową w przypadku komunikatów jest „show". Okno komunikatu jest zamykane automatycznie w momencie, kiedy użytkownik użyje jednego z wyzwalaczy zdefiniowanych dla komunikatu, np. „OK", „Yes", „No", itp. Stąd, obiekty reprezentujące komunikaty nie są uwzględniane w stosie ekranów. Ponadto, komunikaty nie prezentują danych dziedzinowych, przez co nie ma potrzeby przekazywania obiektów DTO jako parametry wywołania operacji `show…()`.

Reguła V3. *Dla każdego pojęcia SimpleView powiązanego z pojęciem typu Screen utwórz w klasie odpowiadającej pojęciu typu Screen kontrolki ekranowe oraz kod inicjujący w metodzie* `init()` *odpowiadające każdemu atrybutowi wskazywanemu przez pojęcie Simple View. Utwórz również element grupujący kontrolki wygenerowane dla danego pojęcia Simple View. Nazwa elementu grupującego widoczna na ekranie jest nazwą pojęcia Simple View.*

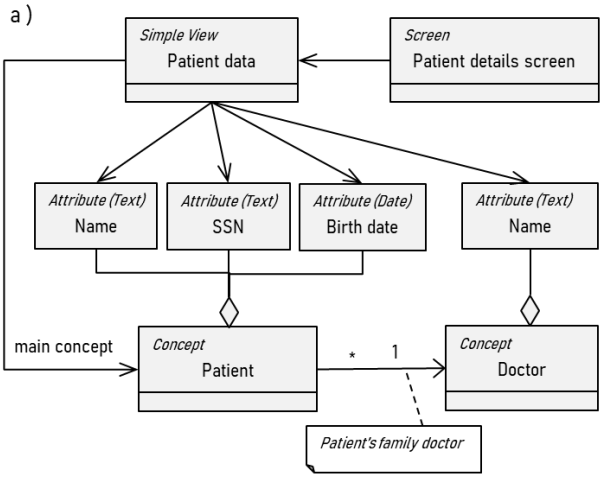
Jeżeli atrybut należy do pojęcia „main concept" (pojęcie Concept powiązane z pojęciem SimpleView relacją typu „main concept") lub pojęcia Concept bezpośrednio powiązanego z pojęciem „main concept" a krotność po stronie tego pierwszego jest równa 1, typ tworzonej kontrolki jest determinowany przez typ atrybutu według poniższych reguł mapowania:

- *Text, Number, Floating point number → pole tekstowe jednowierszowe,*
- *Description → pole tekstowe wielowierszowe,*
- *Secret text → pole tekstowe jednowierszowe z ukrytym tekstem,*
- *True or false → pole typu checkbox,*
- *Date → kontrolka daty,*
- *Time → kontrolka czasu.*

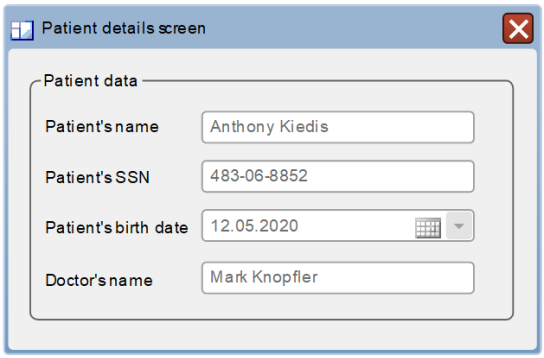
Każda kontrolka opisana jest etykietą ekranową, której nazwa prezentowana na ekranie odpowiada nazwie atrybutu poprzedzonej nazwą pojęcia typu Concept, do którego należy dany atrybut.

Jeżeli atrybut należy do pojęcia Concept bezpośrednio powiązanego z pojęciem „main concept" a krotność po stronie tego pierwszego jest większa niż 1, atrybutowi odpowiada kolumna tabeli stanowiącej listę elementów. Tekst w nagłówku kolumny zawiera nazwę atrybutu poprzedzony nazwą pojęcia typu Concept, do którego należy.

a)



b)



c)

```
1  public class PatientDetailsScreen extends Screen {
2      // ...
3      private Label patientsSsnLabel;
4      private TextField patientsSsnTextField;
5      private Label patientsBirthDateLabel;
6      private DatePicker patientsBirthDateDatePicker;
7      // ...
8      public void init(ShowPatientDetailsPresenter presenter, PatientDetailsDTO patientDetails) {
9          // ...
10         patientsSsnLabel = new Label(”Patient’s name”);
11         patientsSsnLabel.addLayoutData(layout);
12         this.add(patientsSsnLabel);
13         patientsSsnTextField = new TextField();
14         patientsSsnTextField.setEditable(false);
15         patientsSsnTextField.addLayoutData(layout);
16         this.add(patientsSsnTextField);
17         // ...
18     }
19     // ...
20 }
```

Rysunek 80 – Reguła V3: translacja pojęcia typu Simple View na kod implementujący kontrolki ekranowe w klasie odpowiadającej pojęciu typu Screen

Reguła V3 zilustrowana jest na Rysunek 80. Jest to jedna z podstawowych reguł w warstwie widoku, gdyż dotyczy elementów ekranowych, za pomocą których możliwa jest interakcja użytkownika z systemem. W przedstawionym przykładzie pojęcie Patient data definiuje zakres danych prezentowanych na ekranie Patient details screen poprzez wskazanie wybranych atrybutów pojęć dziedzinowych: Patient (które jest pojęciem głównym dla widoku danych) oraz Doctor. Każdy wskazywany atrybut przekłada się na zestaw kontrolek ekranowych: etykietę oraz pole danych zależne od typu atrybutu. W przykładzie występują trzy atrybuty typu Text, którym odpowiadają pola tekstowe (TextField) oraz jeden atrybut typu Date, któremu odpowiada pole daty (DatePicker). Wszystkie kontrolki odpowiadające jednemu widokowi danych umieszczane są na ekranie w obrębie element grupującego odpowiadającego temu

widokowi. Ma to na celu poprawę czytelności w przypadku ekranów prezentujących wiele widoków danych.

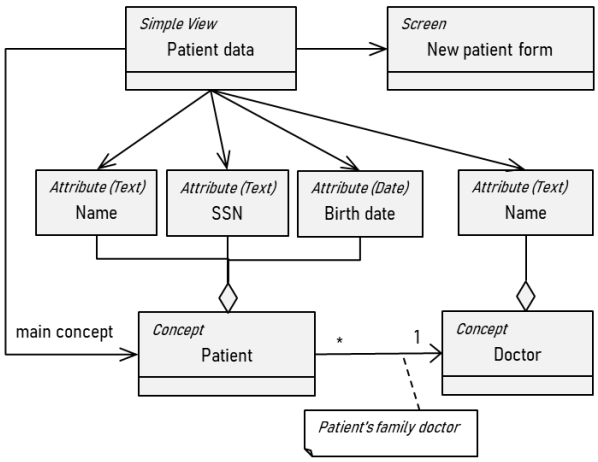
W przypadku użytego tutaj środowiska docelowego, kontrolki implementowane są jako atrybuty klasy ekranowej oraz kod w metodzie `init()`, który jest wykonywany podczas wyświetlania ekranu użytkownikowi. Kod ten inicjalizuje każdą kontrolkę poprzez ustawienie niezbędnych właściwości (zależnie od typu kontrolki) oraz umieszczenie jej w odpowiednim miejscu prezentowanego okna, zgodnie z przyjętym rozmieszczeniem elementów ekranowych (`layout`). Obecna wersja języka RSL nie daje możliwości określania układu ekranów aplikacji. Również reguły translacji nie podają ścisłych szczegółów z tym związanych. Decyzja co do rozmieszczenia elementów ekranowych powinna być, zatem, podjęta podczas implementacji reguł translacji i znaleźć odzwierciedlanie w wygenerowanym kodzie.

W opisywanej regule, szczególną uwagę należy zwrócić na krotności pomiędzy pojęciami dziedzinowymi typu Concept, do których należą atrybuty wskazywane są przez widok danych. W przytoczonym przykładzie, pojęcie główne widoku danych (Patient) jest powiązane z pojęciem Doctor z krotnością równą 1 – każdy pacjent ma przypisanego jednego lekarza rodzinnego. Przekłada się to na pojedyncze pole tekstowe prezentujące nazwisko tego lekarza. W przypadku krotności większej niż 1, zamiast pojedynczego pola musiałaby wystąpić lista prezentująca wszystkich lekarzy przypisanych do pacjenta.

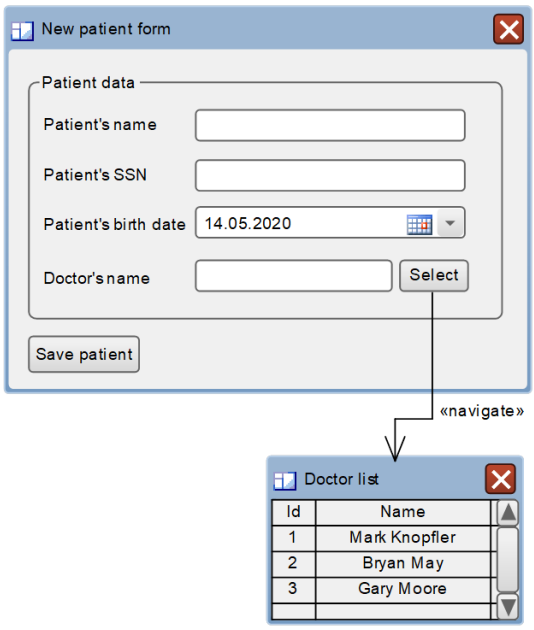
Reguła V4. *Jeżeli pojęcie Simple View powiązane jest z pojęciem Screen relacją typu „Present", dla wszystkich kontrolek odpowiadających atrybutom wskazywanym przez pojęcie Simple View (patrz: reguła V3) wyłącz możliwość edycji wartości prezentowanych przez te kontrolki.*

W przypadku relacji typu „Input" lub „Modify" włącz możliwość edycji wartości prezentowanych przez kontrolki. Ponadto, w przypadku kontrolek odpowiadających atrybutom pojęcia Concept bezpośrednio powiązanego z pojęciem „main concept", utwórz kod listy wyboru istniejącej w systemie instancji obiektu odpowiadającego danemu pojęciu Concept (patrz: reguła M2). Kod obejmuje kontrolkę wyzwalającą wyświetlenie listy oraz obsługę zdarzenia w metodzie `actionPerformed()`. *Kod obsługi zdarzenia sprowadza się do wywołania metody prezentera, która zwraca listę pobranych z modelu obiektów DTO stanowiących element listy wyboru.*

a)



b)



c)

```
1  public class NewPatientForm extends Screen {
2      // ...
3      private Label doctorsNameLabel;
4      private TextField doctorsNameTextField;
5      private Button selectDoctorsNameButton;
6      // ...
7
8      public void init(RegisterNewPatientPresenter presenter) {
9          // ...
10         doctorsNameTextField = new TextField();
11         doctorsNameTextField.addLayoutData(layout);
12         doctorsNameTextField.setEditable(true);
13         this.add(doctorsNameTextField);
15         selectDoctorsNameButton = new Button(”Select”);
16         selectDoctorsNameButton.addLayoutData(layout);
17         this.add(selectDoctorsNameButton);
18         // ...
19     }
20
21     public actionPerformed(ActionEvent e) {
22         // ...
23         if (e.getActionCommand().equals(”selectDoctorsNameButton”)) {
24             List<DoctorDTO> list = presenter.getDoctorList();
25             this.add(new DoctorSelectionDialog(list));
26         }
27         // ...
28     }
29     // ...
30 }
```

Rysunek 81 – Reguła V4: translacja pojęcia typu Simple View na kod implementujący kontrolki ekranowe w klasie odpowiadającej pojęciu typu Screen

Reguła V4 zilustrowana jest na Rysunek 81. Przedstawiony tu ekran New patient form, podobnie jak ekran Patient details screen przedstawiony w poprzednim przykładzie (Rysunek 80), powiązany jest z tym samym widokiem danych – Patient details. Inny jest natomiast kierunek relacji między widokiem danych a każdym z ekranów. W poprzednim przykładzie

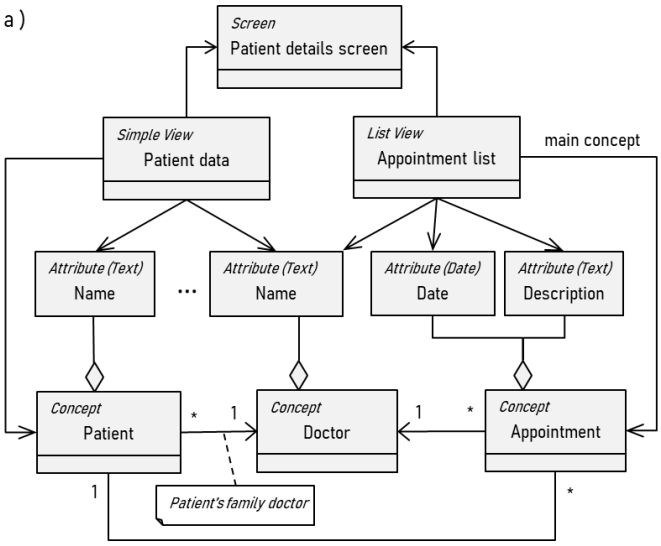
mieliśmy do czynienia z ekranem typu „Present”, dlatego wszystkie pola danych były nieaktywne – nie umożliwiały edycji prezentowanych danych. New patient form jest, natomiast, ekranem typu „Input” – kod w metodzie `init()` włącza możliwość edycji kontrolek (wiersz 12 w przedstawionym kodzie klasy `NewPatientForm`). Ponadto, w przypadku pola „Doctor’s name”, które odpowiada atrybutowi pojęcia Doctor, możliwe jest wprowadzenie nazwiska lekarza z listy wyświetlanej po naciśnięciu przycisku „Select”. Kod obsługi zdarzenia dla tego przycisku znajduje się w metodzie `actionPerformed()`. Zgodnie z regułą zawiera on wywołanie odpowiedniej metody prezentera (`getDoctorList()`) w celu pobrania listy lekarzy a następnie wyświetleniu tej listy użytkownikowi w postaci okienka dialogowego (`DoctorSelectionDialog`).

Reguła V5. *Dla każdego pojęcia List View powiązanego z pojęciem typu Screen utwórz w klasie odpowiadającej pojęciu typu Screen kontrolki ekranowe wraz z inicjującym je kodem w metodzie* `init()` *implementujące tabelę odpowiadającą pojęciu List View. Tabelę umieść w elemencie grupującym kontrolki, którego nazwa widoczna na ekranie jest nazwą pojęcia List View.*

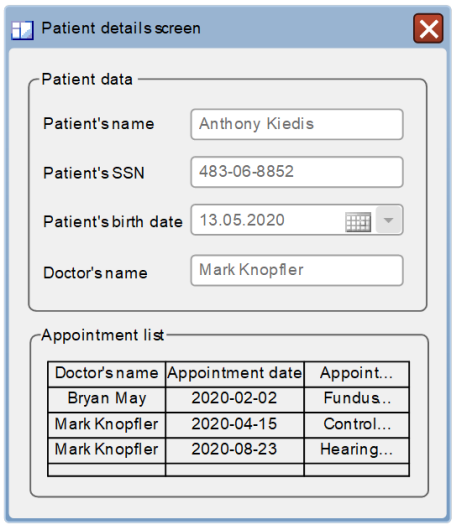
Dla każdego atrybutu należącego do pojęcia „main concept” (pojęcie Concept powiązane z pojęciem SimpleView relacją typu „main concept”) lub innego pojęcia Concept bezpośrednio powiązanego z pojęciem „main concept”, utwórz kolumnę tabeli, której nazwa odpowiada nazwie atrybutu poprzedzonej nazwą pojęcia, do którego ten atrybut należy.

Dla każdego typu atrybutu konwertuj wartość atrybutu na tekst prezentowany w kolumnie tabeli odpowiadającej temu atrybutowi. W przypadku atrybutu należącego do pojęcia Concept bezpośrednio powiązanego z pojęciem „main concept” z krotnością po stronie tego pierwszego większą niż 1, wartości atrybutu dla poszczególnych instancji pojęcia Concept konwertuj na tekst i połącz używając znaku „,” jako separatora.

a)



b)



c)

```
1  public class PatientDetailsScreen extends Screen {
2      // ...
3      private Table appointmentListTable;
4      private AppointmentListTableModel appointmentListTableModel;
5      // ...
6      public void init(ShowPatientDetailsPresenter presenter, PatientDataDTO patientData,
7                       List<AppointmentListItemDTO> appointmentList) {
8          // ...
9          appointmentListTableModel = new AppointmentListTableModel();
10         appointmentListTable = new Table(appointmentListTableModel);
11         appointmentListTable.setSelectionEnabled(true);
12         appointmentListTable.addLayoutData(layout);
13         this.add(appointmentListTable);
14         // ...
15     }
16     // ...
17 }
18
19 public class AppointmentListTableModel extends AbstractTableModel {
20     // ...
21     private List<AppointmentListItemDTO> rows = new ArrayList<AppointmentListItemDTO>();
22     // ...
23     public void addRows(List<AppointmentListItemDTO> rows) {
24         this.rows = rows;
25         // ...
       }
26     // ...
27 }
```

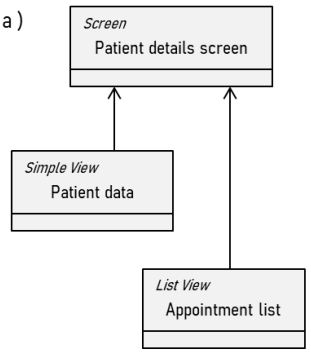
Rysunek 82 – Reguła V5: translacja pojęcia typu List View na kod implementujący kontrolki ekranowe w klasie odpowiadającej pojęciu typu Screen

Reguła V5 zilustrowana jest na Rysunek 82. Jest ona analogiczna do reguły V3 ale dotyczy pojęć typu List View. W przypadku pojęć tego typu, wskazywanym atrybutom odpowiadają

kolumny tabeli, w których prezentowanych jest wiele obiektów bzinesowych odpowiadających pojęciu głównemu widoku danych („main concept”). W zależności od środowiska docelowego, może to wymagać dodatkowej klasy implementującej model danych tabeli, jak w przedstawionym przykładzie stanowiącym rozszerzenie przykładu ilustrującego regułę V3 (Rysunek 80). Oprócz atrybutu `appointmentListTable` reprezentującego element ekranowy typu tabela (`Table`), utworzona została klasa `AppointmentListTableModel`, której zadaniem jest m.in. mapowanie listy obiektów biznesowych na wiersze tabeli.

Reguła V6. *Dla każdego pojęcia Simple View oraz List View powiązanego z pojęciem typu Screen utwórz w klasie odpowiadającej pojęciu Screen prywatny atrybut klasy wskazujący na obiekt transferu danych. Nazwa atrybutu jest nazwą pojęcia zapisaną w notacji Camel case. W przypadku pojęcia Simple View, typem atrybutu jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G8). Jeżeli dopełnienie zdania odnosi się do pojęcia ListView, typem atrybutu jest* `java.util.List<T>`*, gdzie parametrem typu listy* `T` *jest klasa DTO odpowiadająca temu pojęciu (patrz: reguła G9). Jeżeli parametrem wywołania metody* `init()` *jest obiekt tego samego typu co utworzony atrybut, zainicjuj atrybut wartością przekazaną w parametrze. W przeciwnym razie zainicjuj atrybut nowo utworzonym obiektem.*

a)



b)

```
1  public class PatientDetailsScreen extends Screen {
2      // ...
3      private PatientDataDTO patientData;
4      private List<AppointmentListItemDTO> appointmentList;
5      // ...
6      public void init(ShowPatientDetailsPresenter presenter,
7                       PatientDataDTO patientData,
8                       List<AppointmentListItemDTO> appointmentList) {
9          // ...
10         this.patientData = patientData;
11         this.appointmentList = appointmentList;
17         // ...
18     }
19     // ...
20 }
```

Rysunek 83 – Reguła V6: translacja pojęcia typu Simple View oraz List View na atrybuty typu DTO

Reguła V6 zilustrowana jest na Rysunek 83, który przedstawia ekran powiązany z dwoma widokami danych: jeden typu Simple View (Patient data) a drugi typu List View (Appointment

list). Zgodnie z regułą, takiej konstrukcji źródłowej odpowiadają dwa atrybuty klasy implementującej ekran Patient details screen. Pierwszy z nich jest pojedynczym obiektem DTO odpowiadającym pojęciu Simple View, drugi – listą obiektów DTO. Ponieważ pomiędzy widokami danych a ekranem występuje relacja typu „Present", oba atrybuty są inicjowane wartościami odpowiednich parametrów wywołania operacji `init()`. Dane przekazane w ten sposób są następnie wyświetlane w odpowiednich kontrolkach ekranowych. Mapowanie wartości atrybutów obiektów DTO na zawartość kontrolek odbywa się w ramach metody `populateControls()` (patrz: reguła V7).

Reguła V7. *Dla każdego pojęcia typu Screen reprezentującego ekran:*

a) *Jeżeli jest to ekran typu „Present" oraz „Modify" – utwórz w odpowiadającej mu klasie prywatną metodę o nazwie* `populateControls()`. *Dla każdego atrybutu, dla którego została utworzona kontrolka ekranowa (patrz: reguła V3) utwórz w tej metodzie kod ustawiający wartość prezentowaną przez tę kontrolkę na podstawie pola obiektu DTO odpowiadającego danemu atrybutowi (patrz: reguły G7 – G9). Na końcu metody* `init()` *umieść wywołanie metody* `populateControls()`.

b) *Jeżeli jest to ekran typu „Input" oraz „Modify" – utwórz w odpowiadającej mu klasie prywatną metodę o nazwie* `populateDTOs()`. *Dla każdego atrybutu, dla którego została utworzona kontrolka ekranowa utwórz w tej metodzie kod przypisujący polu obiektu DTO odpowiadającemu danemu atrybutowi wartość prezentowaną przez tę kontrolkę.*

Reguła V7 zilustrowana jest na Rysunek 84, który przedstawia kod dwóch klas ekranowych: `PatientDetailsScreen` (a) oraz `NewPatientForm` (b) i jest kontynuacją przykładów ilustrujących wcześniejsze reguły.

W pierwszym przypadku (a) mamy do czynienia z ekranem typu „Present" a zatem, zgodnie z regułą V7, tworzona jest metoda `populateControls()` mapująca wartości atrybutów obiektów DTO na zawartość odpowiadających im kontrolek. W przypadku atrybutów tekstowych jest to zwykłe przypisanie wartości (patrz: wiersz 12), natomiast w przypadku innych typów, np. Date lub int, należy wykonać przekształcenie typów (patrz: wiersze 14-15). W przypadku tabeli, za poprawne mapowanie atrybutów na odpowiednie kolumny w kolejnych

wierszach, odpowiada model tabeli. W naszym przykładzie jest to `AppointmentListTableModel` (patrz: wiersz 19).

W drugim przykładzie (b) mamy do czynienia z ekranem typu „Input", co skutkuje utworzeniem metody `populateDTOs()`, której działanie jest odwrotne do metody `populateControls()` – zawartość kontrolek mapowana jest na wartości odpowiednich atrybutów obiektów DTO, które następnie przekazywane są do prezentera (patrz: reguła V8).

a)
```
1  public class PatientDetailsScreen extends Screen {
2      // ...
3      public void init(ShowPatientDetailsPresenter presenter, PatientDetailsDTO patientDetails,
4                       List<AppointmentListItemDTO> appointmentList) {
5          // ...
6          populateControls();
7      }
8      // ...
9      private void populateControls() {
10         // ...
11         if (patientDetails.getPatientSSN() != null)
12             patientsSsnTextField.setText(patientDetails.getPatientSSN());
13         if (patientDetails.getBirthDate() != null) {
14             SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
15             patientsBirthDateDatePicker.setText(dateFormat.format(patientDetails.getBirthDate()));
16         }
17         // ...
18         if (appointmentList != null)
19             appointmentListTableModel.addRows(appointmentList);
20     }
21     // ...
22 }
```

b)
```
1  public class NewPatientForm extends Screen {
2      // ...
3      public void populateDTOs() {
4          // ...
5          if (patientSsnTextField.getText() != null)
6              patientDetails.setPatientSsn(patientSsnTextField.getText());
7          if (patientBirthDateDatePicker.getText() != null) {
8              SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
9              Date date = dateFormat.parse(patientBirthDateDatePicker.getText());
10             patientDetails.setPatientBirthDate(date);
11         }
12         // ...
13     }
14     // ...
15 }
```

Rysunek 84 – Reguła V7: generacja metod populateControls() oraz populateDTOs()

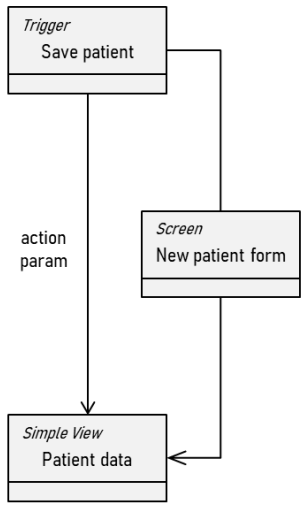
Reguła V8. *Dla każdego pojęcia typu Trigger powiązanego z pojęciem typu Screen utwórz w klasie odpowiadającej pojęciu typu Screen:*

a) kontrolkę ekranową reprezentującą przycisk wraz z inicjującym ją kodem w metodzie `init()`*; nazwa przycisku wyświetlana na ekranie odpowiada nazwie pojęcia typu Trigger;*

b) kod w metodzie `actionPerformed()`*, który (1) w przypadku ekranu typu „Input" oraz „Modify" wywołuje metodę* `populateDTOs()` *oraz (2) wywołuje na obiekcie prezentera (atrybut* `presenter`*) metodę odpowiadającą danemu wyzwalaczowi (patrz: reguła P4). Jako parametry wywołania metody prezentera przekaż obiekty DTO odpowiadające pojęciom typu Data View powiązanym z pojęciem Trigger relacją typu „action param" (patrz: reguła V6).*

Reguła V8 zilustrowana jest na Rysunek 85. Jest to kontynuacja przykładu ilustrującego regułę V4 oraz V7. Scenariusz przypadku użycia „Register new patient" przewiduje, że po wprowadzeniu danych pacjenta, użytkownik wybiera przycisk „Save patient" w celu ich zapisania. Przekłada się to na powiązanie pomiędzy pojęciem „New patient form" oraz pojęciem „Save patient" reprezentującym wspomniany przycisk. Dodatkowo, relacja typu „action param" wskazuje dane, których dotyczy akcja wyzwalana przez przycisk (Rysunek 85a). Takiemu konstruktowi odpowiada kod przedstawiony na Rysunek 85b. Zgodnie z regułą tworzony jest atrybut klasy implementujący przycisk (wiersz 3) wraz z kodem ustawiającym właściwości przycisku i umieszczającym go na ekranie (wiersze 8-10) a także kod w metodzie `actionPerformed()`. Ten ostatni mapuje dane wprowadzone przez użytkownika na wartości atrybutów obiektu `patientData` za pomocą metody `populateDTOs()` i w tej postaci przekazuje do prezentera jako parametr wywołania metody `savePatientTriggered()`.

a)



b)

```
1  public class NewPatientForm extends Screen {
2      // ...
3      private Button savePatientButton;
4      private PatientDataDTO patientData;
5      // ...
6      public void init(RegisterNewPatientPresenter presenter) {
7          // ...
8          savePatientButton = new Button(”Save patient”);
9          savePatientButton.addLayoutData(layout);
10         this.add(savePatientButton);
11         // ...
12         patientData = new PatientDataDTO();
13     }
14     // ...
15     public actionPerformed(ActionEvent e) {
16         // ...
17         if (e.getActionCommand().equals(”savePatientButton”)) {
18             populateDTOs();
19             presenter.savePatientTriggered(patientData);
20         }
21         // ...
22     }
23     // ...
24 }
```

Rysunek 85 – Reguła V8: translacja pojęć typu Trigger na kod kontrolek ekranowych

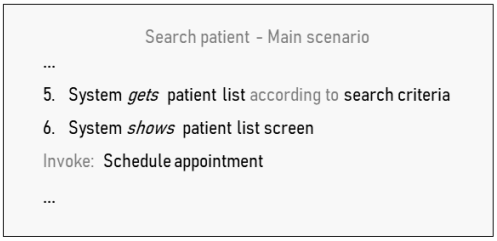
Reguła V9. *Dla każdego zdania typu Invoke, dla którego stan dialogu wskazuje na aktora, utwórz w klasie odpowiadającej pojęciu typu Screen, w kontekście którego występuje zdanie Invoke:*

a) *kontrolkę ekranową reprezentującą przycisk wraz z inicjującym ją kodem w metodzie* `init()`*; nazwa przycisku prezentowana na ekranie odpowiada nazwie wywoływanego przypadku użycia;*

b) *kod w metodzie* `actionPerformed()` *wywołujący na obiekcie prezentera (atrybut* `presenter`*) metodę odpowiadającą danemu zdaniu typu Invoke (patrz: reguła P2). Jeżeli wywoływany przypadek użycia wymaga parametru wejściowego, jako parametr wywołania przekazywana jest wartość identyfikatora obiektu biznesowego odpowiadającego parametrowi wejściowemu wywoływanego przypadku użycia (patrz: reguła P1); Identyfikator ten pobierany jest z obiektu DTO, które odpowiada pojęciu Data View (patrz: reguła V6) powiązanemu relacją typu „main concept” z pojęciem Concept reprezentującym wspomniany obiekt biznesowy; W przypadku listy, wybierany*

jest obiekt DTO odpowiadający elementowi listy, który jest aktualnie zaznaczony na ekranie przez użytkownika.

a)



b)

```
1  public class PatientListScreen extends Screen {
2      // ...
3      private Button scheduleAppointmentButton;
4      private List<PatientListItemDTO> patientList;
5      // ...
6      public void init(SearchPatientPresenter presenter,
7                       List<PatientListItemDTO> patientList) {
8          // ...
9          scheduleAppointmentButton = new Button(”Schedule appointment”);
10         scheduleAppointmentButton.addLayoutData(layout);
11         this.add(scheduleAppointmentButton);
12         // ...
13     }
14     // ...
15     public actionPerformed(ActionEvent e) {
16         // ...
17         if (e.getActionCommand().equals(”scheduleAppointmentButton”)) {
18             long patientId = patientListTableModel.getSelectedRow().getPatientId();
19             presenter.invokeScheduleAppointment(patientId);
20         }
21         // ...
22     }
23     // ...
24 }
```

Rysunek 86 – Reguła V9: translacja zdań typu Invoke na kod kontrolek ekranowych

Reguła V9 zilustrowana jest na Rysunek 86. Jest ona podobna do poprzedniej reguły z tą różnicą, że konstrukcją źródłową jest tutaj zdanie scenariusza oraz jego kontekst a nie model dziedziny. Wynikowy kod jest jednak bardzo zbliżony – dla zdania Invoke tworzony jest kod implementujący przycisk oraz kod obsługi zdarzenia w metodzie `actionPerformed()`, który polega na wywołaniu odpowiedniej metody prezentera w celu uruchomienia dodatkowego przypadku użycia. Prezentowany przykład dotyczy scenariusza głównego przypadku użycia „Search patient”, w którym po wyświetleniu ekranu z listą pacjentów, użytkownik ma możliwość umówienia wizyty dla wybranego pacjenta poprzez użycie przycisku „Schedule

appointment”. Kod obsługi takiego zdarzenia musi więc wywołać metodę `invokeScheduleAppointment()` (wiersz 19). Ponieważ metoda ta wymaga podania identyfikatora wybranego pacjenta, przed wywołaniem metody prezentera, odczytywana jest wartość atrybutu `patientId` dla pacjenta aktualnie wybranego na liście pacjentów (wiersz 18).

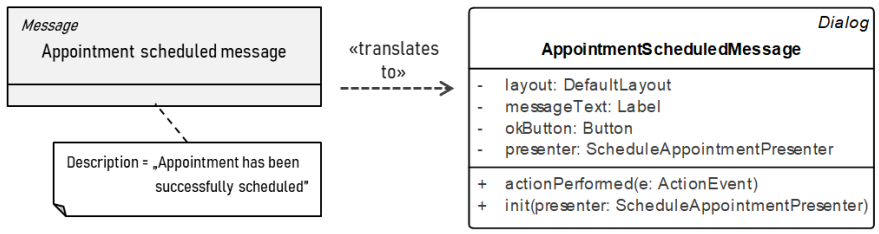
Reguła V10. *Dla każdego pojęcia typu Message oraz Confirmation utwórz w klasie odpowiadającej danemu pojęciu:*

a) *kontrolkę ekranową reprezentującą tekst komunikatu wraz z inicjującym ją kodem w metodzie* `init()`*; tekst komunikatu zawarty jest w opisie pojęcia typu Message;*

b) *w przypadku pojęcia typu Message – kontrolkę ekranową reprezentującą przycisk wraz z inicjującym ją kodem w metodzie* `init()`*, gdzie nazwą przycisku wyświetlaną na ekranie jest „OK” oraz kod w metodzie* `actionPerformed()`*, który (1) zamyka okno komunikatu oraz (2) wywołuje na obiekcie prezentera (atrybut* `presenter`*) metodę informującą prezenter o zamknięciu okna komunikatu (patrz: reguła P7).*

Reguła V10 zilustrowana jest na Rysunek 87. Głównym elementem każdego okna komunikatu (zarówno typu Message jak i Confirmation) jest tekst komunikatu. W modelu dziedziny w języku RSL jest on zawarty w opisie pojęcia. W kodzie klasy odpowiada mu etykieta ekranowa typu Label (wiersze 3, 8-10). Ponadto, każde okno komunikatu typu Message musi udostępniać przycisk „OK” zamykający okno oraz przekazujący sterowanie do obiektu prezentera. Jak w przypadku każdego przycisku jest to realizowane w ramach metody `actionPerformed()` (wiersze 19-22).

a)



b)

```
1  public class AppointmentScheduledMessage extends Dialog {
2      // ...
3      private Label messageText;
4      private Button okButton;
5      // ...
6      public void init(ScheduleAppointmentPresenter presenter) {
7          // ...
8          messageText = new Label(”Appointment has been successfully scheduled”);
9          messageText.addLayoutData(layout);
10         this.add(messageText);
11         okButton = new Button(”OK”);
12         okButton.addLayoutData(layout);
13         this.add(okButton);
14         // ...
15     }
16     // ...
17     public actionPerformed(ActionEvent e) {
18         // ...
19         if (e.getActionCommand().equals(”okButton”)) {
20             userClose();
21             presenter.AppointmentScheduledMessageOKTriggered();
22         }
23         // ...
24     }
25     // ...
26 }
```

Rysunek 87 – Reguła V10: translacja pojęć typu Message oraz Confirmation na kod kontrolek ekranowych

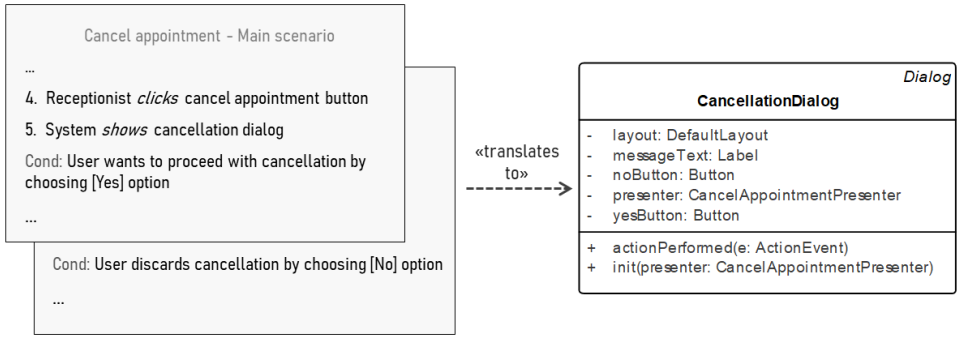
Reguła V11. *Dla każdego pojęcia typu Confirmation, do którego odnoszą się zdania System-to-Confirmation bezpośrednio poprzedzające zdania warunkowe, utwórz w klasie odpowiadającej pojęciu Confirmation:*

a) *kontrolki ekranowe reprezentujące przyciski odpowiadające każdemu zdaniu warunkowemu wraz z inicjującym je kodem w metodzie* `init()`*; nazwy przycisków wyświetlane na ekranie odpowiadają tekstowi ujętemu w nawiasy kwadratowe w treści odpowiedniego zdania warunkowego;*

b) *kod w metodzie* `actionPerformed()` *dla każdego przycisku, który (1) zamyka okno komunikatu oraz (2) wywołuje na obiekcie prezentera (atrybut* `presenter`*)*

metodę informującą prezenter o akcji użytkownika polegającej na wybraniu przez użytkownika określonego przycisku w oknie komunikatu (patrz: reguła P11).

a)



b)

```
1  public class CancellationDialog extends Dialog {
2      // ...
3      private Button yesButton;
4      private Button noButton;
5      // ...
6      public void init(CancelAppointmentPresenter presenter) {
7          // ...
8          messageText = new Label(”Are you sure you want to cancel the appointment?”);
9          // ...
10         yesButton = new Button(”Yes”);
11         yesButton.addLayoutData(layout);
12         this.add(yesButton);
13         noButton = new Button(”No”);
14         noButton.addLayoutData(layout);
15         this.add(noButton);
16         // ...
17     }
18     // ...
19     public actionPerformed(ActionEvent e) {
20         // ...
21         if (e.getActionCommand().equals(”yesButton”)) {
22             userClose();
23             presenter.cancellationDialogYesTriggered();
24         }
25         if (e.getActionCommand().equals(”noButton”)) {
26             userClose();
27             presenter.cancellationDialogNoTriggered();
28         }
29         // ...
30     }
31     // ...
32 }
```

Rysunek 88 – Reguła V11: translacja pojęć typu Confirmation używanych w kontekście zdań warunkowych na kod kontrolek ekranowych

W przypadku komunikatów typu Confirmation, w odróżnieniu od Message, przyciski widoczne w oknie dialogowym zależą od zdań warunkowych następujących bezpośrednio po zdaniu, którego efektem jest wyświetlenie danego komunikatu. W przedstawionym przykładzie mamy dwa takie zdania warunkowe, którym odpowiada kod dwóch przycisków: `yesButton` oraz `noButton`. Zdarzenie na każdym z nich skutkuje wykonaniem odpowiadającego im fragmentu kodu w metodzie `actionPerformed()`, który zamyka okno i wywołuje odpowiednią metodę prezentera: `cancellationDialogYesTriggered()` lub `cancellationDialogNoTriggered()`.

6 Implementacja transformacji RSL to Java

Rozdział ten prezentuje transformację „RSL to Java", która jest przykładową implementacją przedstawionych reguł translacji. Podczas implementacji transformacji, przyjęto następujące założenia projektowe:

a) wygenerowany kod powinien być zgodny ze zdefiniowanymi regułami translacji,

b) transformacja będzie generować kod w języku Java z wykorzystaniem konkretnego środowiska programistycznego zgodnego ze wzorcem MVP,

c) wygenerowany kod będzie na tyle kompletny, aby można było go bezpośrednio skompilować i uruchomić,

d) wygenerowany kod będzie sformatowany zgodnie z dobrymi praktykami dla języka Java (m.in. każda instrukcja w osobnym wierszu, stosowanie wcięć, stosowanie konwencji nazewniczych Camel Case oraz Pascal Case, itp.), aby zapewnić czytelność i łatwość ręcznej modyfikacji,

e) kod transformacji będzie miał przejrzystą strukturę umożliwiającą jej łatwe rozszerzanie oraz zmianę środowiska programistycznego.

Transformacja „RSL to Java" została napisana w języku MOLA i składa się z ponad 150 procedur, które implementują 38 reguł semantycznych oraz dodatkowe szczegóły technologiczne niezbędne do wygenerowania kompletnego i poprawnego kodu dla danego środowiska. Ze względu na złożoność, rozdział ten ogranicza się do opisu ogólnej struktury transformacji oraz wybranych procedur.

Należy zaznaczyć, że kod generowany przez transformację miejscami odbiega nieznacznie od definicji reguł semantycznych przedstawionych w rozdziale 5. Wynika to z zastosowanego konkretnego środowiska docelowego Echo3 – generowany kod powinien być zgodny z wymaganiami oraz dobrymi praktykami dla tego środowiska. Wynika to również z faktu, że język MOLA oraz jego środowisko deweloperskie METAclipse mają charakter prototypowy i obarczone są pewnymi ograniczeniami w zakresie składni języka oraz funkcjonalności i wydajności środowiska. W niektórych przypadkach wymaga to stosowania pewnych obejść i optymalizacji zarówno w kodzie transformacji jak i w kodzie docelowym w języku Java.

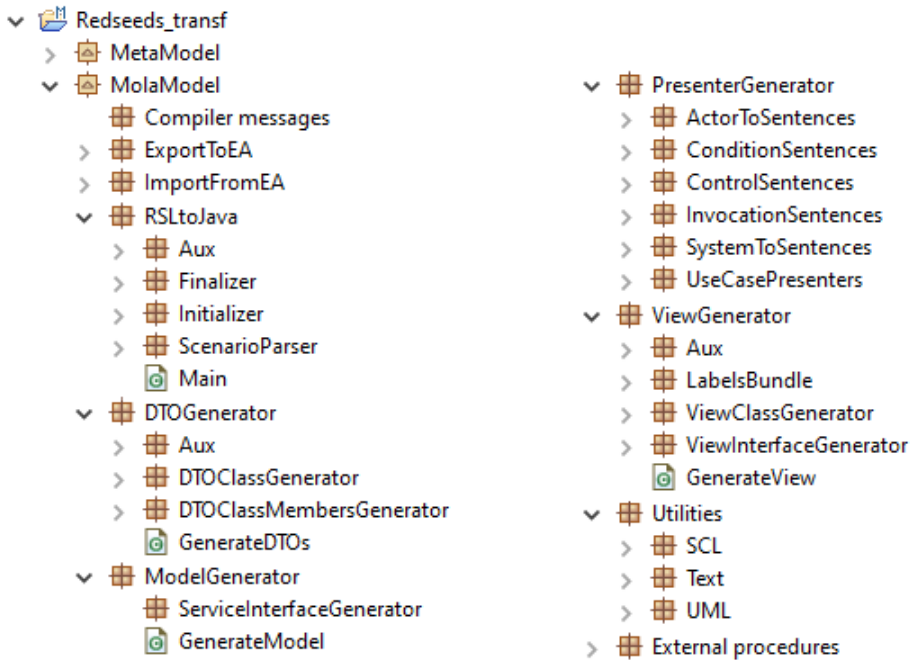
6.1 Wybór środowiska docelowego

Jednym ze środowisk programistycznych umożliwiających tworzenie aplikacji w języku Java, zgodnych ze wzorcem MVP, jest Echo3. Środowisko to przeznaczone jest do tworzenia jednostronicowych aplikacji internetowych (ang. Single Page Application) [163] [164] działających po stronie serwera (ang. server-side). Model programowania w Echo3 oparty jest na komponentach i zdarzeniach (ang. event-driven), co umożliwia tworzenie aplikacji działających w przeglądarce, których zachowanie i możliwości zbliżone są do natywnych aplikacji desktopowych napisanych z użyciem technologii Java Swing czy Eclipse SWT. Dzięki temu Echo3 upraszcza tworzenie aplikacji internetowych z bogatym i wysoce interaktywnym interfejsem użytkownika typu request-response, pozbawionych niedogodności typowych dla aplikacji opartych o statyczne strony HTML i technologie pokrewne, m.in. słabej responsywności wynikającej z konieczność przeładowywania stron czy trudniejsze testowanie i lokalizacja błędów. Podobnie jak wspomniane środowiska desktopowe, Echo3 oferuje zestaw komponentów do tworzenia interaktywnych interfejsów użytkownika, m.in. przyciski, pola tekstowe, okna dialogowe, układy stron, itd.

Wybór platformy Echo3 został dokonany arbitralnie. Niemniej, fakt, że platforma Echo3 umożliwia tworzenie kompletnych aplikacji internetowych z użyciem języka Java, nie pozostał bez wpływu na wybór. Użycie opartego o klasy, obiektowego języka programowania ułatwia generowanie kodu docelowej aplikacji – tym bardziej, że jest on w dużej mierze generowany na podstawie modelu obiektowego w języku UML.

6.2 Przegląd struktury transformacji

Ogólna struktura transformacji RSL to Java przedstawiona jest na Rysunek 89. Cała transformacja, zgodnie z przyjętym założeniem projektowym, została podzielona na kilka pakietów, które mogą być skompilowane niezależnie, tzn. bez potrzeby kompilacji pozostałych pakietów. Pozwala to zoptymalizować czas potrzebny na kompilację całej transformacji przy wprowadzaniu zmian w wybranych pakietach. Co ważniejsze, znacząco ułatwia to wymianę całych komponentów, np. w przypadku konieczności zastosowania innej technologii w warstwie widoku, zmianie sposobu generowania obiektów transferu danych, itp.

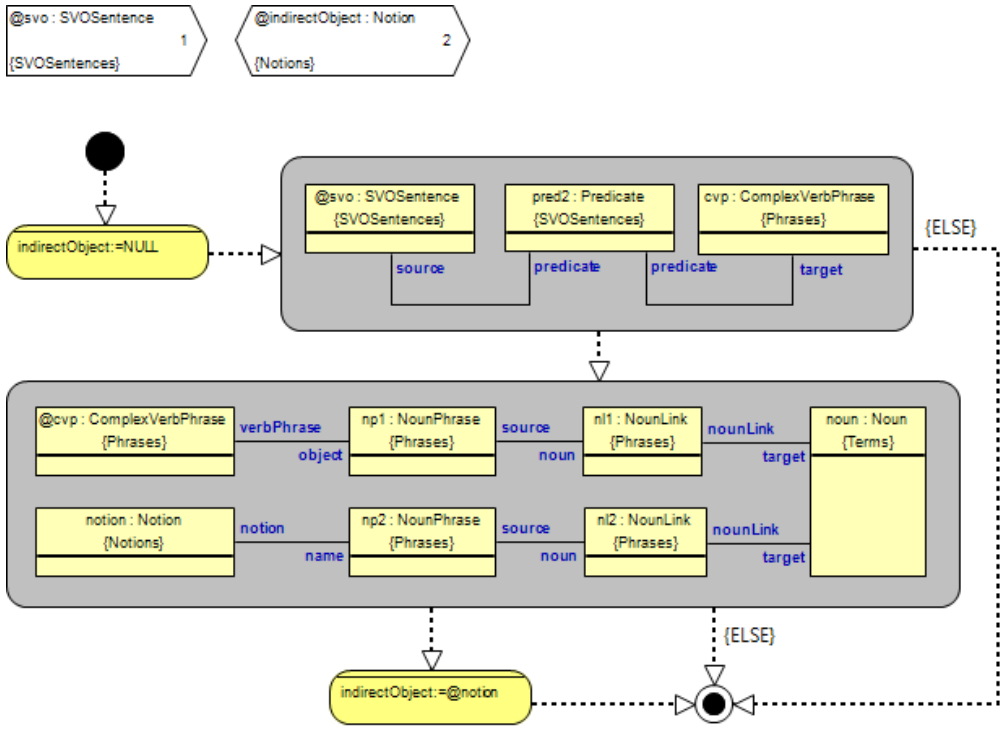


Rysunek 89 – Struktura transformacji RSL to Java

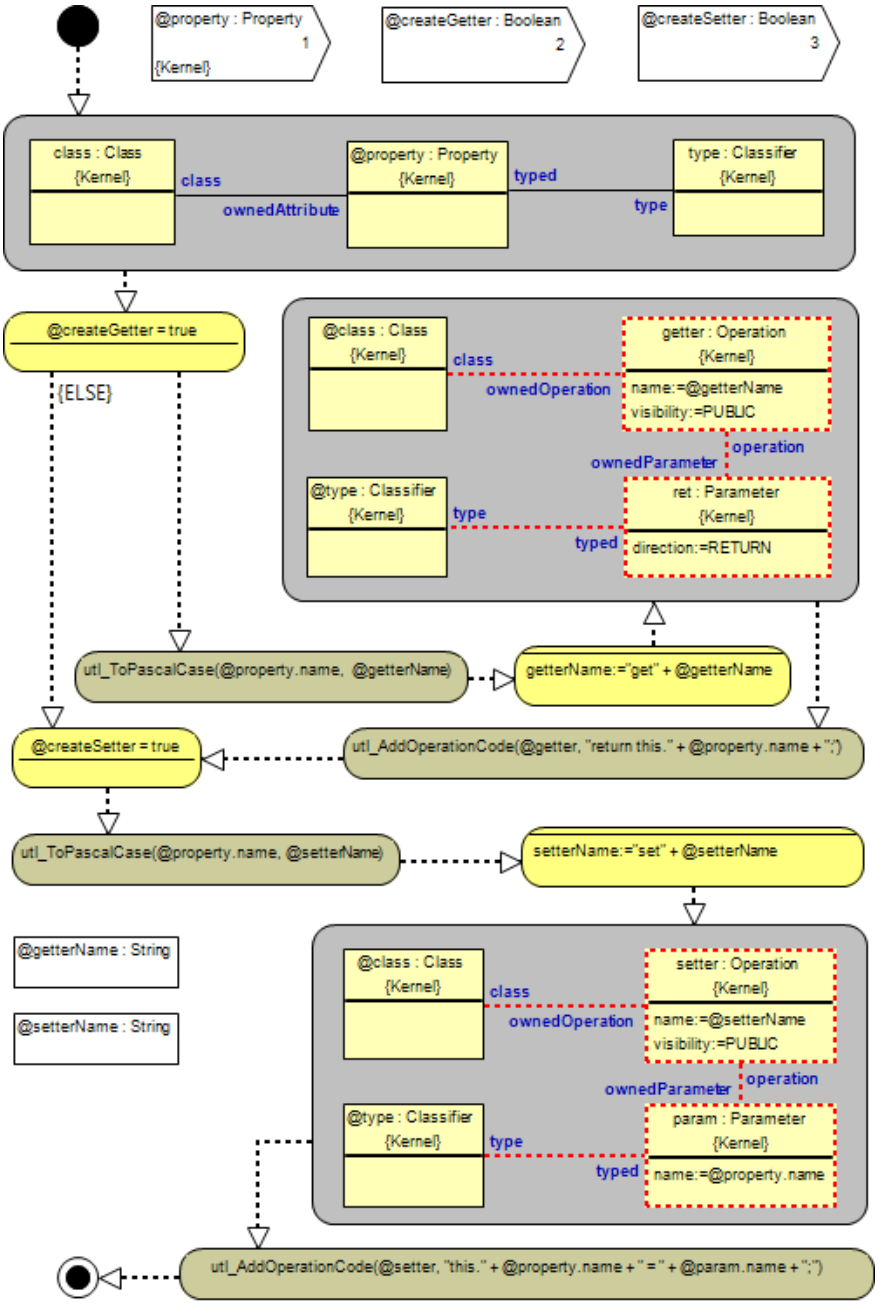
Podstawowym pakietem jest RSLtoJava. Zawiera on główną procedurę Main(), od której transformacja rozpoczyna przetwarzanie. Zawiera on również podpakiet Initializer definiujący procedury przygotowujące model docelowy (DetailedDesignModel w języku SCL), w którym następnie tworzona jest ogólna struktura docelowej aplikacji w postaci pakietów kodu oraz podstawowych klas i interfejsów. Podpakiet ScenarioParser implementuje procedury pozwalające w łatwy sposób iterować po zdaniach scenariuszy na potrzeby ich dalszego przetwarzania zgodnie z regułami translacji. ScenarioParser wykorzystuje procedury pomocnicze zdefiniowane w pakiecie Aux. Zadaniem procedury w pakiecie Finalizer jest usunięcie, po wykonaniu transformacji, wszystkich elementów tymczasowych, ułatwiających generację modelu docelowego, np. pomocniczych powiązań AllocationLink czy stereotypów.

Pozostałe pakiety, tj. DTOGenerator, ModelGenerator, PresenterGenerator oraz ViewGenerator – jak wskazują ich nazwy – definiują procedury generujące kod dla poszczególnych warstw modelu MVP oraz klasy transferu danych, zgodnie ze zdefiniowaną semantyką translacyjną. Najbardziej rozbudowane są pakiety PresenterGenerator oraz ViewGenerator z racji liczby i złożoności implementowanych przez nie reguł semantycznych, opisanych odpowiednio w rozdziałach 6.5 i 6.6. W przypadku pakietu PresenterGenerator, procedury przetwarzające poszczególne typy zdań scenariusza, zdefiniowane są w podpakietach odpowiadających tym typom. Struktura pakietu ViewGenerator jest

w największym stopniu uzależniona od wybranej platformy programistycznej. Dla platformy Echo3 wykorzystanej w opisywanej tutaj implementacji, pakiet ViewGenerator zawiera następujące podpakiety: ViewInterfaceGenerator – odpowiedzialny za generowanie kodu interfejsu `IView` oraz realizującej ten interfejs klasy `ViewImpl`; ViewClasGenerator – generujący klasy reprezentujące poszczególne ekrany aplikacji oraz komunikaty; LabelsBundle – generujący plik tekstowy definiujący etykiety ekranowe dla każdego elementu interfejsu użytkownika, dzięki czemu można je dowolnie zmieniać bez konieczności rekompilacji kodu aplikacji; Aux – zawierający szereg procedur pomocniczych, ułatwiających implementację transformacji.



Rysunek 90 – Przykładowa procedura z pakietu Utilities operująca na metamodelu jezyka RSL: utl_ GetSVOIndirectObject



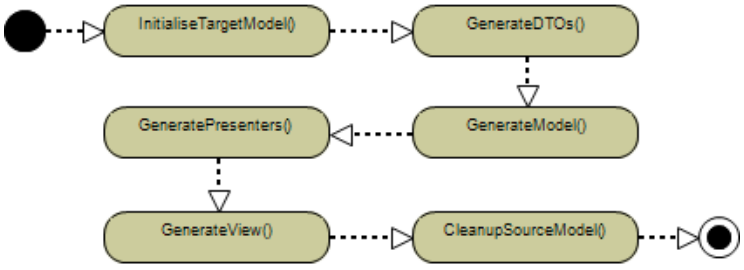
Rysunek 91 – Przykładowa procedura z pakietu Utilities operująca na metamodelu jezyka UML: utl_ CreateGetterAndSetter

Pakiet Utilities definiuje szereg generycznych procedur narzędziowych, operujących na metamodelu języka SCL, UML oraz przetwarzających tekst. Są one intensywnie wykorzystywane przez procedury we wszystkich innych pakietach transformacji RSL to Java w celu uproszczenia i zwiększenia czytelności jej kodu. Chociaż pakiet Utilities został zaimplementowany na potrzeby transformacji RSL to Java, są one w większości niezależne od zaprezentowanej semantyki translacyjnej i mogą być używane również przez inne transformacje. Rysunek 90 przedstawia przykład procedury z pakietu Utilities operującej na

metamodelu języka RSL. Procedura utl_GetSVOIndirectObject() zwraca dopełnienie dalsze zdania SVO przekazanego jako parametr. Jeżeli przekazane zdanie nie jest zdaniem złożonym, tzn. nie posiada złożonej frazy czasownikowej ComplexVerbPhrase, procedura zwraca NULL. Rysunek 91, z kolei, przedstawia przykład procedury operującej na metamodelu języka UML. Procedura utl_CreateGetterAndSetter() tworzy metody dostępowe dla atrybutu klasy. Najpierw znajduje typ atrybutu oraz klasę, do której należy atrybut a następnie tworzy w tej klasie metody typu „get" i „set" wraz z odpowiednimi parametrami oraz kodem tych metod.

Pakiety ExportToEA oraz ImportFromEA widoczne na Rysunek 89 nie są częścią transformacji RSL to Java lecz są niezależnymi transformacjami dostępnymi standardowo w narzędziu ReDSeeDS. Ich zadaniem jest przekształcanie modeli zgodnych z językiem UML na modele zgodne z metamodelem narzędzia Enterprise Architect (EA), który istotnie różni się od oficjalnego metamodelu UML [79]. Dzięki temu możliwa jest integracja narzędzi ReDSeeDS i EA – wynikowy model transformacji „RSL to Java" może być automatycznie wyeksportowany do EA, aby wykorzystać wbudowany w niego generator kodu.

Transformacja „RSL to Java" wykorzystuje również procedury zewnętrzne udostępniane przez narzędzie METAclipse w pakiecie External procedures. Dzięki temu mechanizmowi możliwe jest uruchamianie z poziomu transformacji zewnętrznych procedur napisanych w języku C++. Standardowo dostępne są procedury do wykonywania operacji na plikach tekstowych oraz do wyświetlania komunikatów i okien dialogowych w czasie wykonania transformacji.



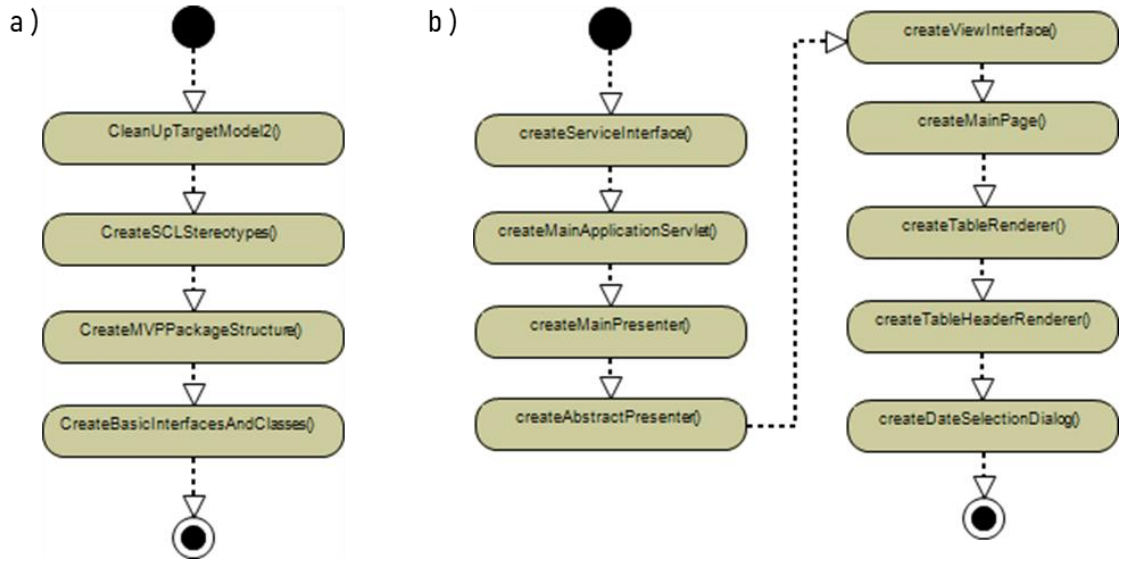
Rysunek 92 – Główna procedura transformacji RSL to Java

Rysunek 92 przedstawia główną procedurę transformacji – procedurę Main(). Jest to prosta sekwencja wywołań procedur z poszczególnych pakietów. Przetwarzanie rozpoczyna się od przygotowania docelowego modelu UML oraz wygenerowania ogólnej struktury kodu aplikacji (procedura InitialiseTargetModel()). W modelu docelowym tworzone są następnie konstrukcje składające się na poszczególne warstwy w architekturze MVP, zgodnie ze zdefiniowanymi

regułami translacji. Są to kolejno procedury: GenerateDTOs(), GenerateModel(), GeneratePresenters(), GenerateView(). Transformacja kończy się wywołaniem procedury CleanupSourceModel(), której zadaniem jest usunięcie z modelu źródłowego wszystkich pomocniczych elementów utworzonych w trakcie działania transformacji.

6.3 Gnerowanie ogólnej struktury kodu

Ogólny schemat tworzenia inicjalnej struktury docelowej aplikacji przedstawia Rysunek 93. Część (a) rysunku prezentuje sekwencję działań wykonywanych w ramach procedury InitialiseTargetModel(). Najpierw przygotowywany jest model docelowy poprzez usunięcie wszelkich elementów, które zostały utworzone w trakcie poprzedniego wykonania transformacji oraz tworzone są stereotypy (elementy typu `Stereotype`) wykorzystywane pomocniczo do oznaczania elementów metamodeli w celu ułatwienia implementacji transformacji. Odpowiadają za to odpowiednio procedury: CleanUpTargetModel() oraz CreateSCLStereotypes().

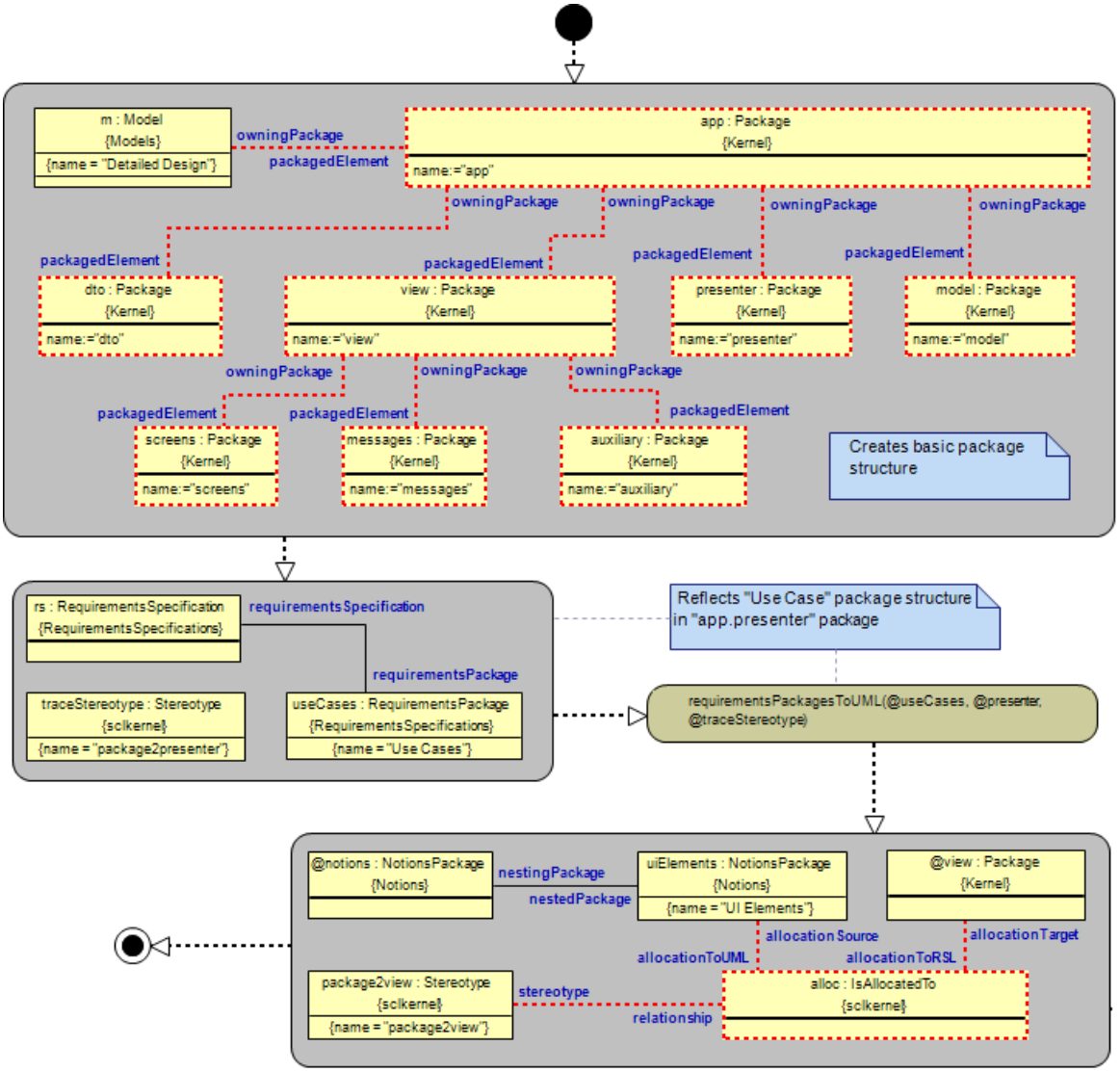


Rysunek 93 – Procedura InitialiseTargetModel() (a) oraz CreateBasicInterfacesAndClasses() (b)

Kolejnym krokiem jest utworzenie struktury pakietów kodu, zgodnej z przyjętym środowiskiem translacyjnym (patrz: rozdział 5.1) oraz wybranymi regułami translacji. Krok ten realizowany jest przez procedurę CreateMVPPackageStructure() zilustrowaną na Rysunek 94.

Pierwsza reguła tej procedury tworzy podstawową, niezależną od modelu źródłowego, strukturę pakietów klas, która jest umieszczana w modelu o nazwie „Detailed Design”, stanowiącym standardowy element każdego Software Case’a. Wynikiem działania tej reguły

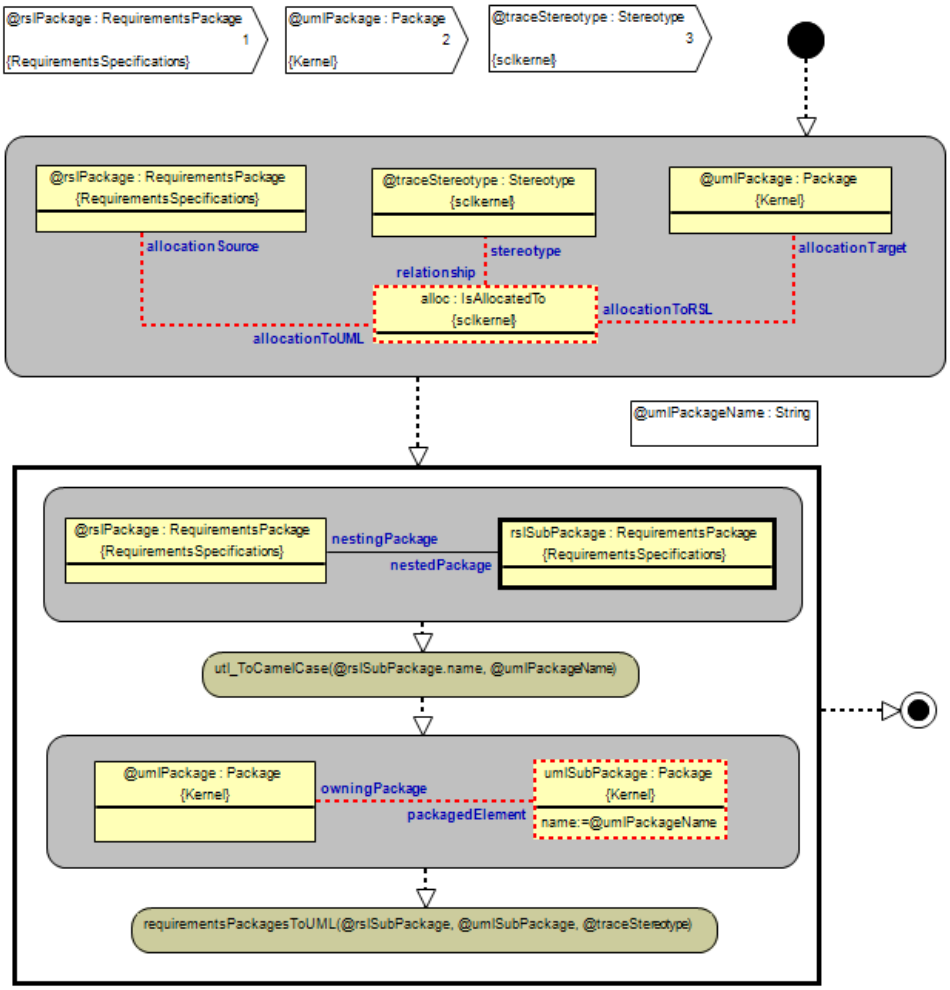
jest struktura pakietów odpowiadająca tej pokazanej na Rysunek 53 w rozdziale 5.1. Pakiet „view” zawiera jeden dodatkowy podpakiet o nazwie „auxiliary”, który przechowuje pomocnicze klasy niezbędne do implementacji interfejsu użytkownika w środowisku Echo3.



Rysunek 94 – Procedura CreateMVPPackageStructure()

Dalsza część procedury, generuje podpakiety wynikające ze struktury modelu wymagań w języku RSL. Drzewo pakietów grupujące przypadki użycia w modelu RSL odzwierciedlane jest w pakiecie „presenter”. W ten sposób realizowana jest reguła translacji G1. Odpowiedzialna jest za to rekurencyjna procedura RequirementsPackagesToUML() przedstawiona na Rysunek 95. Jej dwa pierwsze parametry oznaczają odpowiednio pakiet RSL, którego zawartość ma być odzwierciedlona oraz pakiet docelowy UML, w którym nastąpi odzwierciedlenie. Trzeci parametr to stereotyp, którym zostaną oznaczone relacje identyfikowalności (`IsAllocatedTo`) pomiędzy pakietami źródłowymi i docelowymi.

Relacja identyfikowalności tworzona jest także w ostatniej regule – w tym przypadku pomiędzy pakietem „UI Elements”, przechowującym pojęcia RSL reprezentujące elementy interfejsu użytkownika, a pakietem „view”, w którym będą umieszczane klasy implementujące te pojęcia.



Rysunek 95 – Procedura requirementsPackagesToUML()

Ostatnim krokiem inicjalizacji modelu docelowego jest utworzenie interfejsów oraz klas, które nie wynikają bezpośrednio z modelu wymagań lecz stanowią ramy przyjętego środowiska docelowego, niezbędne do jego poprawnego funkcjonowania (patrz: rozdział 5.1). Jest to realizowane w ramach procedury GenerateBasicInterfacesAndClasses() przedstawionej na Rysunek 93b. Jest ona sekwencją podprocedur, których wynikiem jest wyjściowy kod klas i interfejsów, który następnie jest uzupełniany w dalszej części transformacji, implementującej zdefiniowane reguły translacji. Co istotne, generowany kod uwzględnia wszystkie niezbędne

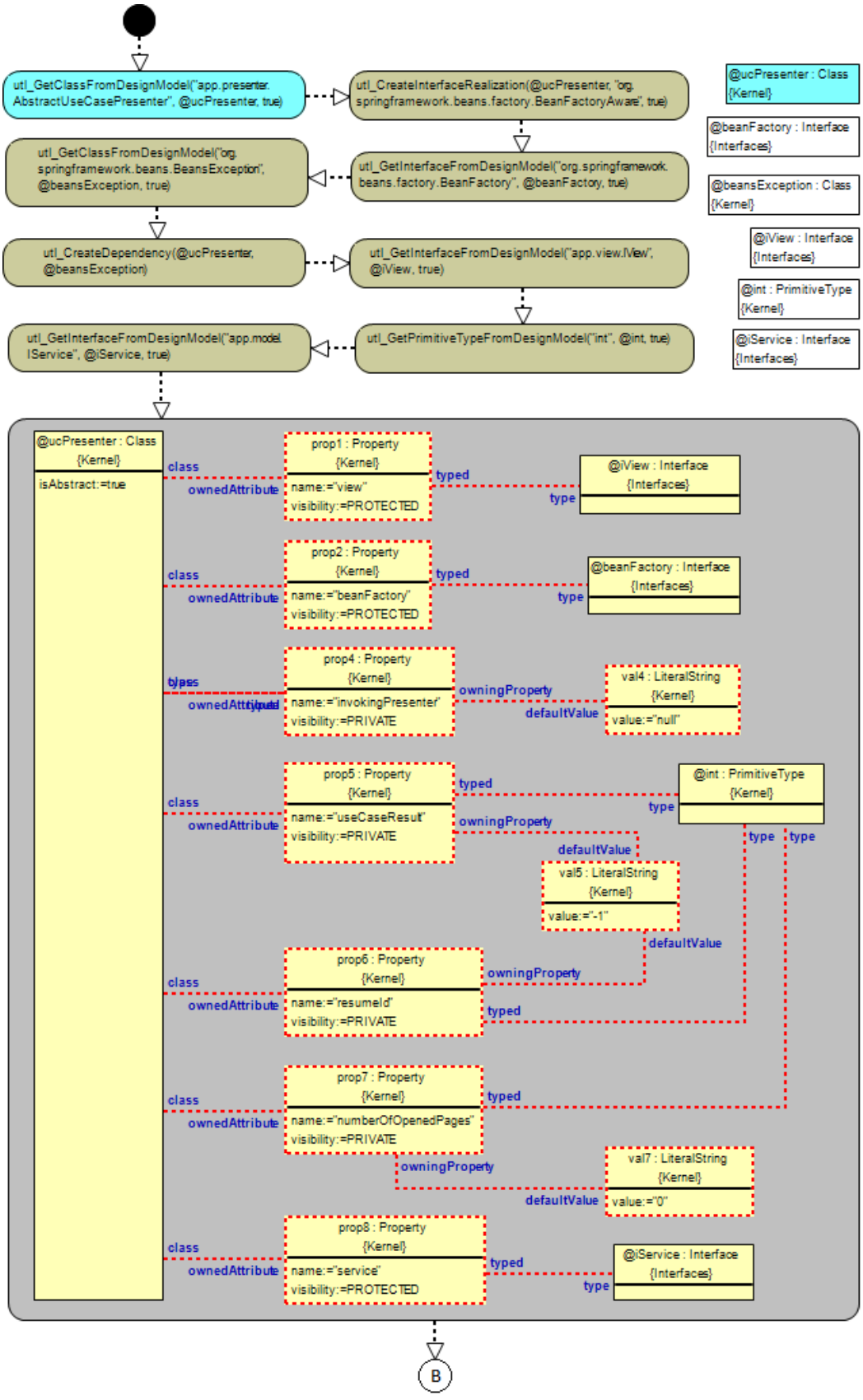
zależności od elementów środowisk technologicznych (Echo3, Spring[30], Java) aby umożliwić jego bezbłędną kompilację, m.in. import wymaganych klas oraz pakietów, dziedziczenia klas oraz realizacje interfejsów, itp. W wyniku wykonania kolejnych podprocedur, generowane są następujące elementy:

a) createServiceInterface() – interfejs `IService` oraz jego implementacja `ServiceImpl`,
b) createMainApplicationServlet() – klasa `MainApplicationServlet` będąca specyficzną dla środowiska Echo3 implementacją serwletu zwracającego instancję aplikacji w odpowiedzi na żądanie HTTP,
c) createMainPresenter() – klasa `MainPresenter`,
d) createAbstractPresenter() – klasa `AbstractPresenter`,
e) createViewInterface() – interfejs `IView` oraz jego implementacja `ViewImpl`, pełniąca jednocześnie rolę instancji aplikacji Echo3 zwracanej przez `MainApplicationServlet`,
f) createMainPage() – klasa `MainPage`,
g) createTableRenderer(), createTableHeaderRenderer() oraz createDateSelectionDialog() – klasy pomocnicze Echo3 implementujące generyczne mechanizmy interfejsu użytkownika.

Najistotniejszą, a zatem wartą szerszego omówienia procedurą spośród wymienionych jest createAbstractPresenter(). Jej kod przedstawia Rysunek 96 oraz Rysunek 97. Wynikiem jej działania jest klasa implementująca logikę wspólną dla wszystkich klas prezenterów, jak opisano w rozdziale 5.1. Pierwszą część procedury stanowi sekwencja ośmiu wywołań procedur narzędziowych z pakietu Utilities. Pierwsze z nich (wywołanie procedury utl_GetClassFromDesignModel()) tworzy nową klasę (lub pobiera już istniejącą, jeżeli została wcześniej utworzona), której lokalizację i nazwę określa pierwszy parametr wywołania. W tym przypadku jest to klasa `AbstractUseCasePresenter` umieszczana w pakiecie `app/presenter`. Kolejne cztery wywołania procedur mają związek z zastosowania środowiska Spring w celu wstrzykiwania zależności (ang. Dependency Injection) między obiektami klas generowanej aplikacji. Procedury te tworzą klasy oraz interfejsy specyficzne dla środowiska Spring aby następnie utworzyć relacje pomiędzy nimi a klasą `AbstractUseCasePresenter` oraz użyć je jako typy atrybutów lub parametrów operacji

[30] https://spring.io/

klasy. Należy przy tym zaznaczyć, że klasy środowiska Spring (jak i wszystkie inne klasy zewnętrznych bibliotek, w tym Echo3) tworzone są jedynie po to aby docelowy kod klas generowanej aplikacji odzwierciedlał wszystkie wymagane odwołania do klas bibliotecznych, np. import odpowiednych klas. Kod klas bibliotecznych nie jest generowany – aby skompilować wygenerowany kod aplikacji, należy zapewnić obecność odpowiednich bibliotek zewnętrznych.



Rysunek 96 – Procedura CreateAbstractPresenter (część A)

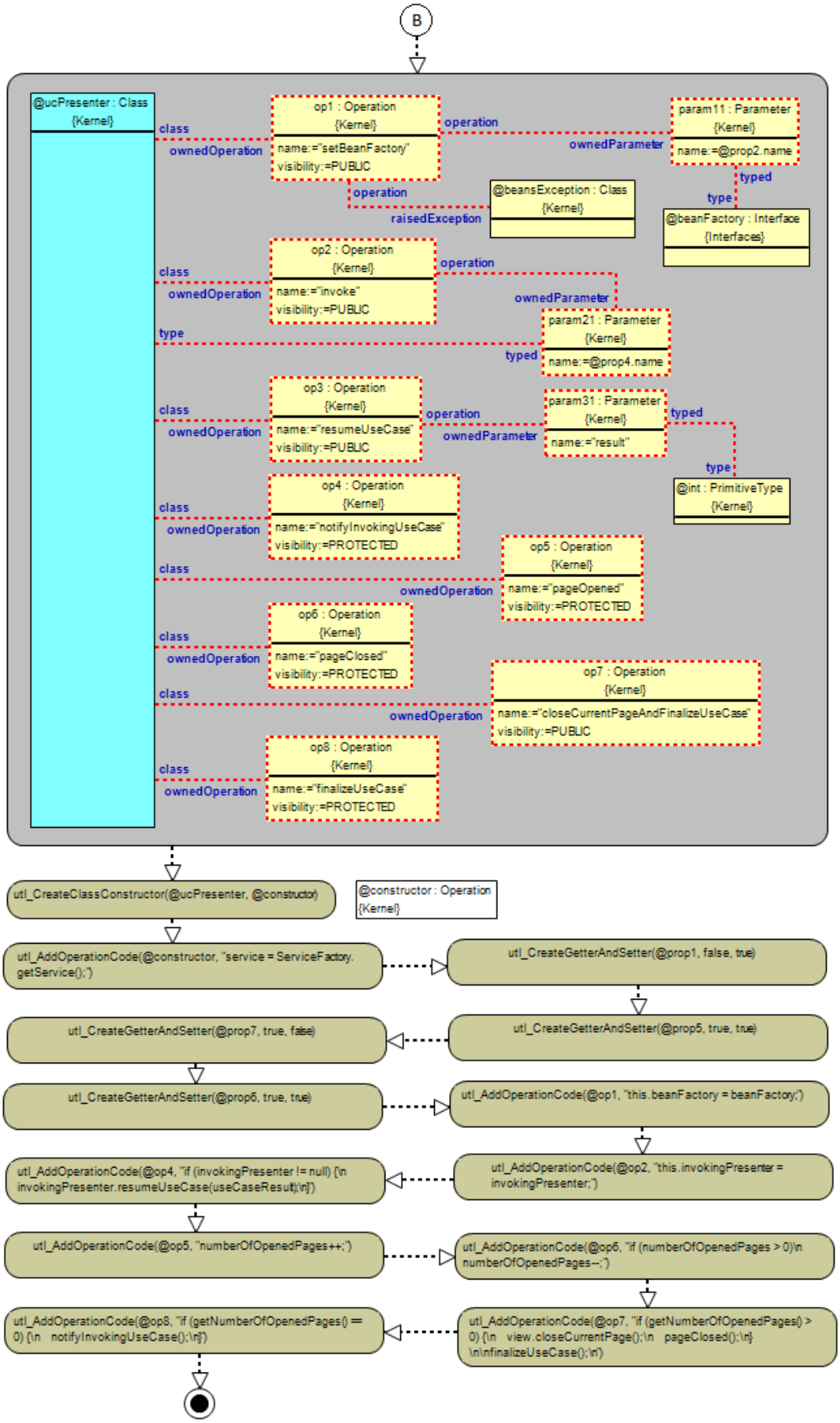
Kolejne wywołania procedury narzędziowej utl_GetInterfaceFromDesignModel() tworzą również interfejs IView oraz IService a także typ podstawowy „int” i przypisują utworzone elementy do zmiennych lokalnych (@iView, @iService, @int), aby odwołać się do nich w dalszej części procedury, tj. w regule przedstawionej w dolnej części Rysunek 96. Reguła ta tworzy wszystkie niezbędne atrybuty klasy AbstactUseCasePresenter, zgodnie z definicją przedstawioną w rozdziale 5.1 oraz dodatkowy atrybut (beanFactory) wynikający z zastosowania środowiska Spring. Dla każdego atrybutu określana jest nazwa, widoczność oraz typ. Do określenia typów atrybutów, wykorzystane są m.in. zmienne lokalne stanowiące referencje do utworzonych (lub pobranych) wcześniej elementów. Dla niektórych atrybutów określane są także wartości inicjalne (defaultValue).

Kolejna reguła omawianej procedury (przedstawiona w górnej części Rysunek 97) tworzy wszystkie niezbędne operacje klasy AbstactUseCasePresenter a w zasadzie sygnatury operacji – kod operacji tworzony jest w dalszej części procedury. Podobnie jak w przypadku atrybutów, tworzone operacje wynikają z definicji abstrakcyjnego środowiska translacyjnego. Wyjątkiem jest operacja setBeanFactory(), która służy do realizacji wstrzykiwania zależności z użyciem biblioteki Spring. Dla niektórych operacji, oprócz nazwy i widoczności, tworzone są parametry wejściowe, definiowany jest typ zwracany oraz typ zgłaszanych wyjątków.

Omawiana reguła nie tworzy natomiast metod typu „get” oraz „set” dla atrybutów, które tego wymagają. Metody te tworzone są w ostatniej części omawianej procedury (patrz: dolna część Rysunek 97), poprzez wywołanie procedury narzędziowej utl_CreateGetterAndSetter() dla odpowiednich atrybutów. Sposób działania tej procedury opisano w rozdziale 6.2.

Oprócz metod dostępowych, ostatnia część procedury CreateAbstractUseCasePresenter() tworzy również kod operacji utworzonych we wcześniejszej regule. Kod operacj dodawany jest poprzez wywołanie procedury utl_AddOperationCode(), której pierwszy parametr określa operację a drugi – kod tej operacji.

Efektem wykonania procedury CreateAbstractPresenter() jest kompletny kod klasy AbstractUseCasePresenter, z której dziedziczą wszystkie konkretne klasy prezenterów przypadków użycia, tworzone w dalszej części transformacji, zgodnie z regułami translacji do warstwy prezentera (patrz: rozdział 6.5).

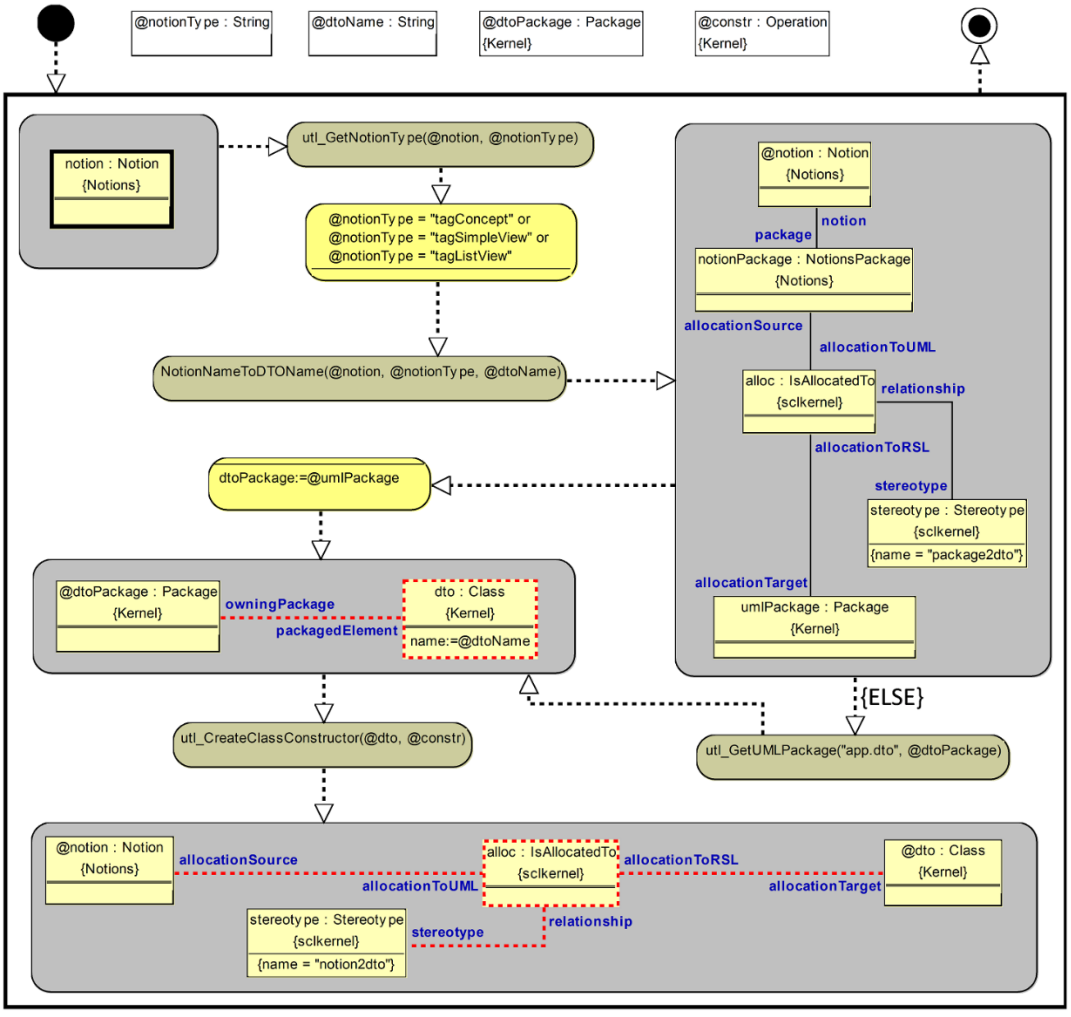


Rysunek 97 – Procedura CreateAbstractPresenter (część B)

6.4 Generowanie klas DTO

Sposób generowania klas typu DTO na podstawie elementów dziedzinowych języka RSL określają reguły translacji od G7 do G9. Klasy te służą do enkapsulacji danych przesyłanych pomiędzy warstwami w modelu MVP przez co zwiększają czytelność generowanego kodu oraz czynią go bardziej uporządkowanym. Za implementację wspomnianych reguł odpowiada pakiet GenerateDTOs(), którego najistotniejsze fragmenty zostały opisane poniżej.

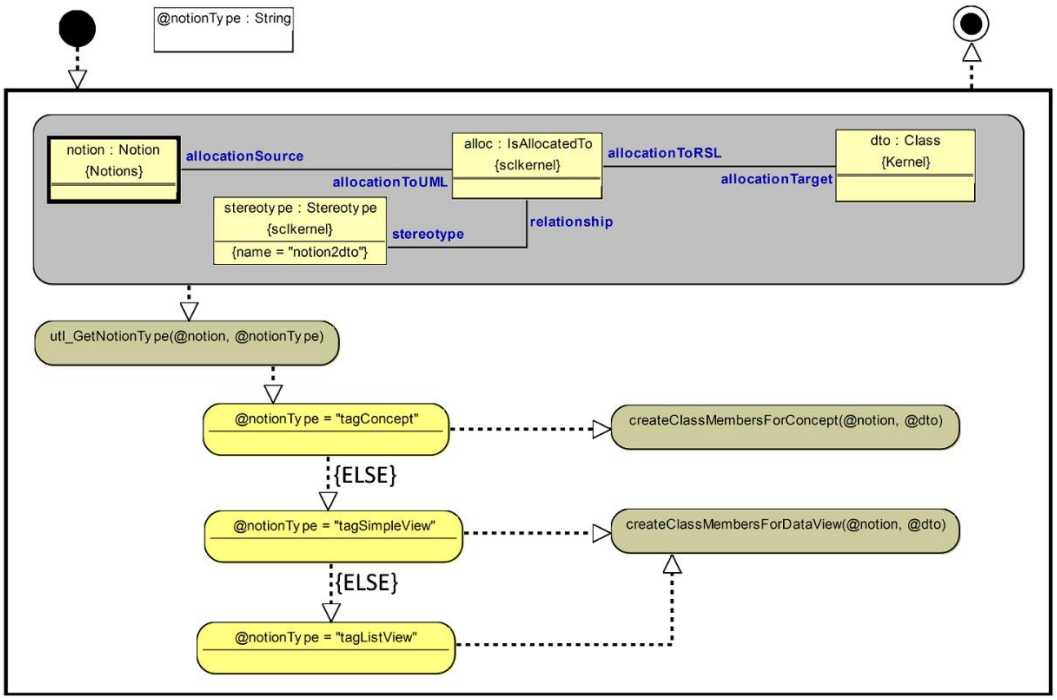
Pierwszą zasadniczą procedurą jest CreateDTOClasses(), której zadaniem jest utworzenie pustych klas (bez atrybutów oraz metod) odpowiadających pojęciom dziedzinowym określonym przez reguły translacji oraz umieszczenie ich w odpowiednim pakiecie klas. Procedura ta zilustrowana jest na Rysunek 98.



Rysunek 98 – Procedura CreateDTOClasses()

Składa się ona z pojedynczej pętli for-each, która iteruje po wszystkich pojęciach dziedzinowych (Notion) w modelu źródłowym. Dla każdego pojęcia sprawdzany jest najpierw

jego typ poprzez wywołania procedury pomocniczej utl_GetNotionType(). Zgodnie z regułami G7 – G9, klasy DTO tworzone są dla pojęć typu „Concept”,”Simple View” oraz „List View”. Dla każdego pojęcia spełniającego ten warunek tworzona jest nazwa klasy DTO (procedura NotionNameToDTOName()) a następnie pobierany jest pakiet UML, w którym będzie umieszczona nowoutworzona klasa (reguła w górnym prawym rogu na Rysunek 98). Jeśli taki pakiet został wcześniej utworzony, jest on powiązany z pakietem pojęć RSL, w którym znajduje się aktualnie przetwarzane pojęcie, za pomocą relacji „IsAllocatedTo” ze stereotypem „package2dto”. Jeżeli taki pakiet nie istnieje, klasa będzie umieszczona w standardowym pakiecie „dto”. Kolejna reguła tworzy klasę DTO w wyznaczonym pakiecie a następujące po niej wywołanie procedury utl_CreateConstructor() dodaje do klasy standardowy konstruktor. Ostatnim krokiem jest wykonanie reguły, która tworzy pomiędzy klasą a odpowiadającym jej pojęciem relację „IsAllocatedTo” ze stereotypem „notion2dto”.



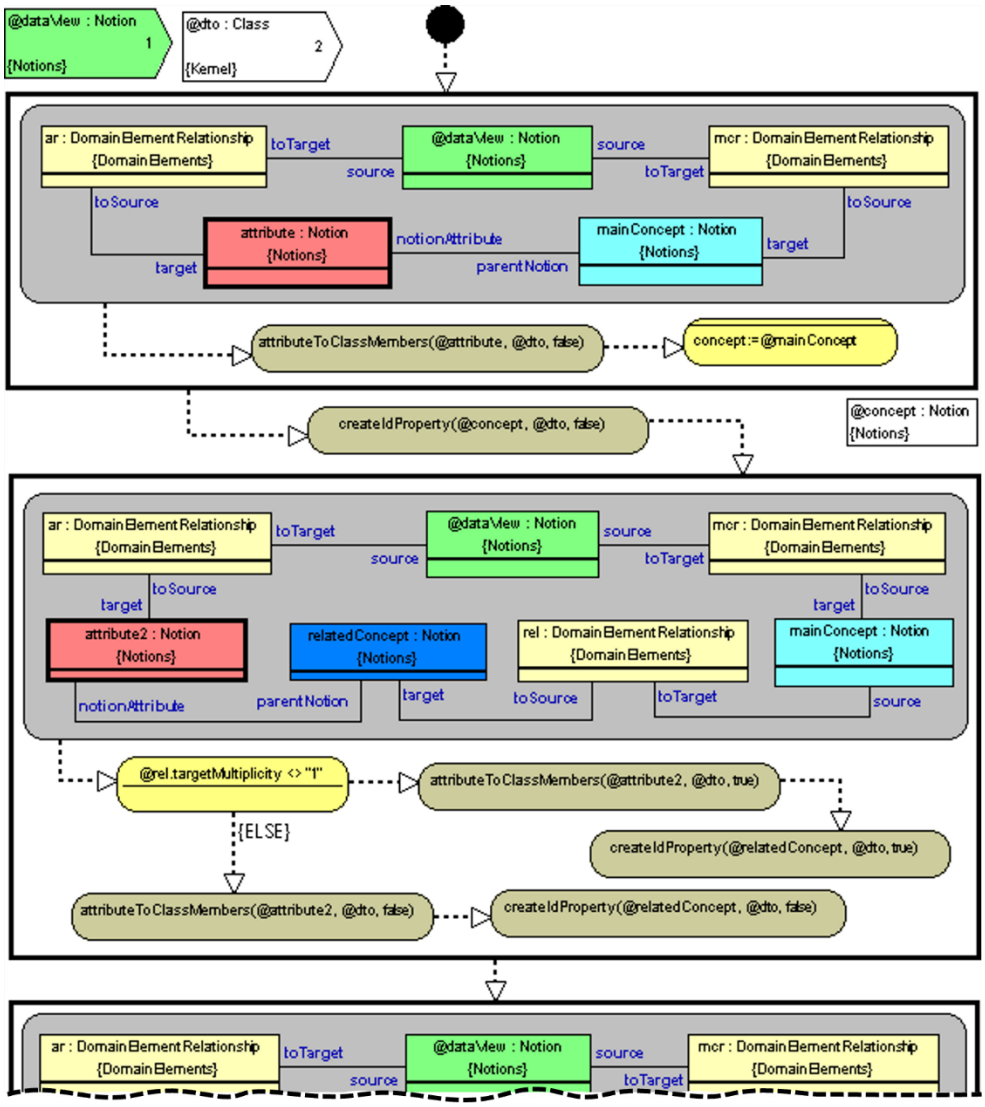
Rysunek 99 – Procedura CreateDTOClassMembers()

Drugą zasadniczą procedurą wykonywaną w celu wygenerowania kompletnych klas DTO jest CreateDTOClassMembers(), pokazana na Rysunek 99. Jest ona odpowiedzialna za wygenerowanie wszystkich atrybutów oraz metod (akcesorów i mutatorów) w klasach utworzonych w ramach procedury CreateDTOClasses(). W tym celu procedura iteruje po wszystkich pojęciach typu Notion, dla których została wcześniej wygenerowana klasa typu

DTO, tj. tych, które posiadają relację typu „IsAllocatedTo” ze stereotypem „notion2dto” łączącą je z obiektem typu „Class”. Następnie, dla każdego pojęcia spełniającego tę regułę, wywoływana jest właściwa procedura generująca atrybuty i metody. W zależności od typu pojęcia jest to createClassMembersForConcept() lub createClassMembersForDataView(). Wynika to z faktu, że, zgodnie z regułami translacji G7 – G9, mechanizm generowania atrybutów i metod dostępowych jest nieco inny dla pojęć dziedzinowych różnych typów.

Procedura createClassMembersForConcept() jest względnie prosta, gdyż generuje atrybuty i metody klasy DTO jedynie na podstawie atrybutów bezpośrednio należących do odpowiedniego pojęcia typu „Concept”. Procedura createClassMembersForDataView() jest bardziej złożona, gdyż musi uwzględniać wskazywane przez pojęcie typu „Simple View” oraz „List View” atrybuty wielu pojęć typu „Concept”. Musi również uwzględniać relacje między pojęciami dziedziny problemu, a także relacje typu „main concept”. Fragment tej procedury został przedstawiony na Rysunek 100.

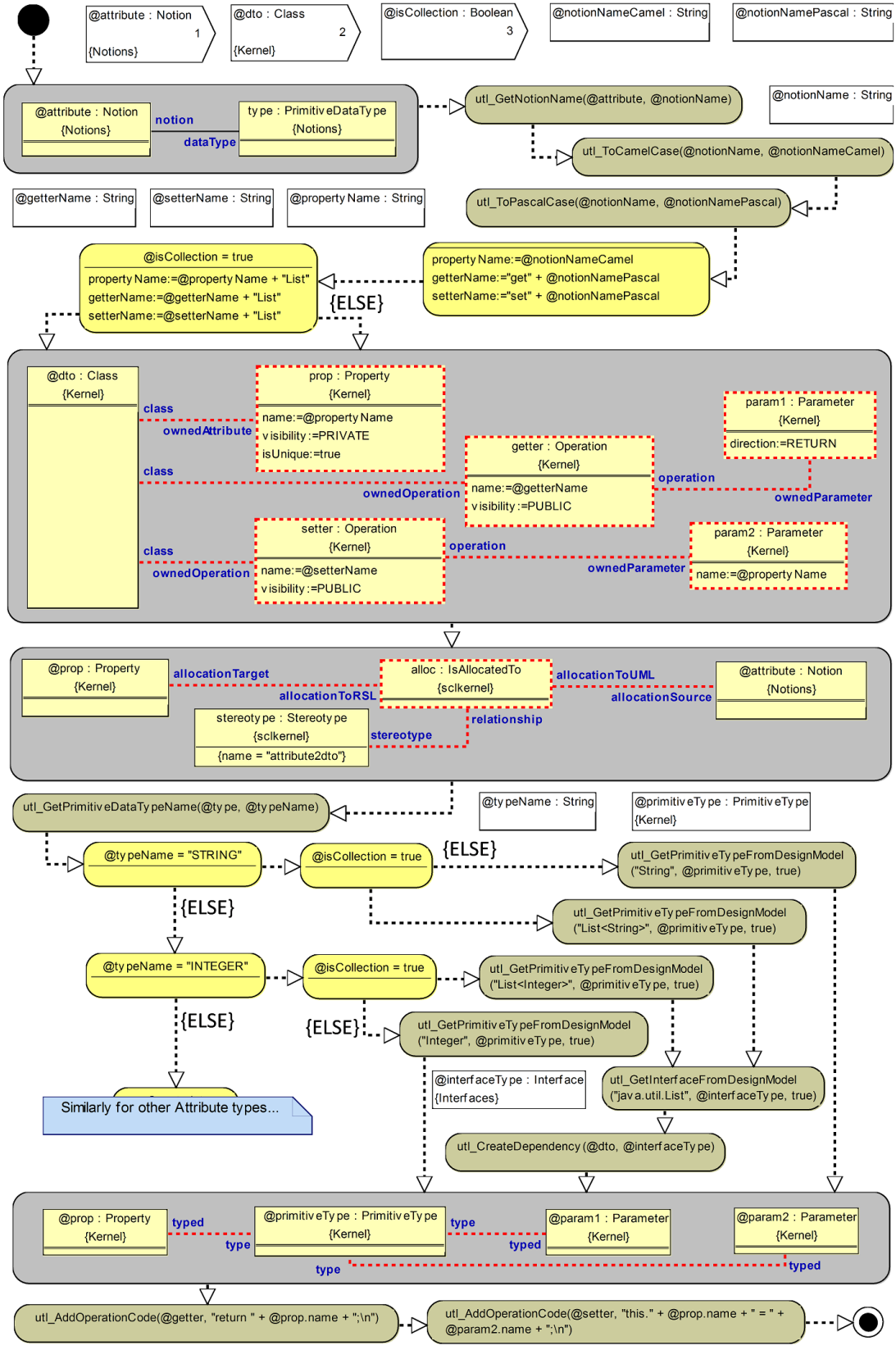
Procedura ta ma dwa parametry wejściowe: pojęcie typu Notion (@DataView) oraz klasę DTO wygenerowaną wcześniej dla tego pojęcia (@dto). Implementacja procedury składa się z trzech pętli for-each, wywoływanych jedna po drugiej. Każda z nich iteruje po atrybutach wskazywanych przez pojęcie @DataView i – zgodnie z regułami translacji G7 oraz G8 – tworzy odpowiadające im właściwości w klasie @dto poprzez wywołanie z odpowiednimi parametrami podprocedur attributeToClassMembers() oraz createIdProperty(). Pierwsza pętla uwzględnia tylko atrybuty (@attribute) należące bezpośrednio do pojęcia „main concept” (@mainConcept) wskazywanego przez widok danych. Dwie kolejne pętle uwzględniają atrybuty pojęć powiązanych (@relatedConcept) z pojęciem „main concept”, biorąc pod uwagę krotność tego powiązania, zgodnie z regułami translacji. Jeżeli krotność jest różna od „1”, w docelowej klasie DTO generowany jest adekwatny atrybut typu kolekcja (wraz z parą metod dostępowych). Dzieje się to poprzez wywołanie procedury attributeToClassMembers() oraz createIdProperty() z trzecim parametrem o wartości „true”. W przypadku krotności równej „1”, w klasie DTO tworzone są atrybuty typu skalarnego. Pętla druga i trzecia są niemal identyczne, różnią się jedynie źródłem i celem relacji @rel.



Rysunek 100 – Procedura CreateClassMembersForDataView()

Sposób tworzenia właściwości w klasach DTO w ramach procedury attributeToClassMembers() przedstawia Rysunek 101. Procedura ma trzy parametry wywołania: atrybut pojęcia dziedzinowego podlegający translacji (@attribute), klasę, w której mają być tworzone właściwości (@dto) oraz wartość typu Boolean (@isCollection), która określa czy tworzona właściwość klasy ma być kolekcją (wartość „true") czy typem skalarnym (wartość „false"). Pierwsza część procedury określa nazwy dla tworzonego atrybutu oraz metod dostępowych, na podstawie nazwy atrybutu pojęcia dziedzinowego oraz przyjętej konwencji nazewniczej dla generowanego kodu. Następnie odpowiedni atrybut (@prop) oraz metody dostępowe (@getter i @setter) dodawane są do klasy DTO. Tworzona jest również relacja indentyfikowalności (IsAllocatedTo) między atrybutem źródłowym a atrybutem klasy. W dalszej części określany jest typ atrybutu klasy oraz parametrów metod dostępowych poprzez mapowanie typu atrybutu źródłowego na typ języka Java, zgodnie z regułami translacji.

Na koniec, do utworzonych operacji klasy dodawany jest kod typowy dla akcesorów i mutatorów.

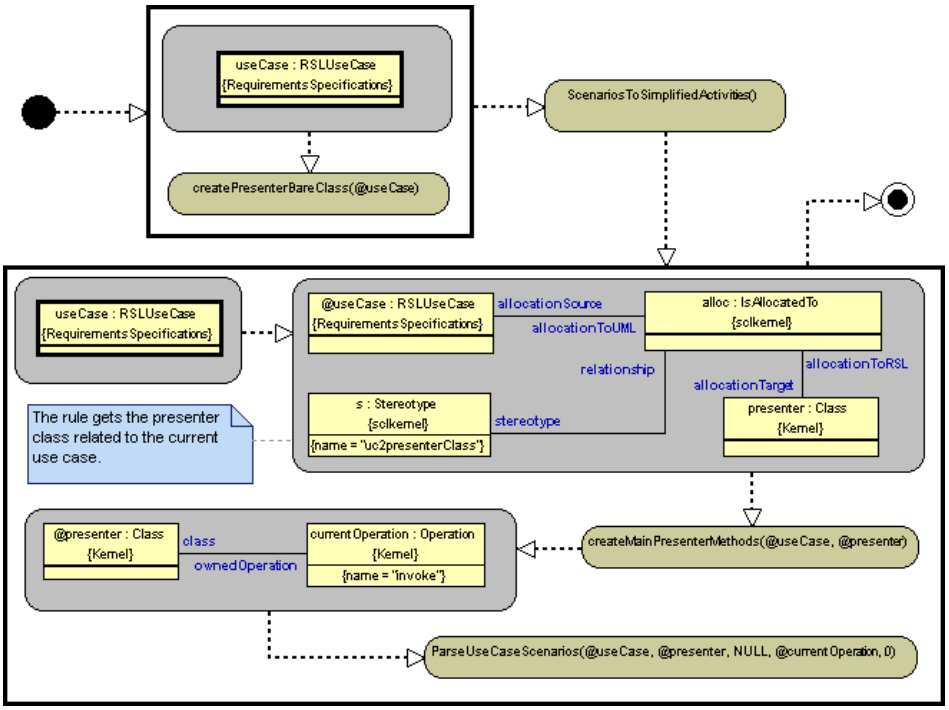


Rysunek 102 – Procedura AttributeToClassMembers()

6.5 Generowanie kodu w warstwie prezentera

Generator warstwy prezentera jest najbardziej rozbudowaną i najciekawszą częścią transformacji RSL to Java. Implementuje on reguły translacji G2 i G5 (opisane w rozdziale 5.2) oraz P1 do P16 (opisane w rozdziale 5.4). Pozwala to generować kompletny kod logiki aplikacji (wraz z wszystkimi niezbędnymi odwołaniami do warstw widoku oraz modelu), który w pełni odzwierciedla logikę zapisaną w scenariuszach przypadków użycia.

Na implementację tej części transformacji składa się 50 procedur w języku MOLA. Rozdział ten ogranicza się do przedstawienia ogólnego schematu działania parsera scenariuszy przypadków użycia oraz wybranych procedur przetwarzających najważniejsze z punktu widzenia logiki aplikacji typy zdań.



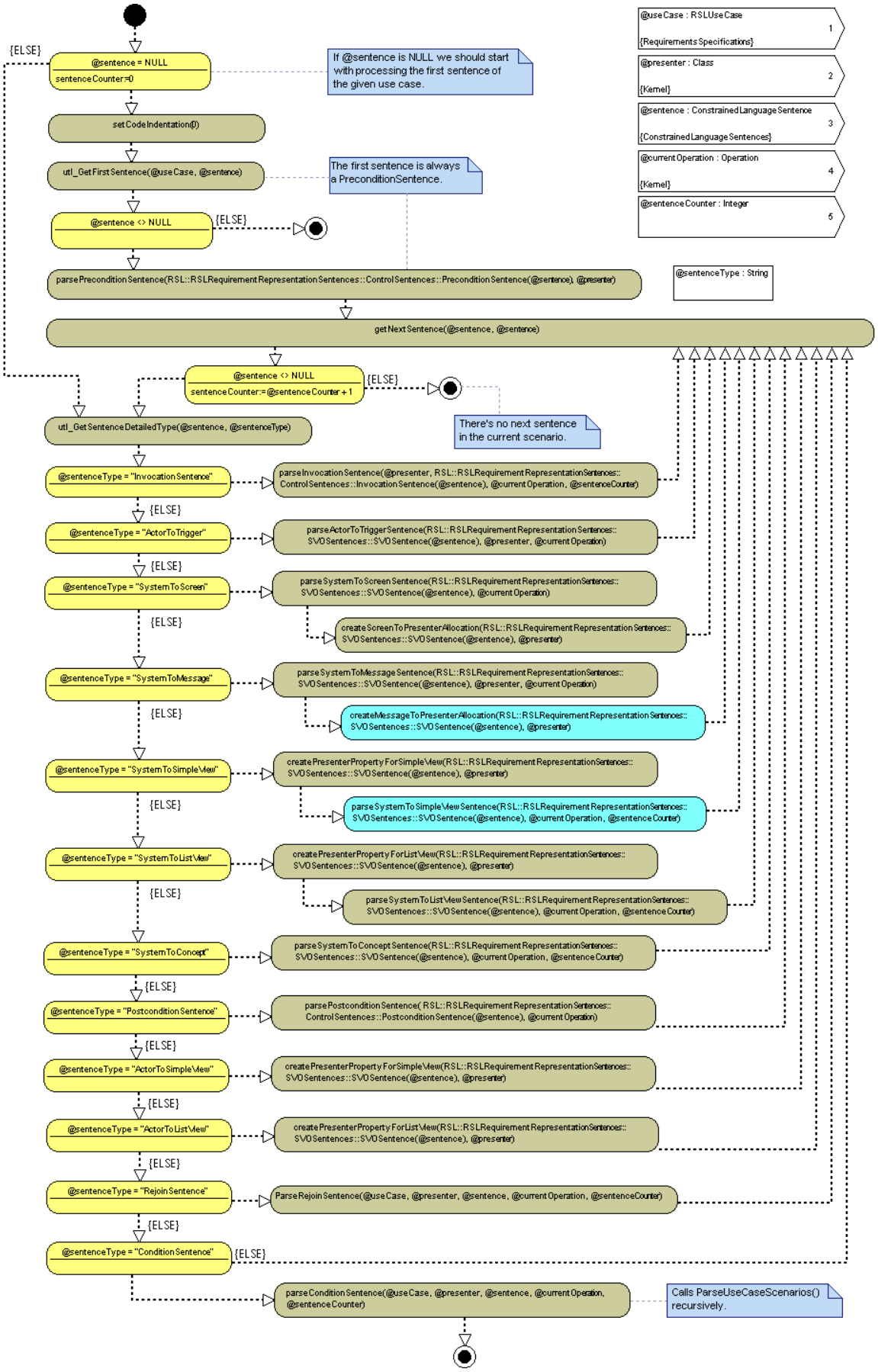
Rysunek 103 – Procedura GeneratePresenters()

Generowanie warstwy prezentera rozpoczyna się od wywołania procedury GeneratePresenters() przedstawionej na Rysunek 103. Pierwsza pętla for-each, w ramach tej procedury, iteruje po wszystkich przypadkach użycia i dla każdego z nich tworzy – zgodnie z

regułą G2 – odpowiednią klasę prezentera dziedziczącą z klasy AbstractUseCasePresenter wraz z podstawowymi generycznymi metodami. Odpowiada za to procedura createPresenterBareClass().

Kolejnym krokiem jest wywołanie procedury ScenariosToSimplifiedActivities(). Jest to pomocnicza procedura, która dla każdego przypadku użycia tworzy grafową reprezentację jego scenariuszy, zgodną z metamodelem opisanym w rozdziale 4.7. Grafowa reprezentacja scenariuszy, zamiast linearnej, ułatwia i przyspiesza przechodzenie między kolejnymi zdaniami scenariuszy, zwłaszcza w przypadku rozgałęzień warunkowych oraz pętli. Z grafowej postaci scenariuszy korzystają procedury pomocnicze intensywnie wykorzystywane przez generator warstwy prezentera. Są to: getNextSentence(), getPreviousSentence(), getNumberOfOutgoingConditions().

Druga pętla procedury GeneratePresenters(), dla każdego przypadku użycia oraz odpowiadającej mu klasy prezentera, wywołuje dwie procedury: createMainPresenterMethods() oraz ParseUseCaseScenarios(). Pierwsza z nich implementuje regułę G5 tworząc odpowiednie właściwości w klasie `MainPresenter`. Druga – parsuje scenariusze przypadku użycia i generuje kod zgodnie z regułami P1 do P16. Warto zwrócić uwagę, że parametrami wywołania tej procedury są m.in.: @useCase, @presenter oraz @currentOperation. Pierwszy i drugi wskazują odpowiednio aktualnie przetwarzany przypadek użycia oraz powiązaną z nim klasę prezentera. Drugi parametr, natomiast, wskazuje metodę klasy prezentera, od której procedura ParseUseCaseScenarios() rozpocznie wstawianie kodu generowanego w ramach bieżącego wywołania tej procedury. Przy pierwszym wywołaniu dla danego przypadku użycia, kiedy rozpoczynamy parsowanie scenariusza od pierwszego zdania, będzie to zawsze metoda `invoke()`, która uruchamia całą logikę prezentera przy wywołaniu przypadku użycia (patrz: opis środowiska translacyjnego w rozdziale 5.1). Stąd, przed wywołaniem procedury ParseUseCaseScenarios() wykonywana jest reguła pobierająca operację `invoke()` zdefiniowaną w klasie prezentera dla aktualnego przypadku użycia.



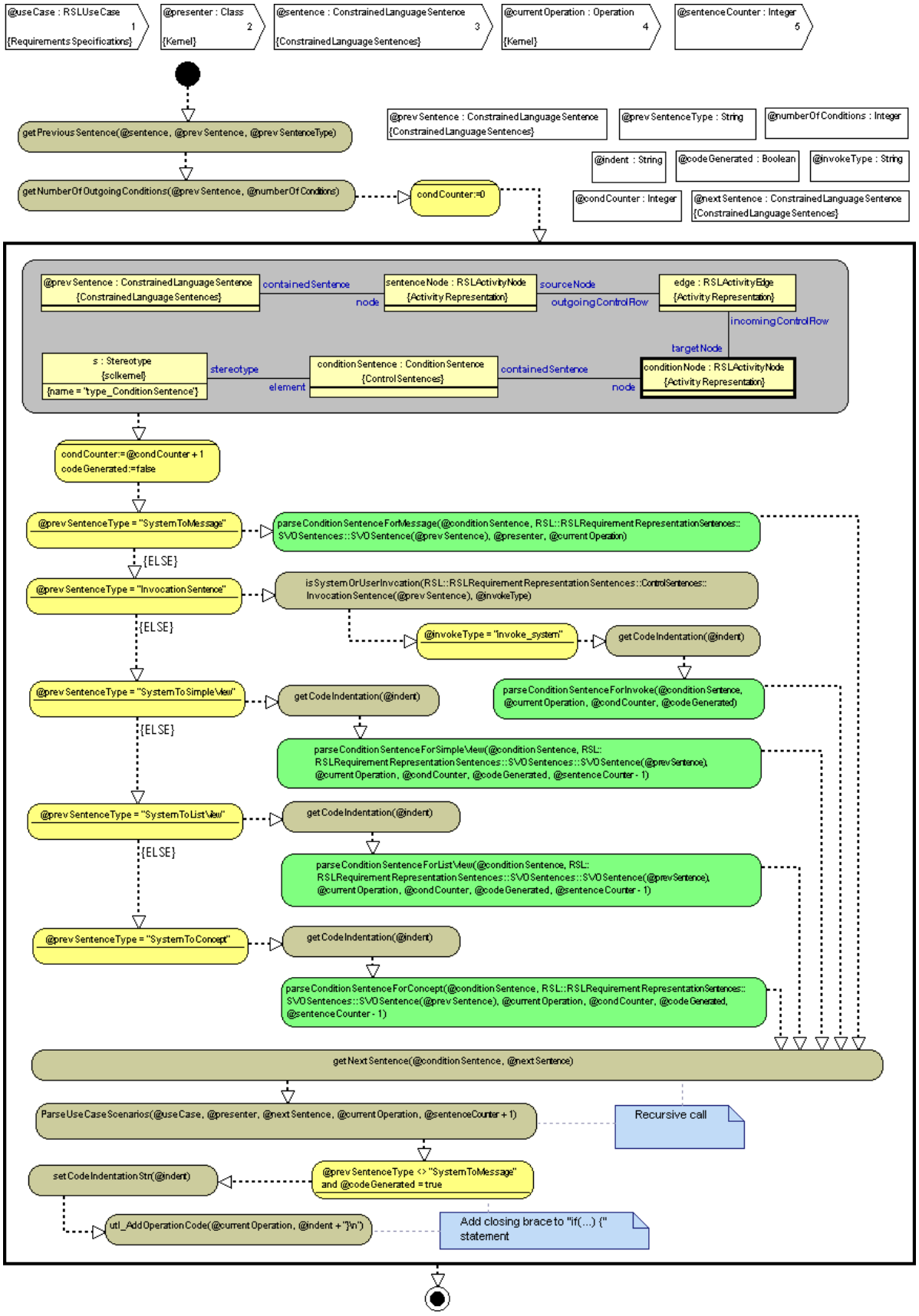
Rysunek 104 – Procedura ParseUseCaseScenarios()

Procedura ParseUseCaseScenarios() została przedstawiona na Rysunek 104. Iteruje ona po kolejnych zdaniach scenariuszy danego przypadku użycia i generuje odpowiadający im kod poprzez wywoływanie procedur dla konkretnych typów zdań. Aby zapewnić, że zdania wszystkich scenariuszy zostaną właściwie przetworzone – również w przypadku wielokrotnych rozgałęzień warunkowych – procedura wykorzystuje rekurencję, co będzie omówione dalej.

Oprócz parametrów wejściowych opisanych wcześniej, procedura ParseUseCaseScenarios() posiada jeszcze dwa dodatkowe parametry. Parametr @sentenceCounter jest licznikiem zdań przetworzonych w ramach wykonania procedury i służy przede wszystkim jako identyfikator zdania typu Invoke przy generowaniu bloków warunkowych w metodzie `resumeUseCase()`, zgodnie z regułą P3. Parametr @sentence, z kolei, wskazuje konkretne zdanie, od którego procedura rozpoczyna przetwarzanie dalszej części scenariusza. Ma to znaczenie przy rekurencyjnym wywołaniu procedury w przypadku rozgałęzienia warunkowego. Jeżeli wartość parametru jest równa NULL, procedura rozpoczyna przetwarzanie scenariusza od początku, czyli od zdania typu Precondition. W tym celu wywoływana jest procedura parsePreconditionSentence(), która implementuje regułę translacji P1 (patrz: górna część procedury na Rysunek 104).

Wszystkie pozostałe typy zdań (oprócz zdań typu ConditionSentence) przetwarzane są po kolei, zgodnie z kolejnością ich występowania w scenariuszach danego przypadku użycia. Jest to realizowane w pętli rozpoczynającej się od pobrania kolejnego zdania (getNextSentence()) i przypisania do zmiennej @sentence; licznik zdań @sentenceCounter jest inkrementowany. Następnie badany jest typ pobranego zdania (utl_GetSentenceDetailedType()) i – w zależności od typu – wywoływane są odpowiednie procedury przetwarzające zdanie, zgodnie z regułami translacji. Procedury te wymagają przekazania zdania konkretnego typu, stąd przy ich wywołaniu typ obiektu wskazywanego przez zmienną @sentence jest rzutowana z abstrakcyjnego typu ConstrainedLanguageSentence na jeden z typów konkretnych.

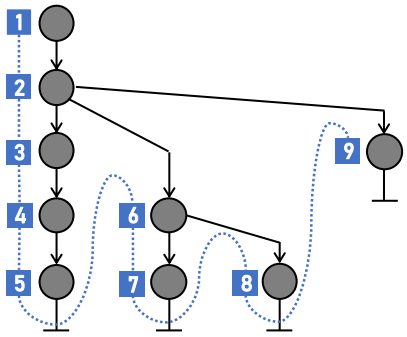
W przypadku zdań typu ConditionSentence uruchamiana jest procedura parseConditionSentence(), która spełnia dwa zadania. Po pierwsze – przetwarza zdania warunkowe zgodnie z regułami translacji. Po drugie – zapewnia, że wszystkie alternatywne scenariusze przypadku użycia powstałe w wyniku rozgałęzienia warunkowego, będą właściwie przetworzone i zostanie dla nich wygenerowany kod. Drugie zadanie realizowane jest poprzez rekurencyjne wywołanie procedury ParseUseCaseScenarios().



Rysunek 105 – Procedura ParseConditionSentence()

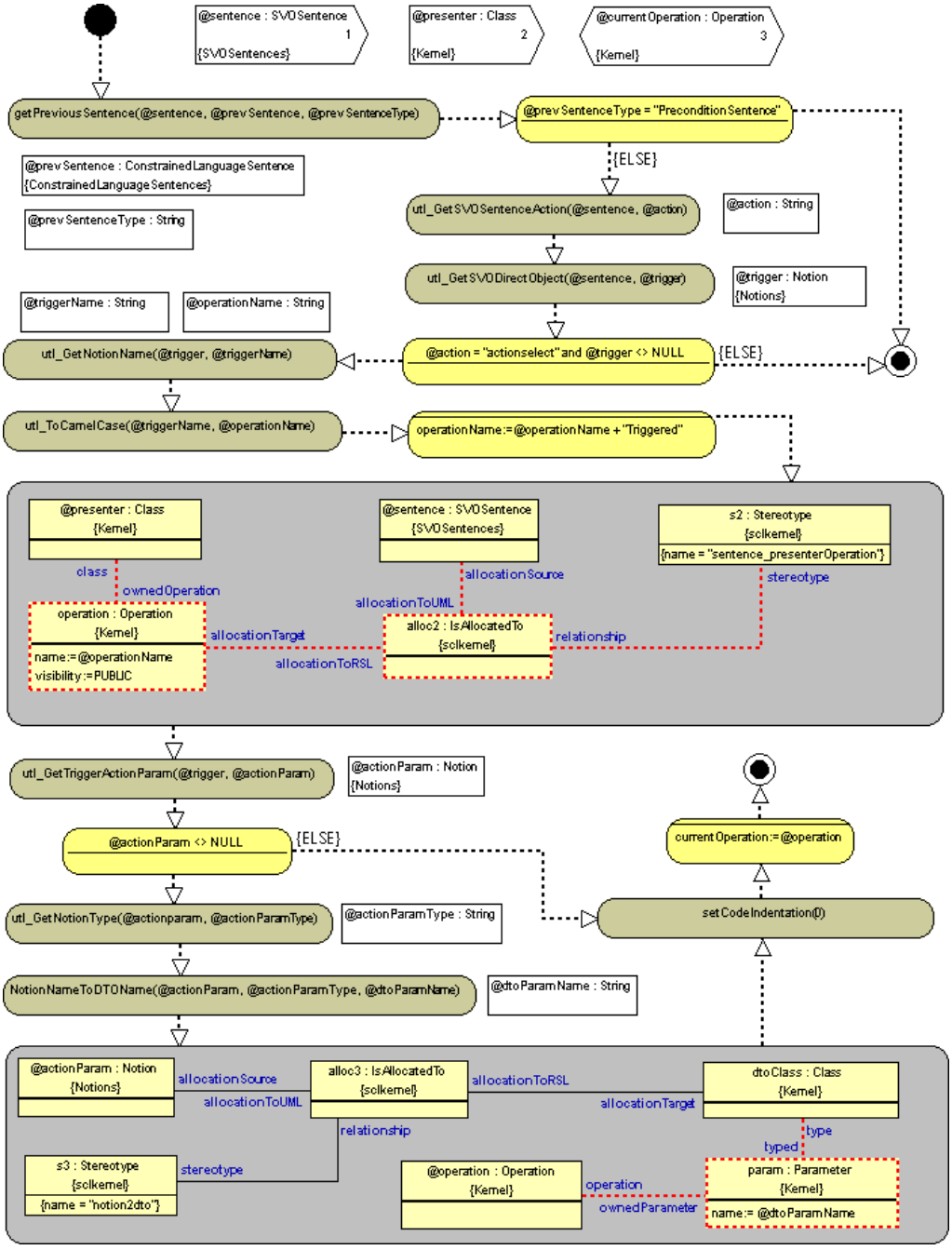
Procedura parseConditionSentence() zilustrowana jest na Rysunek 105. Pierwszym istotnym krokiem tej procedury jest pobranie zdania, które w scenariuszu poprzedza aktualnie przetwarzane zdanie warunkowe (@prevSentence). Zdanie to stanowi kontekst, od którego zależy sposób i miejsce generowania kodu dla zdania warunkowego, zgodnie z regułami translacji P11 do P14. Wywołanie procedur implementujących te reguły odbywa się w pętli for-each, w zależności od typu zdania poprzedzającego (@prevSentenceType). Warto tutaj przypomnieć, że warunkowe rozgałęzienie scenariusza w języku RSL oznacza utworzenie wielu instancji zdania warunkowego ConditionSentence – tylu, ile rozłącznych warunków zostanie zdefiniowanych w scenariuszu. Stąd, pętla for-each iteruje po wszystkich zdaniach warunkowych następujących po @prevSentence. Pozwala to wygenerować bloki instrukcji if-else uwzględniające wszystkie warunki. Kod generowany w tych blokach zależy, z kolei, od przebiegu scenariusza po każdym ze zdań warunkowych. Dlatego, po przetworzeniu zdania warunkowego w ramach danego przebiegu pętli, następuje rekurencyjne wywołanie procedury ParseUseCaseScenarios(), gdzie jako parametr @sentence przekazywane jest zdanie następujące bezpośrednio po zdaniu warunkowym. Dzięki temu pojedyncze wywołanie procedury ParseUseCaseScenarios() dla przypadku użycia pozwala wygenerować kod logiki aplikacji dla dowolnej struktury scenariuszy w ramach tego przypadku, zgodnie ze schematem przedstawionym na Rysunek 109.

Warto również zwrócić uwagę na procedurę pomocniczą getCodeIndentation(), która zwraca wartość globalnej zmiennej określającej poziom wcięcia kodu generowanego dla aktualnie przetwarzanego zdania. Procedura ta razem z dwiema dodatkowymi procedurami – increaseCodeIndentation() oraz decreaseCodeIndentation() – zapewniają, że wygenerowany kod (w szczególności bloki kodu w instrukcjach if-else) jest czytelnie sformatowany, zgodnie z dobrymi praktykami dla języka Java.



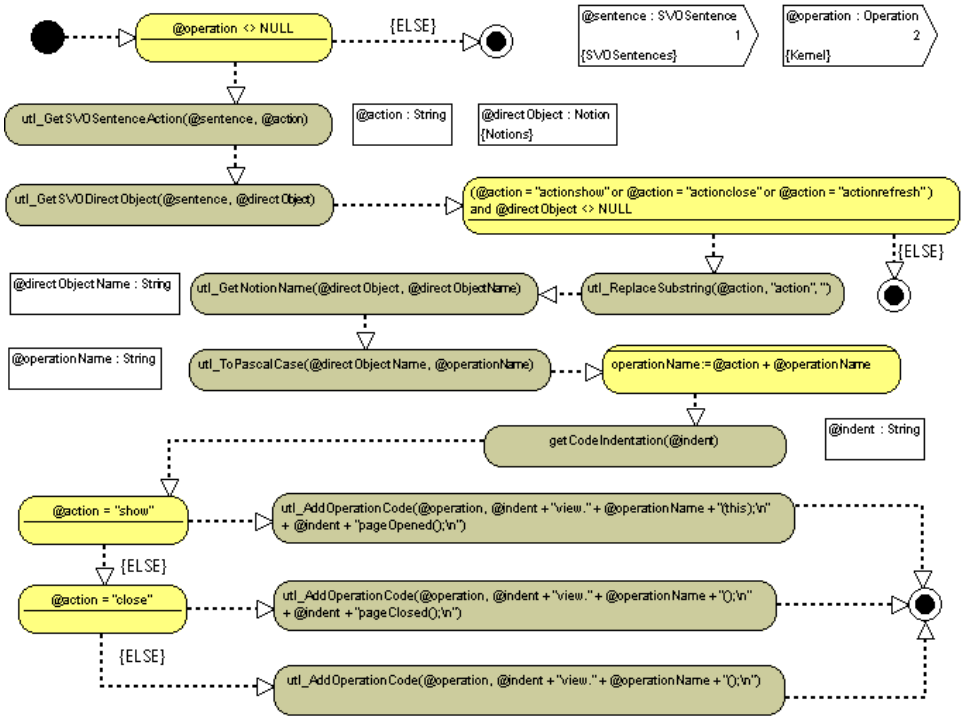
Rysunek 106 – Kolejność przetwarzania zdań scenariuszy przypadku użycia przez procedurę ParseUseCaseScenarios()

Dalsza część tego rozdziału przedstawia w szczegółach procedury wywoływane podczas parsowania scenariuszy przypadku użycia, czyli w ramach wykonania procedury ParseUseCaseScenarios(). Opisane poniżej procedury implementują reguły translacji dla trzech – najciekawszych z punktu widzenia obserwowalnego zachowania aplikacji – typów zdań: Actor-to-Trigger, System-to-Screen oraz Invoke.



Rysunek 107 – Procedura ParseActorToTriggerSentence()

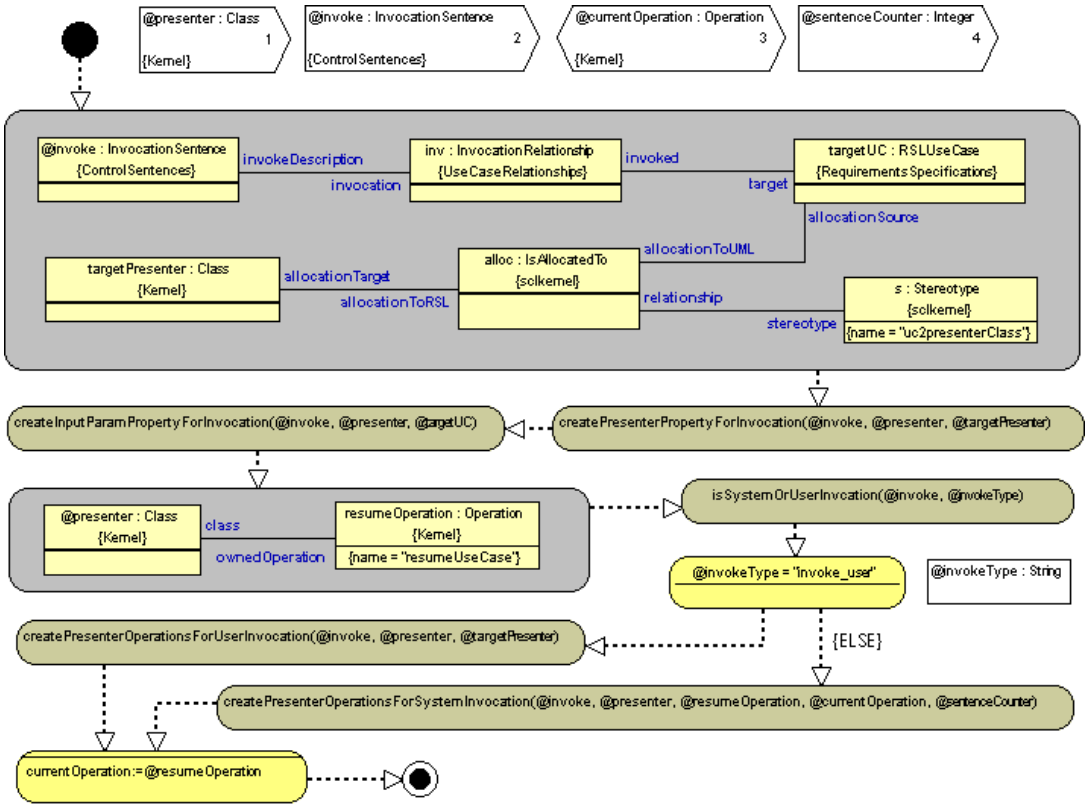
Rysunek 107 przedstawia procedurę parseActorToTriggerSentence(). Realizuje ona regułę semantyczną P4. Zgodnie z regułą, procedura sprawdza najpierw typ zdania poprzedzającego i kończy działanie jeżeli jest to zdanie typu Precondition. (w przyjętym środowisku translacyjnym, logika scenariusza następująca po zdaniu Actor-to-Trigger, realizowana jest w ramach metody invoke()). W przeciwnym wypadku, procedura tworzy w klasie prezentera operację, która będzie wywołana przez warstwę widoku w momencie wyzwolenia przez aktora akcji odpowiadającej zdaniu Actor-to-Trigger. W tym celu określana jest akcja odpowiadająca orzeczeniu zdania poprzez wywołanie utl_GetSVOSentenceAction(). W obecnej implementacji musi to być akcja typu „Select". Następnie określana jest nazwa operacji i uruchamiana reguła dodająca operację do klasy prezentera. Jeżeli wyzwalacz posiada relację „action param" (utl_GetTriggerActionParam()), do operacji dodawany jest parametr wywołania w postaci obiektu odpowiedniej klasy DTO. Jest to realizowane w ostatniej regule omawianej procedury. Utworzona operacja zwracana jest, poprzesz parametr @currentOperation, do procedury wywołującej, dzięki czemu kod generowany dla zdań następujących po aktualnym zdaniu Actor-to-Trigger umeszczany jest w tej operacji w ramach dalszego przebiegu procedury ParseUseCaseScenarios().



Rysunek 108 – Procedura ParseSystemToScreenSentence()

Zdania typu Actor-to-Trigger zmieniają stan dialogu z „aktor” na „system”. Zmiana odwrotna następuje w wyniku wykonania akcji wyrażonej zdaniem typu System-to-Screen. Do przetwarzania zdań tego typu służy procedura ParseSystemToScreenSentence() przedstawiona na Rysunek 108. Ta względnie prosta procedura generuje w klasie prezentera kod odwołujący się do warstwy widoku (poprzez interfejs `IView`) w celu wykonania operacji na ekranach prezentowanych użytkownikowi, zgodnie z akcją wyrażoną orzeczeniem zdania System-to-Screen. Tym samym realizuje ona częściowo regułę semantyczną P5. Parametrami wywołania procedury jest przetwarzane zdanie (@sentence) oraz operacja w której ma być umieszczony kod (@operation). Procedura sprawdza najpierw na jaką akcję mapuje się orzeczenie zdania (utl_GetSVOSentenceAction()) i na tej podstawie ustala nazwę operacji interfejsu `IView`, która ma być wywołana (@operationName). Obecna implementacja ogranicza się do standardowych akcji „show”, „close” i „refresh”. Następnie, do ciała metody @operation dodawany jest kod wywołujący operację interfejsu `IView`. W przypadku akcji „show” i „close” po wywołaniu operacji interfejsu następuje wywołanie lokalnej metody prezentera, odpowiednio `pageOpened()` i `pageClosed()`, w celu modyfikacji licznika otwartych ekranów aplikacji. Rola takiego licznika wyjaśniona została w opisie środowiska translacyjnego w rozdziale 5.1. W celu czytelnego formatowania, dodawany kod uwzględnia aktualny poziom wcięcia pobierany za pomocą procedury getCodeIndentation(). Omawiana procedura nie uwzględnia ostatniej części reguły P5 (wyrażonej w podpunkcie c), która dotyczy klasy reprezentującej ekran interfejsu użytkownika. Ta część reguły zaimplementowana jest w części transformacji generującej kod warstwy widoku.

Najciekawszymi z punktu widzenia logiki aplikacji są zdania typu Invoke. Sposób ich translacji na kod docelowy opisują reguły semantyczne P2 i P3. Transformacja RSL to Java implementuje te reguły w postaci kilku procedur. Procedurą od której rozpoczyna się przetwarzanie zdania typu Invoke jest ParseInvocetionSentence() przedstawiona na Rysunek 109. Jest ona wywoływana z poziomu procedury ParseUseCaseScenarios() w momencie napotkania w scenariuszu zdania typu Invoke (patrz: Rysunek 104). Pierwszym krokiem wywoływanej procedury jest wykonanie reguły, która znajduje wywoływany przypadek użycia (@targetUC), na który wskazuje aktualnie przetwarzane zdanie „Invoke”. Jest to możliwe dzięki relacji „InvokeRelationship łączącej zdanie „Invoke” z docelowym przypadkiem użycia (patrz: fragment metamodelu języka RSL przedstawiony na Rysunek 42). Reguła znajduje również klasę prezentera (@targetPresenter) utworzoną wcześniej dla docelowego przypadku użycia.

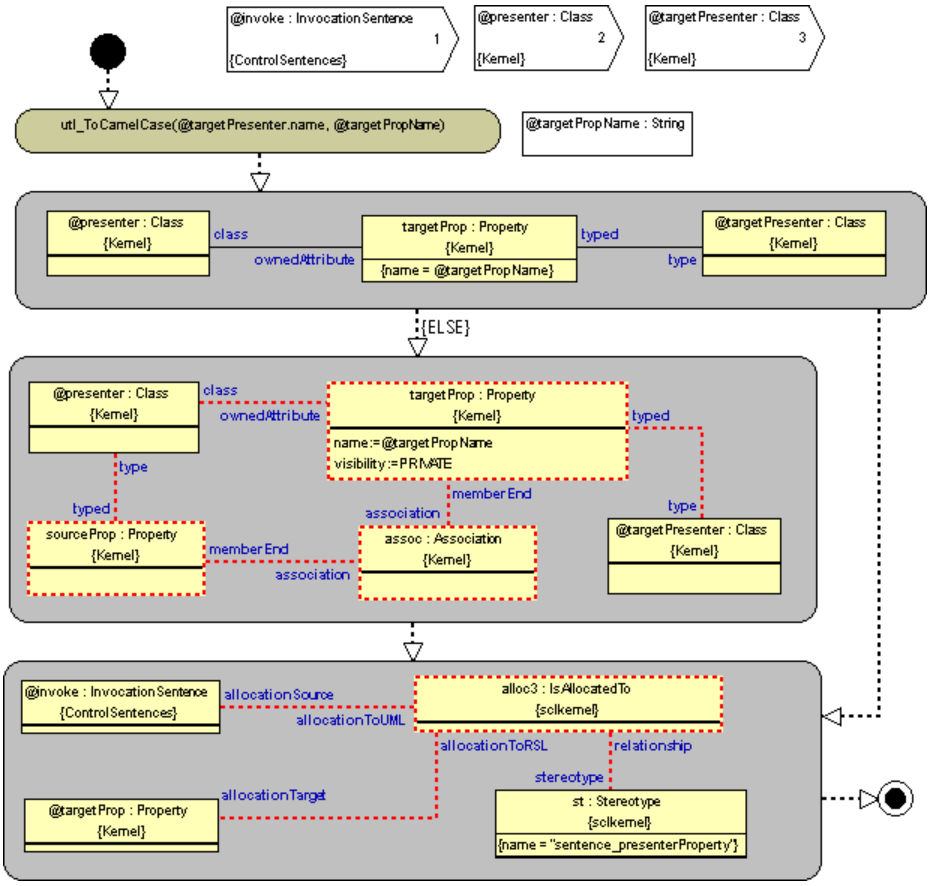


Rysunek 109 – Procedura ParseInvocationSentence()

W kolejnym kroku, w klasie prezentera przypadku wywołującego, tworzone są atrybuty zgodnie z regułami translacji P2 i P3, wyrażonymi w podpunktach a oraz b tych reguł. Jest to realizowane przez dwie procedury: createPresenterPropertyForInvocation() oraz createInputParamPropertyForInvocation().

Pierwsza z nich, przedstawiona na Rysunek 110, tworzy atrybut wskazujący obiekt prezentera przypadku wywoływanego. Procedura sprawdza najpierw, czy taki atrybut nie został już utworzony podczas przetwarzania innego zdania „Invoke". Jeżeli nie – klasa prezentera jest rozszerzana o taki atrybut. Warto zauważyć, że dla atrybutu tworzona jest dodatkowo asocjacja (metaklasa Association). Nie ma ona wpływu na docelowy kod, ale stanowi ważną informację o relacji między dwiema klasami prezenterów w modelu klas UML generowanym w wyniku wykonania transformacji. Informacja ta może być zwizualizowana na diagramie klas stanowiącym dokumentację systemu. Ostatnia reguła tworzy zależność identyfikowalności (IsAllocatedTo), z odpowiednim stereotypem, pomiędzy przetwarzanym zdaniem oraz utworzonym atrybutem. Relacja ta jest tworzona również wtedy, gdy atrybut istniał już wcześniej. Może istnieć, zatem, wiele elementów specyfikacji wymagań powiązanych z jednym elementem modelu docelowego. Relacje te są wykorzystywane w innych miejscach

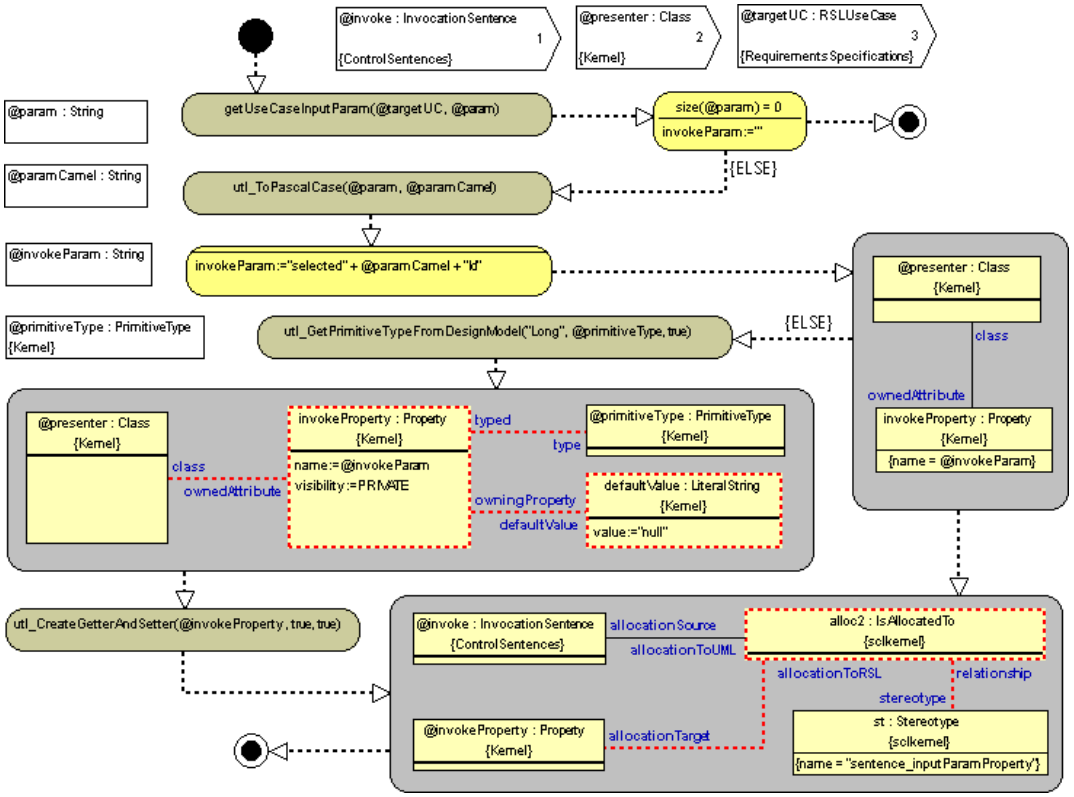
transformacji do łatwego zidentyfikowania elementów, które zostały już wygenerowane dla przetwarzanych elementów języka RSL.



Rysunek 110 – Procedura CreatePresenterPropertyForInvocation()

Drugą z procedur tworzących atrybuty prezentera jest, przedstawiona na Rysunek 111, procedura createInputParamPropertyForInvocation(). Sprawdza ona najpierw czy wywoływany przypadek użycia (@targetUC) wymaga parametru wejściowego. Do pobierania parametru wejściowego służy procedura pomocnicza getUseCaseInputParam(). Analizuje ona zdanie „Precondition" przekazanego przypadku użycia, które może zawierać informację o wymaganym parametrze wejściowym. Istnienie takiego parametru przekłada się, w wygenerowanym kodzie, na parametr metody invoke() w klasie prezentera danego przypadku użycia. Opisuje to reguła semantyczna P1. Stąd, klasa prezentera przypadku wywołującego musi posiadać atrybut, którego wartość jest przekazywana podczas wywoływania metody invoke() prezentera przypadku wywoływanego.

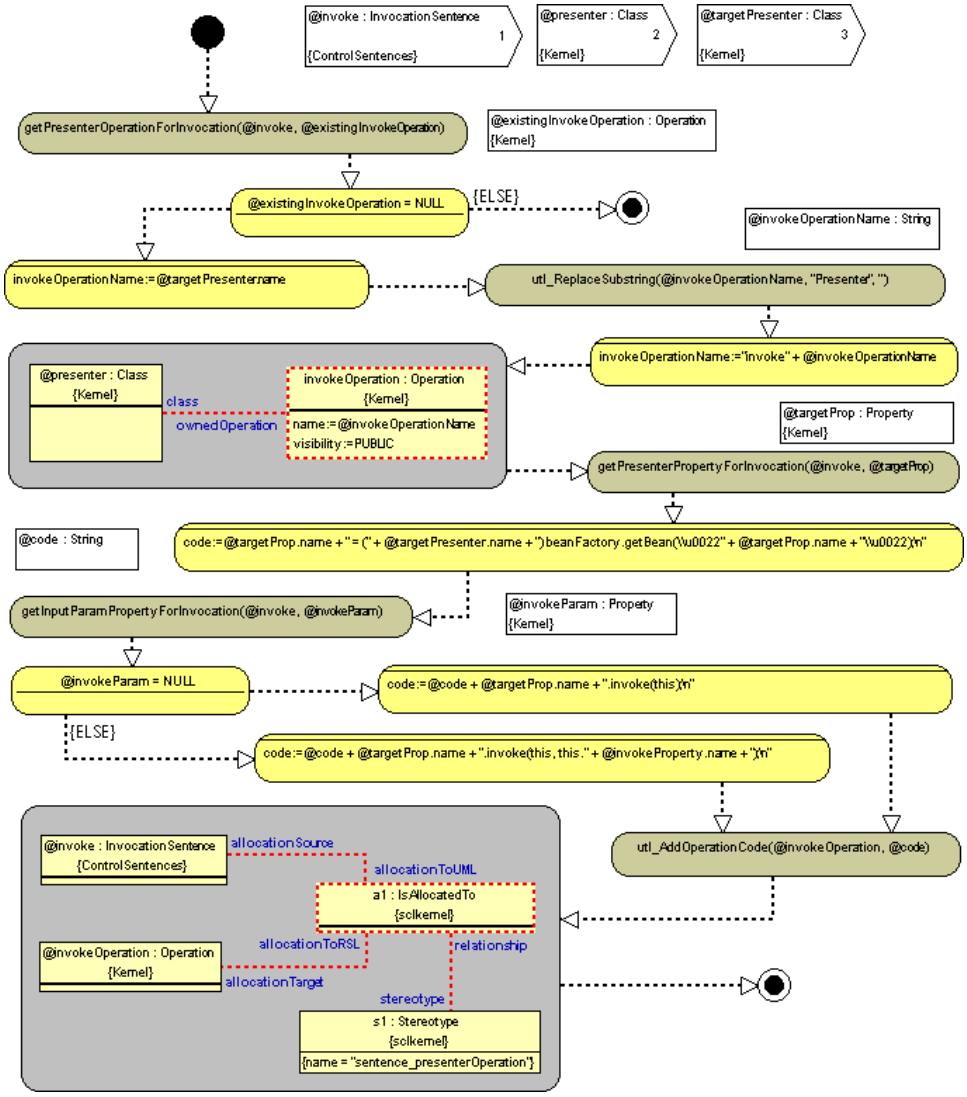
Jeżeli parametr wejściowy istnieje, procedura createInputParamPropertyForInvocation() ustala jego nazwę, zgodnie z regułą semantyczną, a następnie dodaje adekwatny prywatny atrybut (@invokeProperty) do klasy prezentera. Dodatkowo, tworzona jest para standardowych operacji do odczytu i modyfikacji wartości atrybutu. Na koniec, tworzona jest relacja identyfikowalności łącząca przetwarzane zdanie „Invoke" z utworzonym atrybutem. Relacja ta, podobnie jak w procedurze opisywanej wcześniej, tworzona jest również wtedy, gdy atrybut @invokeProperty został utworzony już wcześniej.



Rysunek 111 – Procedura CreateInputParamPropertyForInvocation()

Wracając do procedury ParseInvocationSentence(), po utworzeniu niezbędnych atrybutów, procedura przechodzi do utworzenia operacji w klasie prezentera oraz kodu tych operacji, dla przetwarzanego zdania „Invoke". Elementy te, zgodnie z regułami P2 i P3, zależą od stanu dialogu dla danego zdania „Invoke". Procedura sprawdza więc stan dialogu poprzez wywołanie isSystemOrUserInvocation() i uruchamia jedną z procedur: createPresenterOperationsForUserInvocation() – w przypadku, gdy stan dialogu wskazuje na aktora, bądź createPresenterOperationsForSystemInvocation() – w przeciwnym wypadku. Wcześniej jednak, procedura wyszukuje operację resumeUseCase() utworzoną wcześniej w klasie prezentera. Jest ona jednym z parametrów procedury

createPresenterOperationsForSystemInvocation() a także zwracana jest do procedury ParseUseCaseScenarios() (Rysunek 104), jako operacja bieżąca (@currentOperation) po przetworzeniu danego zdania „Invoke” w scenariuszu.

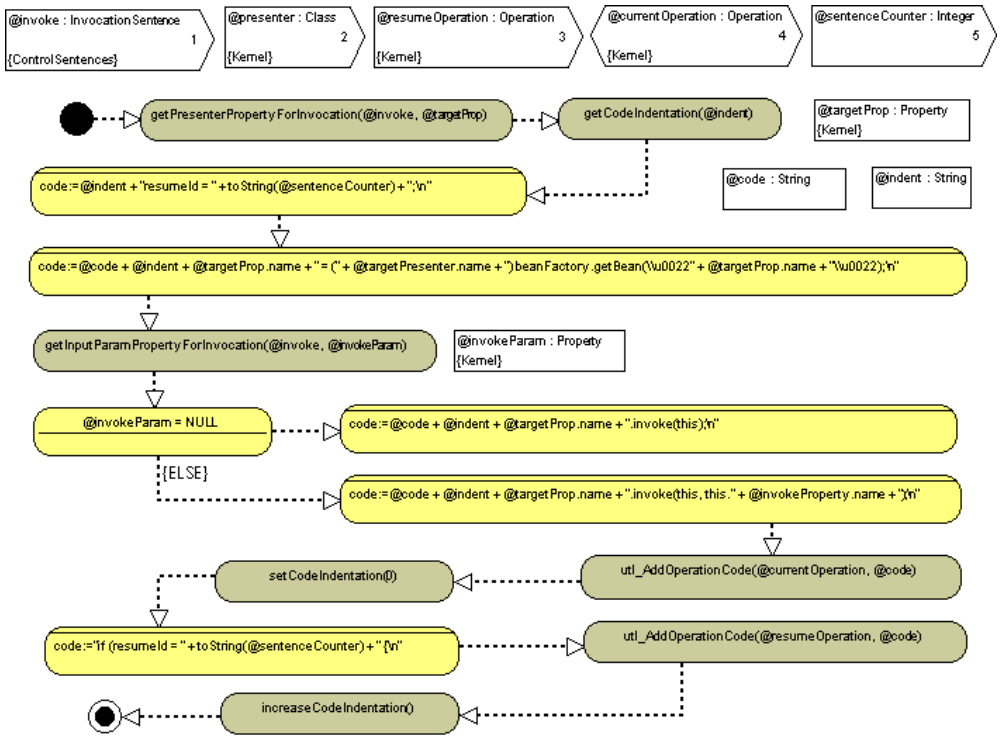


Rysunek 112 – Procedura CreatePresenterOperationsForUserInvocation()

Procedura createPresenterOperationsForUserInvocation() przedstawiona jest na Rysunek 112. Procedura sprawdza najpierw czy w klasie prezentera istnieje już metoda uruchamiająca logikę wywoływanego przypadku użycia poprzez wywołanie procedury getPresenterOperationForInvocation(). Jeżeli taka metoda nie istnieje – jest tworzona. Następnie do ciała metody dodawany jest kod wywołania logiki docelowego przypadku użycia, czyli wywołanie metody invoke() na obiekcie prezentera tego przypadku. Jeżeli wywoływany przypadek wymaga parametru wejściowego (getInputParamPropertyForInvocation()), jako

drugi parametr wywołania metody `invoke()` przekazywana jest wartość atrybutu utworzonego wcześniej w ramach procedury createInputParamPropertyForInvocation().

Kod wywołujący przypadek docelowy poprzedzony jest wierszem kodu, który pobiera obiekt prezentera docelowego za pomocą metody `getBean()`. Kod ten wynika z zastosowania w środowisku docelowym kontenera IoC (ang. Inversion of Control) ułatwiającego przekazywanie instancji klas prezenterów przypadków użycia.



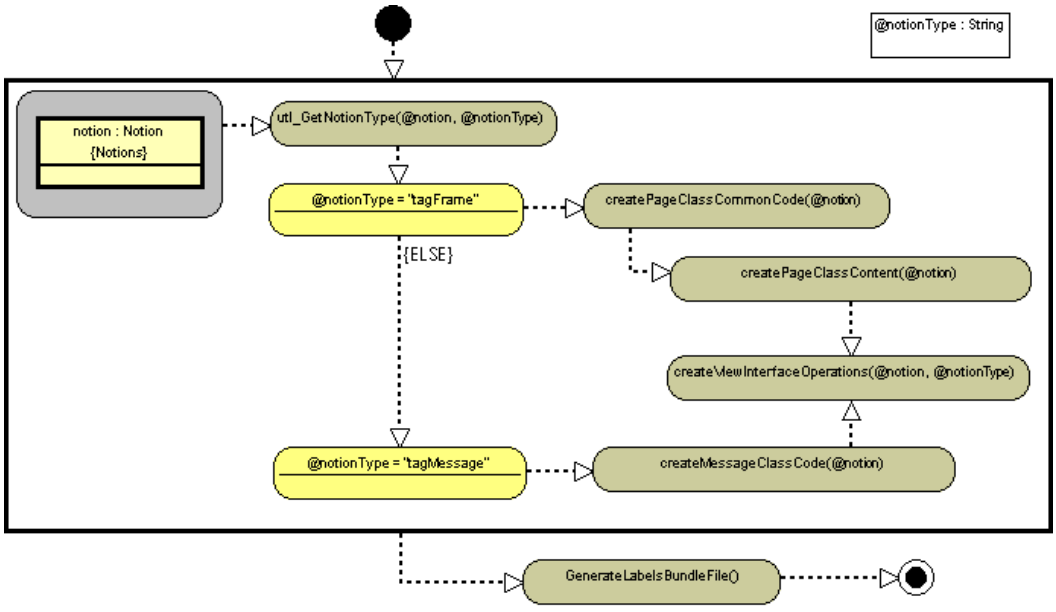
Rysunek 113 – Procedura CreatePresenterOperationsForSystemInvocation()

Rysunek 113 przedstawia procedurę tworzącą kod dla zdań „Invoke” wywoływanych przez system. Procedura CreatePresenterOperationsForSystemInvocation() implementuje regułę semantyczną P3 (podpunkty c i d). Kod wywołania logiki przypadku docelowego jest analogiczny jak w przypadku procedury omówionej wcześniej, przy czym jest on poprzedzony ustawieniem kontekstu wywołania, czyli przypasaniem atrybutowi `resumeId` numeru zdania „Invoke”. Kod umieszczany jest w metodzie @currentOperation przekazanej jako jeden z parametrów wywołania procedury. Dodatkowo, zgodnie z regułą semantyczną, w metodzie `resumeUseCase()` (przekazanej również jako parametr wywołania @resumeOperation) tworzony jest blok warunkowy `if` umożliwiający powrót sterowania po wykonaniu logiki wywoływanego przypadku użycia.

6.6 Generowanie kodu w warstwie widoku

Generator kodu dla warstwy widoku implementuje 11 reguł semantycznych opisanych w rozdziale 5.5. Ta część transformacji „RSL to Java” składa się z ponad 30 procedur w języku MOLA, z których większość jest mocno rozbudowana ze względu ilość kodu, jaką należy wygenerować w warstwie widoku, aby zaimplementować w pełni działający interfejs użytkownika. Jest to złożoność technologiczna, wynikająca z zastosowania konkretnego środowiska Echo3. Dla zwięzłości opisu, poniżej omówione zostały wybrane procedury, przy czym opis skupia się bardziej na sposobie implementacji najważniejszych reguł translacji a mniej na aspektach technologicznych wynikających z zastosowania wspomnianego środowiska.

Wszystkie reguły translacji do warstwy widoku, z wyjątkiem reguły V9, operują na pojęciach dziedzinowych oraz powiązaniach między nimi. Generowanie kodu dla warstwy widoku rozpoczyna się więc od pętli w procedurze GenerateView() (Rysunek 114), która iteruje po wszystkich elementach typu Notion i dla trzech podstawowych typów pojęć z dziedziny aplikacji, tj. Screen, Message oraz Confirmation, wywołuje odpowiednie podprocedury odpowiedzialne za wygenerowanie modelu klas dla tych pojęć i ich powiązań.



Rysunek 114 – Procedura GenerateView()

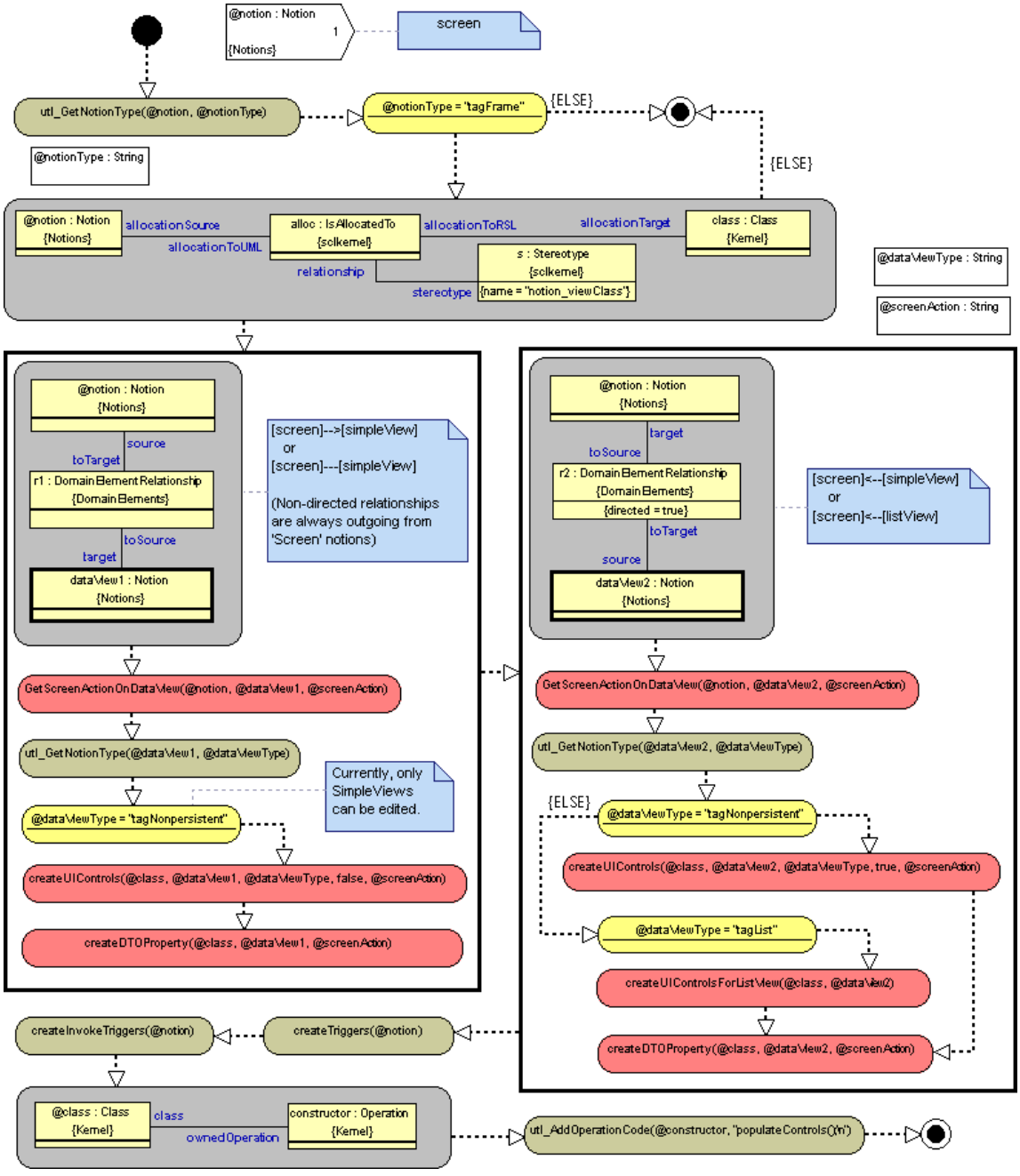
Jeżeli pojęcie jest typu „Screen” (procedura utl_GetNotionType() zwraca wartość „tagFrame”), to w pierwszej kolejności wywoływana jest podprocedura createPageClassCommonCode(). Generuje ona podstawowe atrybuty i metody, jakie musi implementować każda klasa ekranowa, dziedziczącą z generycznej klasy ekranowej środowiska Echo3 (`netapp.echo.app.ContentPane`) oraz implementującą interfejs `nextapp.echo.app.event.ActionListener` w celu zapewnienia zdolności obsługi zdarzeń ekranowych. Ponadto, generowany jest kod zapewniający standardowy układ kontrolek ekranowych (`nextapp.echo.app.layout.GridLayout`).

Następnie, wywoływana jest podprocedura createPageClassContent(). Jej zadaniem jest wygenerowanie kompletnego kodu klas ekranowych, z uwzględnieniem wszystkich elementów interfejsu użytkownika wynikających z modelu RSL. Sposób działania tej procedury został opisany w dalszej części rozdziału.

Na końcu uruchamiana jest procedura createViewInterfaceOperations(), która tworzy w interfejsie `IView`, oraz implementującej go klasie `ViewImpl`, metody pozwalające warstwie prezentera operować na utworzonej klasie ekranowej. Procedura ta implementuje reguły semantyczne V1 oraz V2.

Podobnie jest w przypadku pojęć typu „Message” (procedura utl_GetNotionType() zwraca wartość „tagMessage”) – najpierw tworzony jest kompletny kod klasy, zgodnie z regułami V10 oraz V11, a następnie interfejs `IView` rozszerzany jest o adekwatne operacje. Należy tutaj zaznaczyć różnicę między regułami semantycznymi a implementacją transformacji „RSL to Java”. Semantyka translacyjna definiuje osobne reguły dla pojęć typu „Message” oraz „Confirmation”. W transformacji – dla uproszczenia – wykorzystywane jest tylko pojęcie typu „Message”, które może reprezentować oba typu okien dialogowych. Jeśli w scenariuszu po zdaniu odnoszącym się do pojęcia „Message” następuje rozgałęzienie warunkowe, pojęcie jest traktowane jak „Confirmation”.

Po przetworzeniu wszystkich pojęć w pętli, wywoływana jest jeszcze procedura GenerateLabelsBundleFile(). Generuje ona plik tekstowy klucz-wartość z etykietami ekranowymi elementów interfejsu użytkownika, m.in. przycisków, nagłówków kolumn, statycznych pól tekstowych, itp. Dzięki temu wszelkie etykiety i komunikaty nie muszą być literalnie umieszczone bezpośrednio w kodzie klas i mogą być w łatwy sposób edytowane bez konieczności rekompilacji kodu.



Rysunek 115 – Procedura CreatePageClassContent()

Przyjrzyjmy się procedurze CreatePageClassContent() przedstawionej na Rysunek 115. Jej zadaniem (wraz z wszystkimi wywoływanymi przez nią podprocedurami) jest zidentyfikowanie relacji pomiędzy pojęciami reprezentującymi ekrany aplikacji a pojęciami z dziedziny problemu i wygenerowanie na tej podstawie kodu klas ekranowych, w szczególności odpowiednich kontrolek ekranowych, a także kodu klas pomocniczych implementujących niezbędne w środowisku Echo3 klasy abstrakcyjne (służące np. do

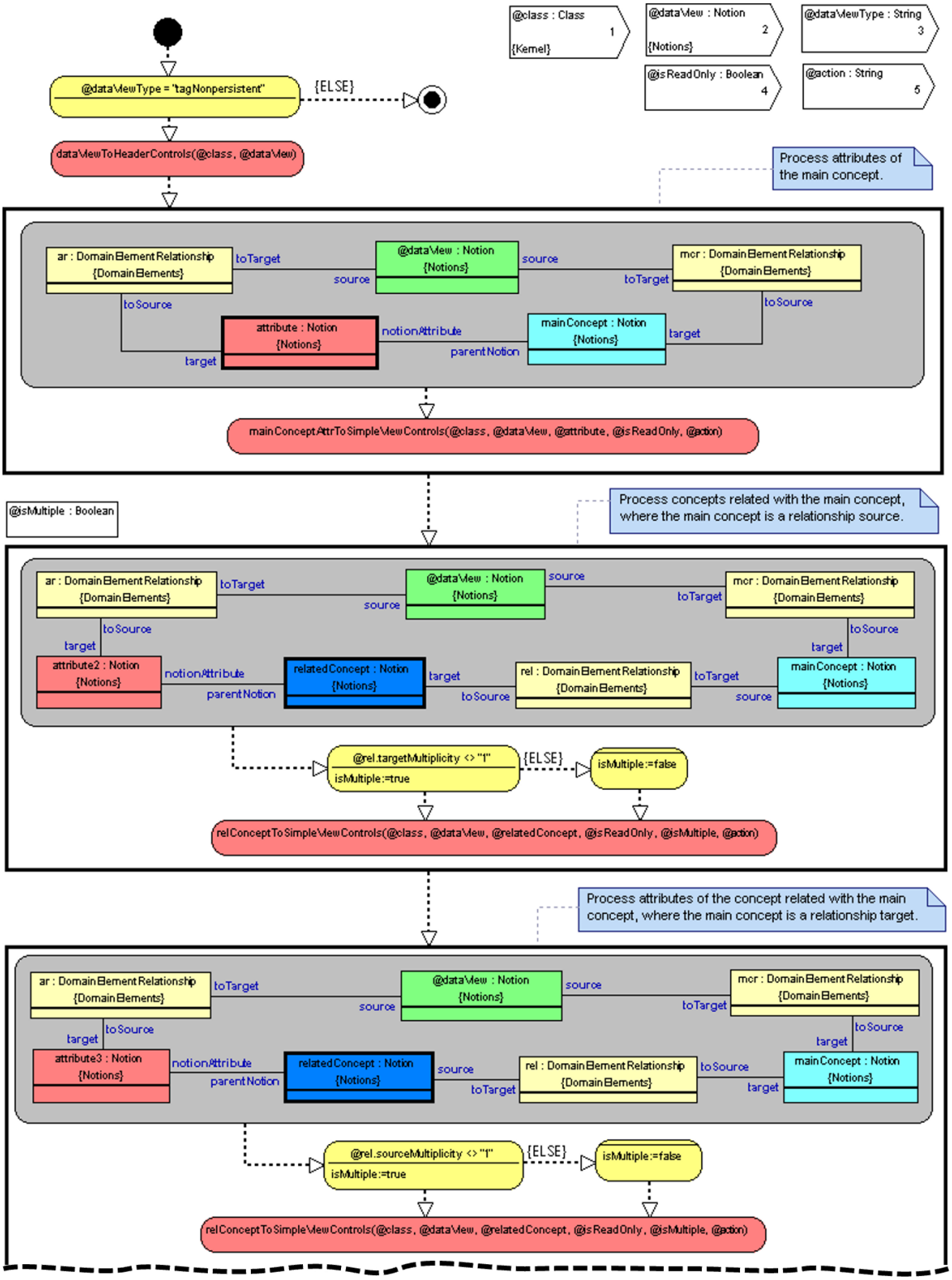
prezentacji danych w formie tabelarycznej, jak `nextapp.echo.app.table.AbstractTableModel` czy `nextapp.echo.app.table.TableCellRenderer`). Ta część transformacji implementuje reguły semantyczne V3 – V8.

Zasadniczą część procedury CreatePageClassContent() tworzą dwie pętle for-each. Obie pętle iterują po widokach danych, dla których istnieje powiązanie z pojęciem typu Screen przekazanym jako parametr wywołania procedury (@notion). Reguła w pierwszej pętli bierze pod uwagę te relacje, których źródłem jest pojęcie ekranowe a celem widok danych; reguła w drugiej pętli – odwrotnie. Źródło oraz kierunkowość relacji determinuje czy dany ekran aplikacji służy do prezentacji czy edycji danych (patrz: rozdział 3.2.2). Od tego zależy w jaki sposób wygenerowany zostanie kod kontrolek ekranowych. Dokładny typ relacji („CREATE", „READ" lub „UPDATE") jest wyznaczany poprzez wywołanie procedury pomocniczej GetScreenActionOnDataView() i przechowywany w zmiennej @screenAction na potrzeby wywoływanych podprocedur. Sprawdzany jest też szczegółowy typ pojęcia @dataView za pomocą procedury utl_GetNotionType()[31]. Wartości ta determinują jakie procedury generujące szczegółowy kod klas ekranowych oraz z jakimi wartościami parametrów wejściowych zostaną wywołane:

- createUIControls() – generuje kompletny kod kontrolek ekranowych dla pojęć typu Simple View, na podstawie powiązań tych pojęć z atrybutami pojęć dziedzinowych; kontrolki mogą być generowane zarówno w trybie do edycji (wywołanie w pierwszej pętli procedury CreatePageClassContent()), jak też w trybie tylko do odczytu (wywołanie w pierwszej pętli);
- createUIControlsForListView() – generuje kompletny kod kontrolek ekranowych dla pojęć typu List View (w postaci tabeli), na podstawie powiązań tych pojęć z atrybutami pojęć dziedzinowych; w obecnej implementacji transformacji, tabele mogą być generowane tylko do odczytu, stąd procedura ta wywoływana jest tylko w drugiej pętli procedury CreatePageClassContent();
- createDTOProperty() – generuje atrybut klasy ekranowej wskazujący na obiekt lub listę obiektów typu DTO, które służą do przekazywania danych między warstwą widoku

[31] Dla pojęć typu Simple View procedura utl_GetNotionType() zwraca wartość „tagNonpersistent" a dla pojęć typu List View – „tagList"

i prezentera; procedura wywoływana jest zarówno dla pojęć typu Simple View jak i List View i implementuje regułę V6.

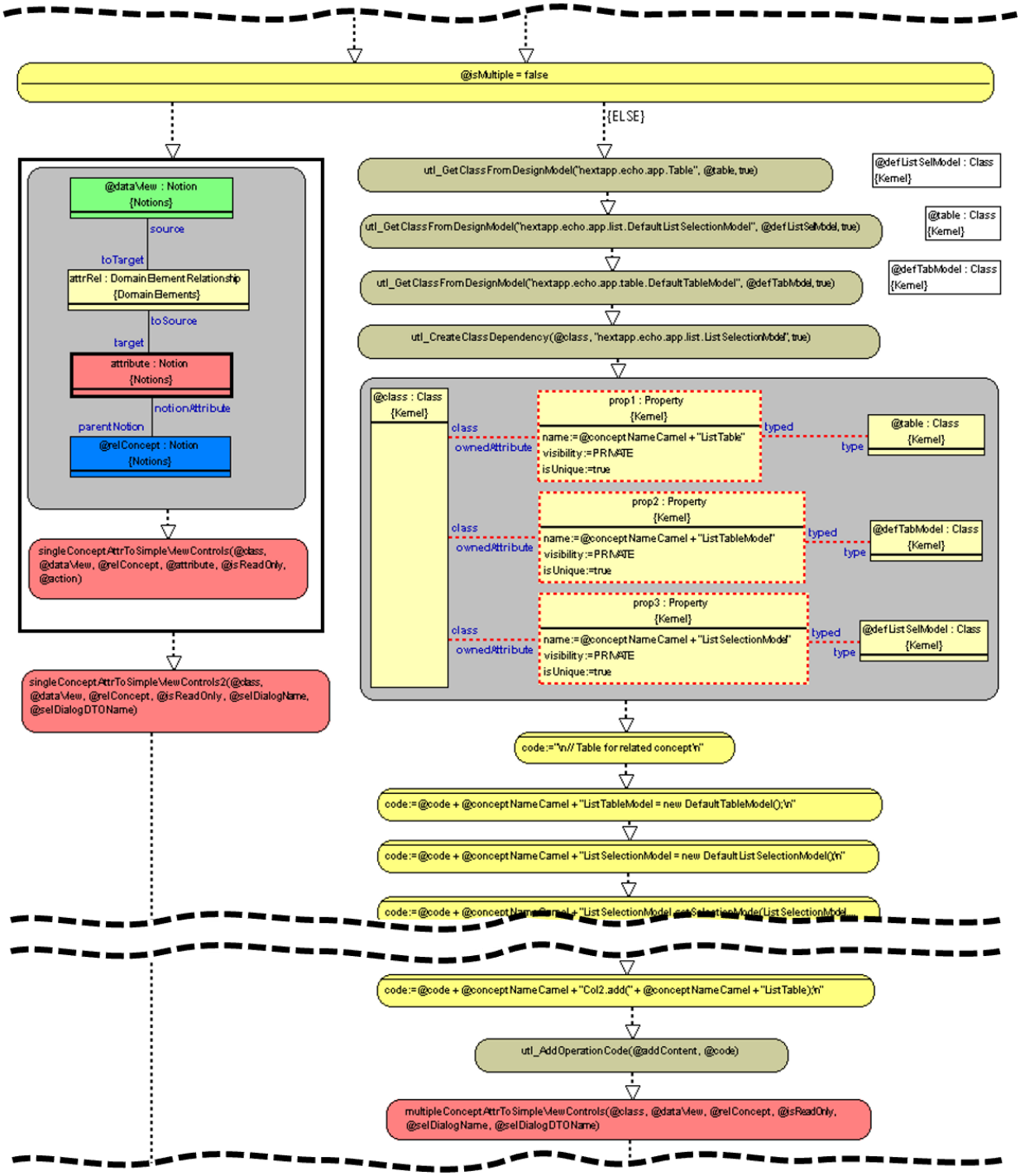


Rysunek 116 – Procedura CreateUIControls()

Rysunek 116 przedstawia fragment procedury CreateUIControls() wywoływanej, jak wspomniano powyżej, dla pojęć typu Simpe View. Każdy taki element w modelu RSL grupuje dane prezentowane na docelowym ekranie aplikacji poprzez wskazanie wybranych atrybutów pojęć dziedzinowych. Do właściwego utworzenia kontrolek ekranowych odpowiadających tym atrybutom, istotne są powiązania między pojęciami dziedzinowymi zawierającymi wskazywane atrybuty a także krotności tych powiązań (patrz: reguły V3 oraz V4). Dlatego też, głównym zadaniem procedury CreateUIControls() jest zidentyfikowanie pojęć dziedzinowych powiązanych w określony sposób i wywołanie odpowiednich podprocedur generujących docelowy kod dla atrybutów tych pojęć.

Pierwsza pętla widoczna na Rysunek 116 iteruje po atrybutach (@attribute) wskazywanych przez aktualnie przetwarzany widok danych (@dataView), ale tylko tych, które należą do pojęcia „main concept" przetwarzanego widoku danych (@mainConcept). Dla każdego takiego atrybutu wywoływana jest procedura mainConceptAttrToSimpleViewControls(), której zadaniem jest umieszczenie w odpowiedniej klasie ekranowej (@class) kodu kontrolki, wygenerowanego zgodnie z zasadami mapowania zdefiniowanymi w regule V3.

Kolejne dwie pętle widoczne na Rysunek 116, iterują po pojęciach dziedzinowych (@relatedConcept) powiązanych z pojęciem „main concept" dla aktualnie przetwarzanego widoku danych. Reguła wybiera tylko te pojęcia, których atrybuty wskazywane są przez widok danych. Dla każdego pojęcia dziedzinowego zgodnego z regułą wywoływana jest procedura relConceptToSimpleViewControls(). Jej zadaniem jest wygenerowanie kodu kontrolek odpowiadającego wszystkim atrybutom danego pojęcia dziedzinowego, które są wskazywane przez widok danych. Zgodnie z regułami V3 i V4, przy generowaniu kodu kontrolek uwzględniana jest krotność relacji między pojęciem dziedzinowym a pojęciem „main concept". W przypadku kiedy krotność po stronie relacji jest równa „1", każdemu atrybutowi odpowiada kontrolka pozwalająca wyświetlać lub edytować pojedynczą wartość. W przypadku krotności większej niż „1", atrybuty wskazywane przez widok danych prezentowane są w formie tabeli. Każdy wiersz tabeli prezentuje wartości atrybutów pojedynczej instancji pojęcia dziedzinowego, do którego te atrybuty należą. Fragment procedury relConceptToSimpleViewControls() widoczny na Rysunek 117, pokazuje dwie alternatywne ścieżki generowania kodu dla atrybutów, wykonywane w zależności od wartości parametru @isMultiple przekazanego przy wywołaniu procedury.



Rysunek 117 – Procedura relConceptToSimpleViewControls()

7 Studium przypadku

Rozdział ten opisuje stadium przypadku wykonane w celu weryfikacji i demonstracji działania transformacji „RSL to Java". Założeniem studium było opracowanie względnie prostej lecz w miarę kompletnej aplikacji, której specyfikacja wymagań pozwalałaby zweryfikować w praktyce wszystkie zaimplementowane reguły translacji. W ramach studium przypadku została wykonana aplikacja Pet Clinic – aplikacja realizująca podstawową funkcjonalność dla przychodni weterynaryjnej. Aplikacja Pet Clinic zyskała popularność jako oficjalny przykład demonstrujący sposób budowy aplikacji internetowych w środowisku Spring, dystrybuowany razem z wczesnymi wersjami tego środowiska [165] [166]. Przykład ten jest nadal chętnie wykorzystywany do demonstrowania różnych technologii tworzenia aplikacji[32], gdyż realizuje zwięzłą, dobrze określoną dziedzinę problemu a jego zakres funkcjonalny uwzględnia typowe dla większości aplikacji biznesowych interakcje użytkownika z systemem za pomocą formularzy, list czy okienek dialogowych.

Aplikacja Pet Clinic została wykonana w następujących krokach:

1. stworzenie specyfikacji wymagań w edytorze języka RSL w narzędziu ReDSeeDS,
2. uruchomienie transformacji „RSL to Java",
3. utworzenie pustego projektu aplikacji internetowej w środowisku programistycznym NetBeans wraz z zaimportowaniem niezbędnych bibliotek środowiska Echo3 oraz Spring,
4. import wygenerowanego kodu do utworzonego projektu,
5. skompilowanie i uruchomienie aplikacji.

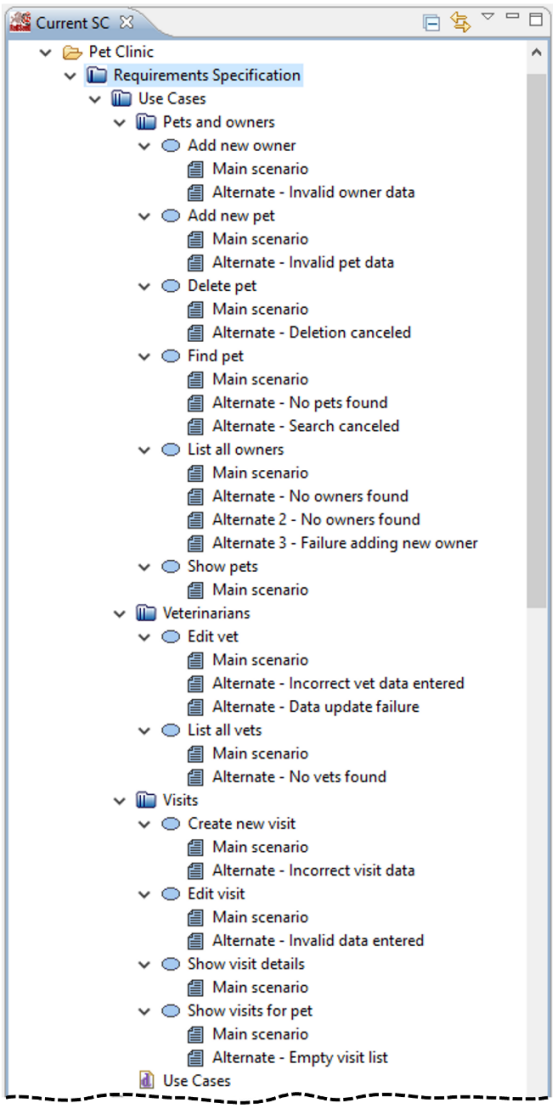
7.1 Specyfikacja wymagań w języku RSL

Struktura modelu źródłowego dla aplikacji Pet Clinic przedstawiona jest na Rysunek 118. Jest zgodna ze specyfikacją języka RSL oraz konwencją narzuconą przez transformację „RSL to Java" (patrz: rozdział 3.1.2). Model przypadków użycia (Rysunek 118a) podzielony jest na trzy podpakiety: „Pets and owners", „Veterinarians" oraz „Visits". Odzwierciedlają one trzy obszary funkcjonalne. W taki sam sposób podzielony został pakiet „Data model" (Rysunek 118b), gdzie każdy z podpakietów zawiera pojęcia z dziedziny problemu powiązane z danym

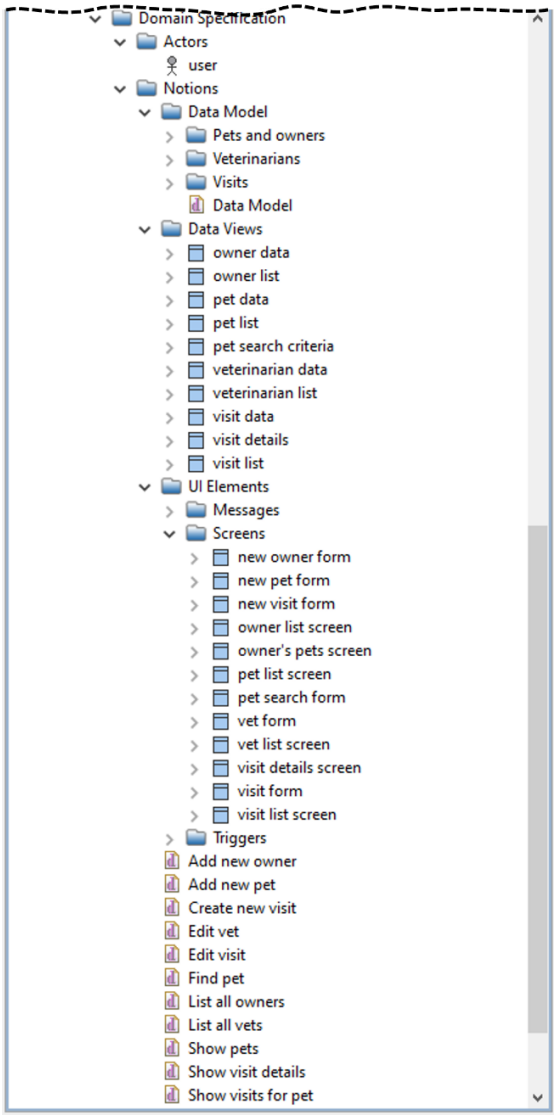
[32] https://spring-petclinic.github.io/

obszarem funkcjonalnym. Dla zwiększenia czytelności modelu, elementy w pakiecie „UI Elements” zostały umieszczone w podpakietach odpowiadających typom tych elementów: „Messages”, „Screens” oraz „Triggers”.

a)



b)



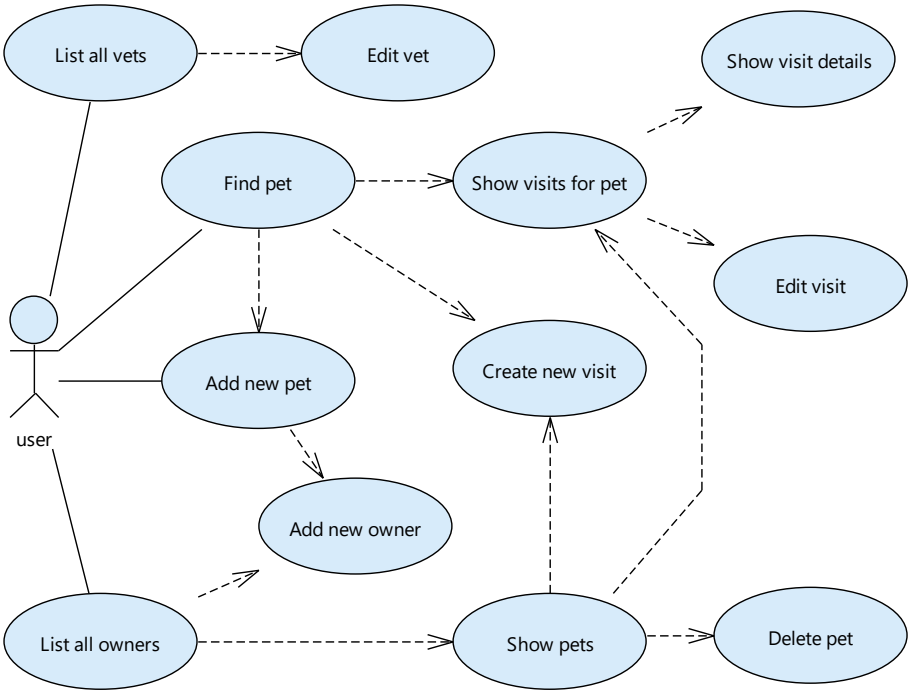
Rysunek 118 – Struktura specyfikacji wymagań dla systemu Pet Clinic w narzędziu ReDSeeDS: (a) specyfikacja wymagań, (b) model dziedzinowy

Logika aplikacji została zdefiniowana za pomocą dwunastu przypadków użycia oraz ich scenariuszy wylistowanych na Rysunek 118a. Relacje między tymi przypadkami użycia prezentuje natomiast diagram na Rysunek 119. Cztery z tych przypadków powiązane są bezpośrednio z aktorem relacją „use”, co oznacza, że reprezentowana przez nie funkcjonalność dostępna jest dla użytkownika bezpośrednio z głównego ekranu aplikacji w postaci czterech przycisków. Szereg relacji „invoke” pomiędzy wspomnianymi czterema a pozostałymi

przypadkami użycia odzwierciedla sposób w jaki użytkownik może nawigować w ramach całej aplikacji. Szczegółowy sposób nawigacji określają scenariusze przypadków użycia. Warto zauważyć, że te same przypadki użycia mogą być wywoływane z różnych miejsc aplikacji, tzn. w toku wykonywania wielu różnych przypadków użycia. Przykładowo, nową wizytę dla zwierzaka („Create new visit") można utworzyć poprzez wskazanie konkretnego zwierzaka i kliknięcie przycisku tworzenia wizyty – zarówno z poziomu listy zwierzaków będącej rezultatem wyszukiwania wg zadanych kryteriów („Find pet"), jak też z poziomu listy zwierzaków należących do danego właściciela („Show pets"). Innym przykładem jest przypadek „Add new pet", który może być wywołany zarówno z poziomu przypadku „Find pet" (np. gdy okaże się, że szukany zwierzak nie został wcześniej dodany do katalogu), jak również bezpośrednio z głównego ekranu aplikacji (wywoływanie bezpośrednio z ekranu głównego jest możliwe dla przypadków, które nie wymagają określenia parametru wejściowego).

Dzięki temu w łatwy sposób można określać miejsca wywołania poszczególnych funkcji systemu w celu odzwierciedlenia procesów biznesowych czy też optymalnych schematów korzystania z aplikacji – takich, które minimalizują liczbę kliknięć dla uzyskania określonych efektów w określonym kontekście biznesowym.

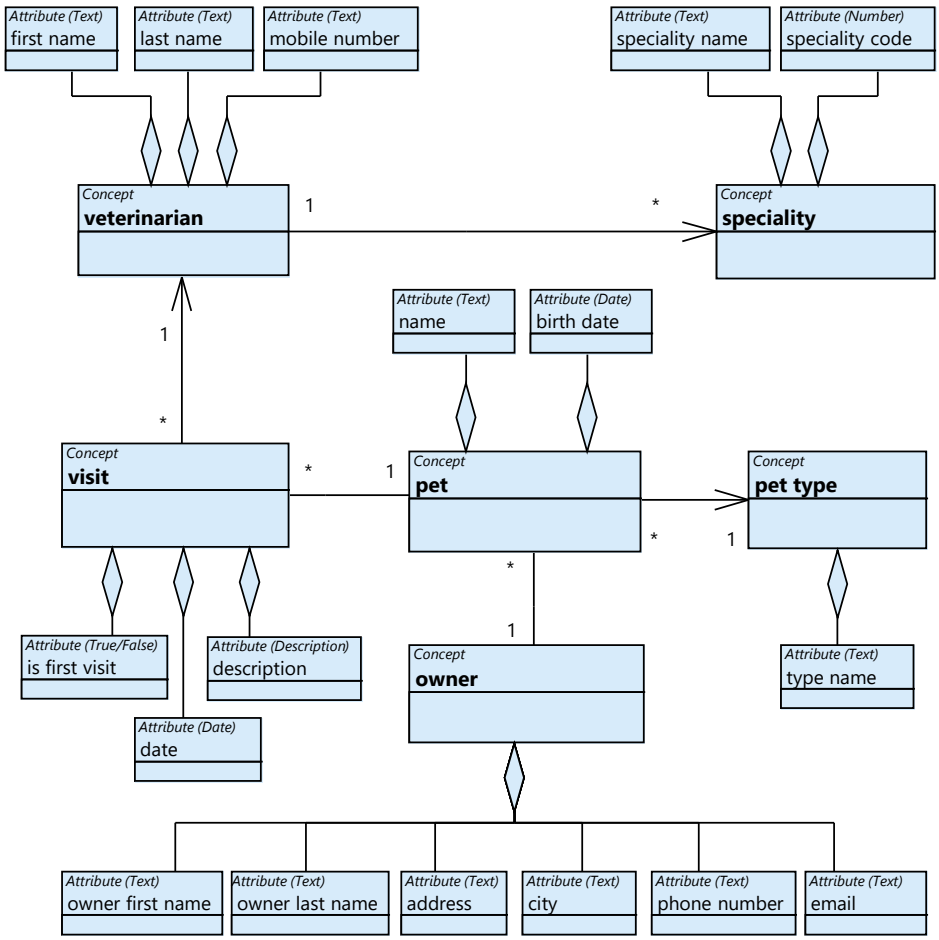
Funkcjonalność aplikacji nie może być definiowana w oderwaniu od pojęć reprezentujących fragment rzeczywistości biznesowej, która w języku RSL odzwierciedlana jest w postaci modelu dziedziny problemu. Model taki dla aplikacji Pet Clinic przedstawiony jest na diagramie na Rysunek 120. Jest to względnie prosty model składający się z sześciu pojęć typu „Concept", z których każde posiada szereg atrybutów istotnych z punktu widzenia tworzonej aplikacji. Warto zwrócić uwagę na krotności relacji między pojęciami dziedzinowymi, gdyż mają one wpływ na sposób generowania elementów interfejsu użytkownika, co zostanie pokazane na konkretnych przykładach. Przedstawiony model dziedziny problemu definiuje zakres danych przetwarzanych w ramach aplikacji Pet Clinic.



Rysunek 119 – Model przypadków użycia systemu Pet Clinic

O ile pojęcia z dziedziny problemu utworzone zostały ręcznie, o tyle wszystkie pojęcia z dziedziny aplikacji (pojęcia w pakietach „Data Views" oraz „UI Elements") zostały utworzone automatycznie w toku pisania scenariuszy przypadków użycia i automatycznie podlinkowane do zdań scenariuszy. Aby uczynić model kompletnym i gotowym do uruchomienia transformacji, automatycznie utworzone pojęcia zostały następnie ręcznie powiązane odpowiednimi relacjami (patrz: rozdział 3.2.2) istotnymi z punktu widzenia semantyki translacyjnej.

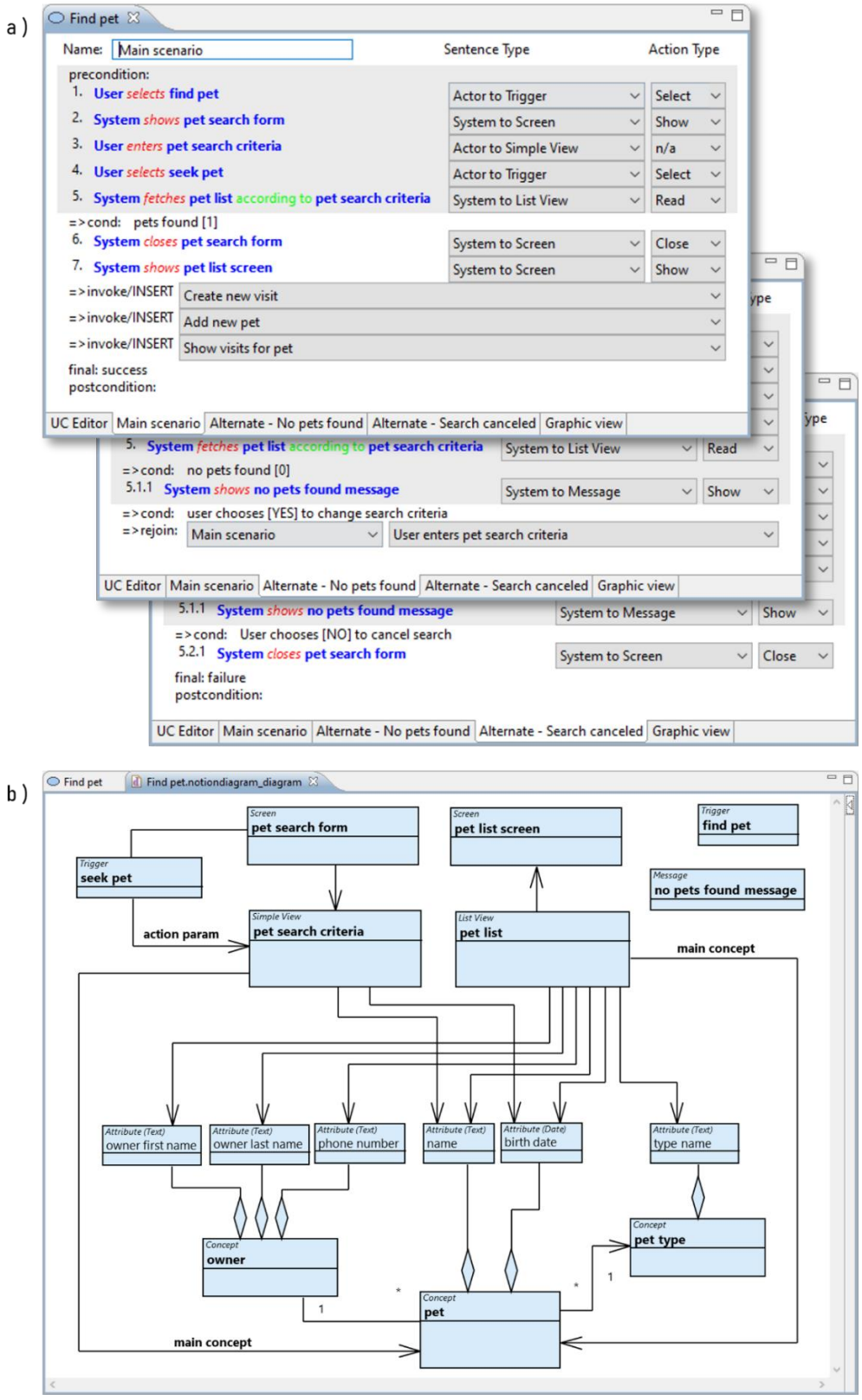
W dalszej części tego rozdziału zostaną szczegółowo omówione dwa przypadki użycia: „Find pet" oraz „Create new visit". Zaprezentowane zostaną ich scenariusze oraz model dziedzinowy. W Rozdziale 7.4 zostanie szczegółowo omówiony również kod wygenerowany dla przedstawionych fragmentów modelu źródłowego, co pozwoli na prześledzenie sposobu działania reguł translacji zaimplementowanych przez transformację „RSL to Java".



Rysunek 120 – Model dziedziny problemu systemu Pet Clinic

Przypadek użycia „Find pet” pozwala użytkownikowi wyszukać wszystkie zwierzaki na podstawie wprowadzonych kryteriów wyszukiwania a następnie wykonać dalsze akcje poprzez wywołanie kolejnych przypadków użycia. Szczegóły przypadku „Find pet” przedstawia Rysunek 121. Możliwe przebiegi interakcji aktora z systemem określają trzy scenariusze przedstawione w górnej części rysunku. Zdania od 1 do 5 są wspólne dla wszystkich scenariuszy. Interakcja rozpoczyna się standardowo od zdania Actor-to-Trigger – wybrania przez użytkownika wyzwalacza „find pet” na ekranie głównym (przypadek użycia połączony jest bezpośrednio z aktorem relacją „use”). W odpowiedzi system wyświetla ekran „pet search form” (zdanie System-to-Screen / Show), na którym użytkownik wprowadza kryteria wyszukiwania zwierzaka „pet search criteria” (zdanie Actor-to-SimpleView) i uruchamia wyszukiwanie przyciskiem „seek pet” (zdanie Actor-to-Trigger). Po tym następuje akcja systemu polegająca na pobraniu listy zwierzaków, które spełniają wprowadzone wcześniej kryteria wyszukiwania. Jest to wyrażone w scenariuszu złożonym zdaniem SVO (zdanie System-to-ListView / Read), w którym dopełnieniem bliższym jest pojęcie „pet list”

reprezentujące oczekiwane wyniki wyszukiwania, natomiast dopełnienie dalsze odnosi się do wprowadzonych kryteriów („pet search criteria”).



Rysunek 121 – Szczegóły przypadku użycia „Find pet”: scenariusze przypadku użycia (a) oraz model dziedzinowy (b)

Dalszy przebieg przypadku użycia uzależniony jest od rezultatów wyszukiwania, co wyrażone jest zdaniem warunkowym. Scenariusz główny zakłada, że operacja wyszukiwania zwróciła rezultaty („pets found [1]”), które mogą być zaprezentowane użytkownikowi. W takiej sytuacji, system najpierw zamyka wyświetlany aktualnie ekran „pet search form” (zdanie System-to-Screen / Close) a następnie wyświetla ekran „pet list screen” (zdanie System-to-Screen / Show) prezentujący rezultaty wyszukiwania w postaci widoku danych „pet list”. Tym samym przypadek użycia kończy się powodzeniem.

Ponieważ scenariusz nie zawiera zdania jawnie zamykającego ostatnio wyświetlony ekran, jest on prezentowany dopóki nie zostanie ręcznie zamknięty przez użytkownika. Zanim to nastąpi, chcemy aby użytkownik miał możliwość wywoływania trzech innych przypadków użycia: „Create new visit”, „Add new pet” oraz „Show visits for pet” za pomocą odpowiednich przycisków dostępnych z poziomu prezentowanego ekranu. Jest to możliwe dzięki umieszczeniu zdań „invoke” po zdaniu prezentującym ekran „pet list screen”.

Scenariusz alternatywny „Alternate – No pets found” jest realizowany, jeżeli operacja wyszukiwania zwierzaków nie zwróci żadnych rezultatów („no pets found [0]”). W takim przypadku, zamiast ekranu z rezultatami wyszukiwania, wyświetlane jest okno dialogowe „no pets found message” zawierające adekwatny komunikat („No pets have been found according to the given criteria. Do you want to change search criteria?”) oraz dwa przyciski: jeden, jeżeli użytkownik chce ponowić wyszukiwanie podając inne kryteria wyszukiwania, drugi – pozwalający zakończyć wyszukiwanie i cały przypadek użycia. Treść komunikatu zdefiniowana została w polu opisu pojęcia „no pets found message” za pomocą pary specjalnych znaczników „<ms>” oraz „</ms>”, stanowiących rozszerzenie notacji RSL na potrzeby transformacji „RSL to Java”. Przyciski dostępne w oknie komunikatu wynikają, natomiast, ze zdania warunkowego następującego po zdaniu System-to-Massage.

W przypadku wyboru przycisku „Yes”, chcemy aby system ponownie wyświetlił użytkownikowi ekran „pet search form” w celu zmiany kryteriów wyszukiwania i powtórzenia całej sekwencji zdarzeń począwszy od zdania nr 3. Jest to realizowane za pomocą zdania „rejoin”. Należy zauważyć, że nigdzie w scenariusz nie istnieje zdanie zamykające ekran „pet search form”, co oznacza, że jest on cały czas wyświetlany. Zdanie „rejoin” wskazuje zatem zdanie nr 3. Biorąc pod uwagę zastosowane środowisko translacyjne, wskazanie w zdaniu „rejoin” zdania nr 2, spowodowałoby wyświetlenie drugiego okna „pet search form” przysłaniającego już otwarte okno.

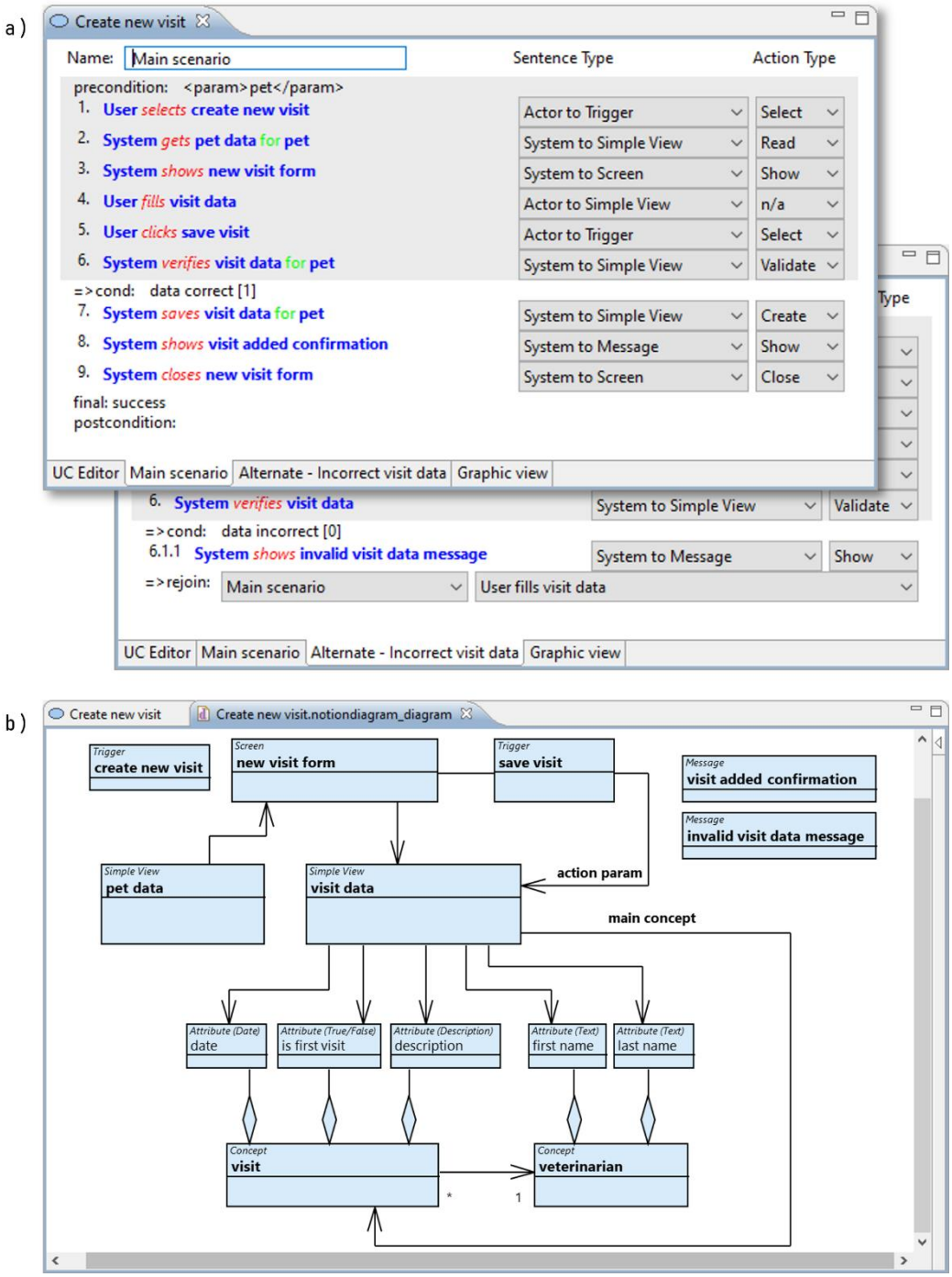
W przypadku wyboru przycisku „No” realizowany jest scenariusz „Alternate – Search canceled”, w ramach którego zamykany jest ekran „search pet form” (zdanie System-to-Screen / Close) a przypadek użycia kończony jest rezultatem „failure”.

Diagram na Rysunek 121b przedstawia pojęcia z dziedziny aplikacji utworzone w toku pisania scenariuszy przypadku użycia „Find pet” oraz powiązane z nimi elementy modelu dziedziny problemu. Jak wynika z reguł translacji, powiązania te są szczególnie istotne z punktu widzenia generowania kodu interfejsu użytkownika, gdyż określają zakres oraz formę prezentacji danych. Prosty widok danych „pet search form” reprezentujący kryteria wyszukiwania, wskazuje jedynie dwa atrybuty należące do pojęcia „pet”; jest również powiązany z ekranem „pet search form” relacją typu „Input”. Spodziewanym rezultatem transformacji będzie w tym przypadku prosty ekran zawierający dwie kontrolki: pole tekstowe, odpowiadające atrybutowi typu Text, oraz pole wyboru daty, odpowiadające atrybutowi typu Date. Ekran ten będzie zawierał również przycisk „seek pet” uruchamiający akcję wyszukiwania. Jej parametrem będą dane wprowadzone przez użytkownika, na co wskazuje relacja „action param”.

Bardziej rozbudowanym widokiem danych jest „pet list”. Wskazuje on atrybuty należące do trzech pojęć typu „Concept”. Istotna w tym przypadku jest relacja „main concept” wskazująca „pet” jako pojęcie główne. Pozwala to transformacji poprawnie zinterpretować krotności między pojęciami biznesowymi. Tutaj, dane każdego zwierzaka będą prezentowane jako pojedyncze wiersze listy a atrybuty należące do pojęć „owner” oraz „pet type” będą prezentowane w danym wierszu jako pojedyncze wartości w odpowiednich kolumnach. Wynika z krotności „1” po stronie tych pojęć. Lista reprezentowana przez widok „pet list” będzie wyświetlany użytkownikowi tylko do odczytu, stąd relacja typu „Present” łącząca je z ekranem „pet list screen”.

Drugim omawianym przypadkiem użycia jest „Create new visit”. Szczegóły tego przypadku przedstawia Rysunek 122. Jego istotną cechą jest parametr wejściowy wskazany w zdaniu „precondition”. Umówienie wizyty jest możliwe tylko dla konkretnego zwierzaka, który musi być wskazany w ramach przypadku wywołującego. W naszym przykładzie przypadkiem wywołującym może być „Show pets” oraz omawiany powyżej „Find pet”. Rezultatem pomyślnego wykonania obu tych przypadków jest lista obiektów typu „pet” prezentowana użytkownikowi. Po wybraniu konkretnego obiektu z listy, użytkownik może utworzyć wizytę poprzez wybór przycisku „create new visit”. Obiekt „pet” przekazany jako parametr wywołania

pojawia się w scenariuszu w zdaniach „System-to-SimpleView” (zdania nr 2, 6 oraz 7) jako dopełnienie dalsze stanowiące kontekst akcji systemowych wykonywanych na widokach danych „pet data” oraz „visit data”. Zgodnie z regułami translacji M1 oraz P9 przekłada się to na odpowiednie parametry wywołania metod warstwy modelu, co zostanie pokazane w Rozdziale 7.4.



Rysunek 122 – Szczegóły przypadku użycia „Create new visit”: scenariusze przypadku użycia (a) oraz model dziedzinowy (b)

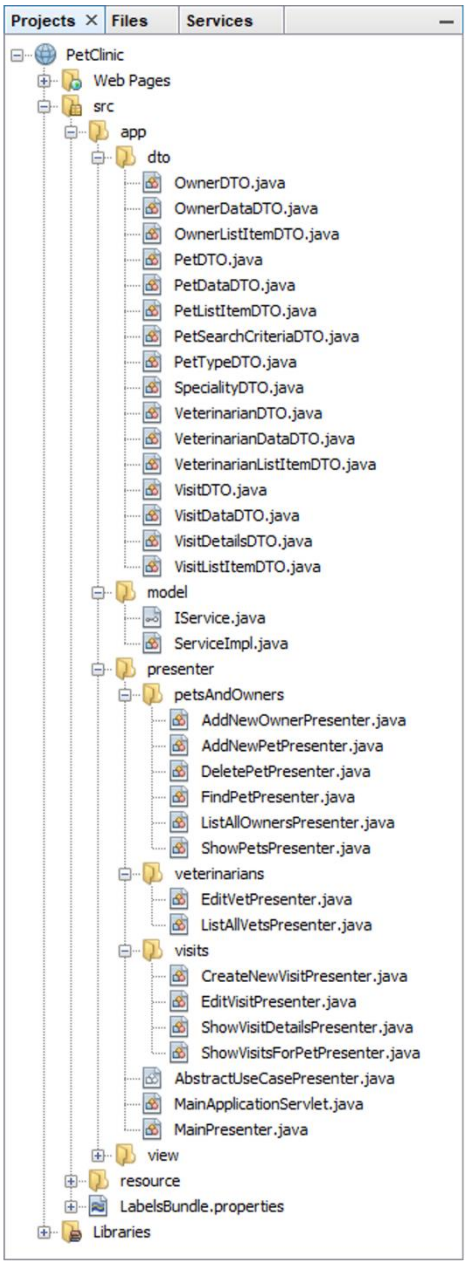
Drugą wartą podkreślenia cechą omawianego przypadku jest powiązanie ekranu „new visit form” z dwoma widokami danych: „pet data” oraz „visit data” (patrz: diagram pojęć na Rysunek 122b). Widok „pet data” jest powiązany z ekranem relacją „Present”. Jego rolą jest wyświetlenie użytkownikowi podstawowych danych zwierzaka („pet”), dla którego ma być utworzona wizyta a także danych jego właściciela („owner”), bez możliwości ich modyfikowania. Dla czytelności wskazywane atrybuty zostały pominięte na diagramie. Ten sam ekran wyświetla jednocześnie pola do wprowadzenia danych wizyty. Stąd, widok danych „visit data” powiązany został z ekranem relacją „Input”. Atrybuty wskazywane przez ten widok należą do pojęcia „visit” – pojęcia głównego – oraz „veterinarian” – weterynarza przeprowadzającego wizytę. Relacja łącząca te dwa pojęcia oznacza, że atrybuty pojęcia „veterinarian” wskazywane przez widok danych „visit data” powinny przełożyć się, zgodnie z regułą V4, na kontrolki umożliwiające użytkownikowi wybór weterynarza z listy wszystkich weterynarzy.

7.2 Ogólna struktura wygenerowanego kodu

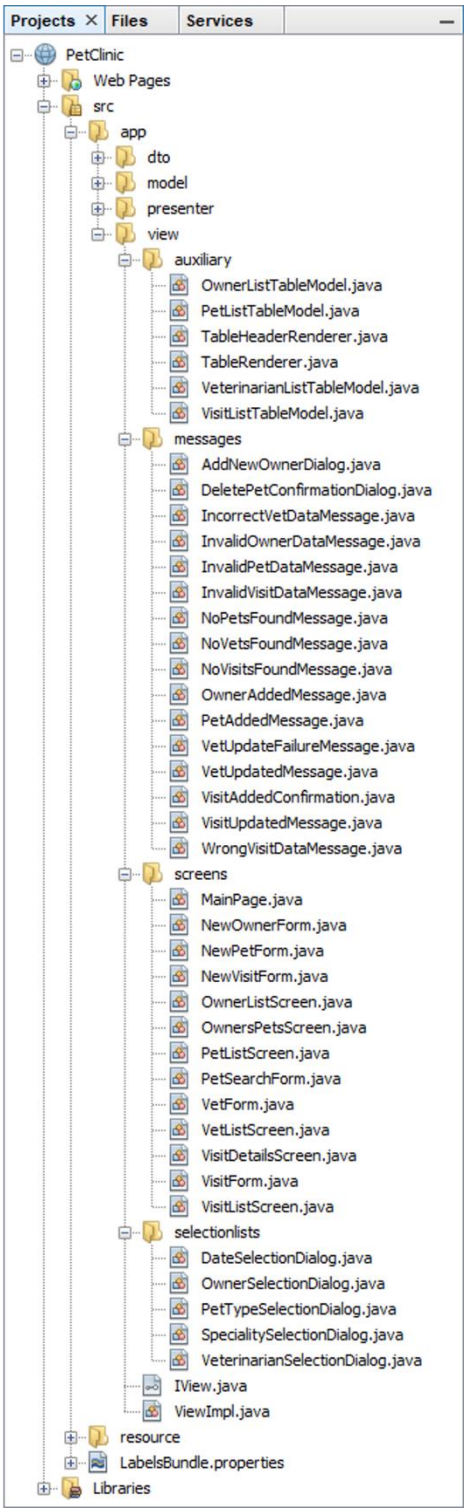
Rysunek 123 prezentuje strukturę projektu Pet Clinic w środowisku deweloperskim. Najważniejszym pakietem jest pakiet `/src/app` zawierający kod aplikacji, będący rezultatem wykonania transformacji RSL to Java dla modelu źródłowego. W skład tego pakietu wchodzą pakiety klas oraz klasy wygenerowane statycznie, zgodnie z wymaganiami uogólnionego środowiska translacyjnego (patrz: rozdział 5.1, Rysunek 53) oraz konkretnego środowiska docelowego Echo3, a także pakiety i klasy wygenerowane na podstawie modelu źródłowego, zgodnie z regułami translacji.

W pakiecie `/src/app/presenter` warto zwrócić uwagę na podpakiety grupujące klasy prezenterów. Odzwierciedlają one strukturę pakietów przypadków użycia w modelu źródłowym. W pakiecie `/src/app/view`, natomiast, oprócz podpakietów `../screens` i `../messages` wynikających z przyjętego środowiska translacyjnego, znajdują się również pakiety `../auxiliary` oraz `../selectionlists`. Oba wynikają z faktu zastosowania środowiska Echo3 i sposobu implementacji określonych elementów interfejsu użytkownika. Pierwszy z nich zawiera klasy reprezentujące dane tabelaryczne dla pojęć typu „List View”. Drugi – klasy implementujące modalne okna dialogowe z listami wyboru wyświetlane w przypadkach opisanych w regule V4, czego przykładem jest lista wyboru weterynarza w przypadku użycia „Create new visit”.

a)



b)



Rysunek 123 – Struktura kodu systemu Pet Clinic: (a) klasy DTO, modelu oraz prezntera, (b) klasy widoku

Oprócz kodu źródłowego aplikacji, w katalogu `/src` znajduje się plik konfiguracyjny `LabelsBundle.properties` oraz podkatalog `../resource`. Plik zawiera etykiety ekranowe elementów interfejsu użytkownika w postaci klucz-wartość. Katalog, natomiast,

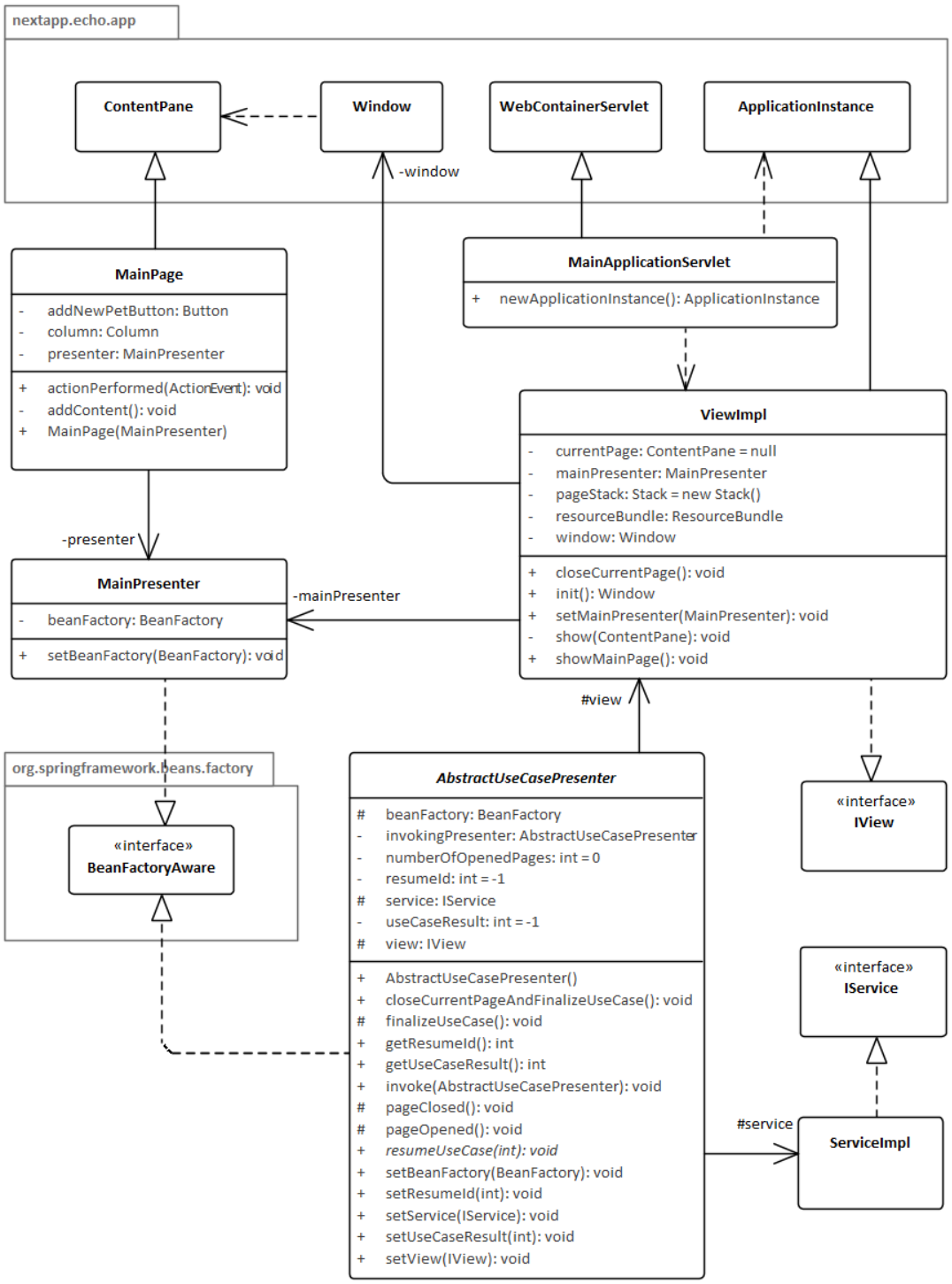
zawiera definicję stylów dla poszczególnych elementów interfejsu użytkownika (plik konfiguracyjny oraz pliki graficzne). Kod wygenerowany dla warstwy widoku odnosi się do obu plików konfiguracyjnych, przy czym, plik z etykietami jest w całości generowany przez transformację a zawartość podkatalogu `../resource` jest dodawana ręcznie do projektu.

Pozostałe katalogi widoczne w drzewie projektu. tj. `/WebPages` oraz `/Libraries` zawierają pliki konfiguracyjne aplikacji internetowej (m.in. deskryptor wdrożenia, plik manifestu, plik konfiguracyjny środowiska Spring) oraz niezbędne biblioteki środowiska docelowego Echo3 oraz Spring.

7.3 Kod generowany statycznie

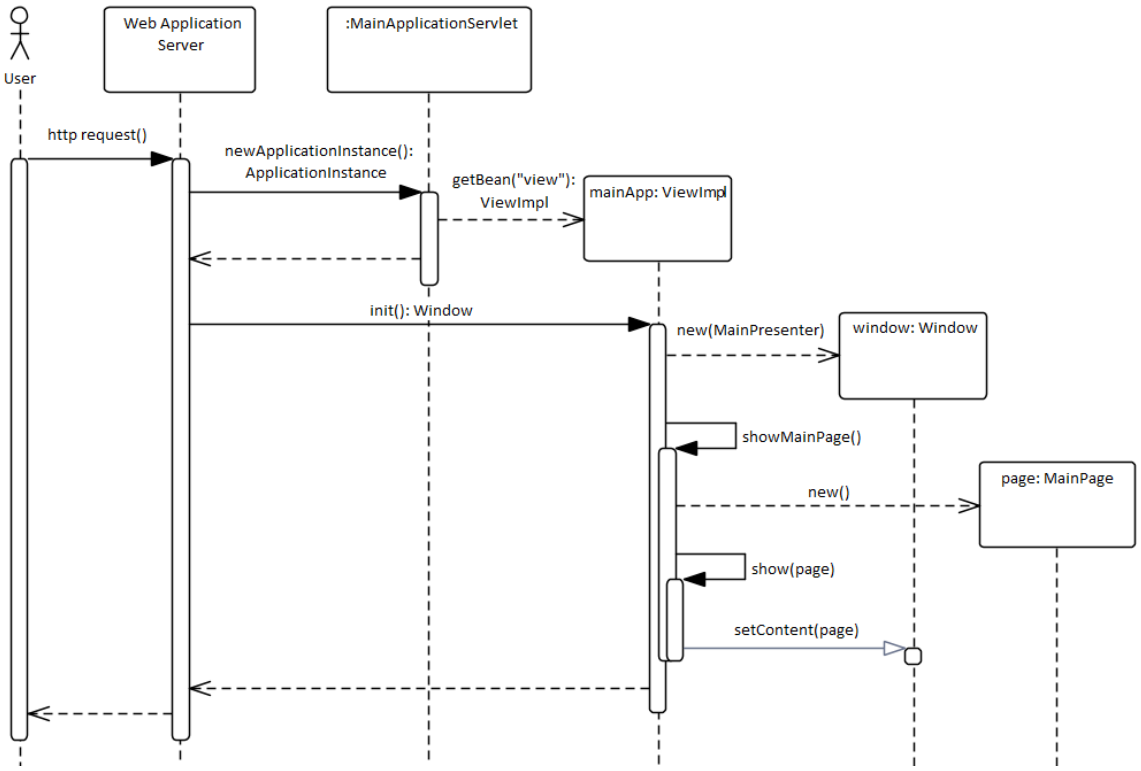
Zgodnie z założeniami projektowymi dla transformacji RSL to Java, wygenerowany kod aplikacji powinien być na tyle kompletny, aby można było go bezpośrednio skompilować i uruchomić. Kompletność oznacza istnienie kodu wygenerowanego na podstawie źródłowej specyfikacji wymagań w języku RSL, jak również kodu wygenerowanego statycznie, stanowiącego „infrastrukturę" aplikacji niezbędną do jej uruchomienia i poprawnego działania. „Infrastruktura" ta musi spełniać założenia środowiska translacyjnego (patrz: rozdział 5.1). Musi być również zgodna z konkretnym środowiskiem docelowym. W przypadku transformacji RSL to Java, środowiskiem docelowym jest Echo3 z elementami Spring (patrz: rozdział 6.1), co implikuje ramowy sposób działania aplikacji.

Diagram na Rysunek 124 przedstawia klasy oraz interfejsy (wraz ze wszystkimi niezbędnymi zależnościami) stanowiące „infrastrukturę" aplikacji PetClinic. Widoczne elementy są efektem wykonania pierwszego etapu transformacji RSL to Java, tj. procedury InitializeTargetModel() opisanej w Rozdziale 6.3. Są one niezależne od modelu źródłowego i dopiero w toku wykonywania dalszych części transformacji, są uzupełniane o atrybuty, metody i zależności wygenerowane na podstawie reguł translacji, co zostanie zaprezentowane w Rozdziale 7.4. Istotne dla poprawnej kompilacji i działania aplikacji są zależności pomiędzy wygenerowanymi klasami a elementami środowiska docelowego widocznymi w pakietach `nextapp.echo.app` oraz `org.springframework.beans.factory`. Elementy te nie są generowane ale muszą być widoczne dla wygenerowanego kodu w postaci zewnętrznych bibliotek, aby odwołania do nich były poprawne.



Rysunek 124 – Podstawowe klasy i interfejsy generowane statycznie oraz ich zależności

Zanim użytkownik wywoła pierwszy przypadek użycia aplikacji, musi zostać wyświetlona strona główna. Mechanizm uruchamiania aplikacji prowadzący do wyświetlenia strony głównej przedstawia diagram sekwencji na Rysunek 125.



Rysunek 125 – Diagram sekwencji – uruchamianie aplikacji

Uruchomienie aplikacji następuje poprzez wpisanie przez użytkownika (`User`) adresu URL aplikacji w przeglądarce. Po otrzymaniu żądania użytkownika, serwer aplikacji (`Web Application Server`) pobiera z głównego serwletu (`MainApplicationServlet`) instancję aplikacji. Rolę instancji aplikacji pełni obiekt klasy `ViewImpl`, który jest pobierany metodą `getBean()` z kontenera IoC Spring. Rolą instancji aplikacji jest utworzenie i zwrócenie – na żądanie serwera – okna aplikacji (`nextapp.echo.app.Window`). W tym celu klasa `ViewImpl` musi implementować abstrakcyjną metodę `init()` odziedziczoną z nadklasy `nextapp.echo.app.ApplicationInstance`. Cały statycznie wygenerowany kod klasy `ViewImpl`, włącznie z metodą `init()`, zaprezentowany jest na Listing 1. Klasa `nextapp.echo.app.Window` stanowi główny kontener aplikacji typu Single Page Application, w którym umieszczane są konkretne ekrany aplikacji prezentowane użytkownikowi w oknie przeglądarki (muszą one dziedziczyć z klasy `nextapp.echo.app.ContentPane`), zgodnie z przebiegiem przypadków użycia. Pierwszym ekranem dodawanym do kontenera podczas uruchamiania aplikacji jest oczywiście ekran główny aplikacji `MainPage`. Dzieje się to w konsekwencji wywołania metody `showMainPage()`. Z poziomu tego ekranu użytkownik ma możliwość wywoływania przypadków użycia, zgodnie z modelem RSL.

Listing 1 - Kod klasy ViewImpl generowany statycznie

```
1   public class ViewImpl extends ApplicationInstance implements IView {
2
3       private ContentPane currentPage = null;
4       private MainPresenter mainPresenter;
5       private Stack pageStack = new Stack();
6       private ResourceBundle resourceBundle;
7       private Window window;
8
9       public void finalize() throws Throwable {
10          super.finalize();
11      }
12
13      public void closeCurrentPage() {
14          if (!pageStack.isEmpty()) {
15              currentPage = (ContentPane) pageStack.pop();
16              show(currentPage);
17          }
18      }
19
20      public Window init() {
21
22          // Load stylesheet
23          StyleSheet styleSheet;
24          try {
25              styleSheet = StyleSheetLoader.load("resource/style/Default.stylesheet.xml",
26                  Thread.currentThread().getContextClassLoader());
27          } catch (SerialException ex) {
28              throw new RuntimeException(ex);
29          }
30          this.setStyleSheet(styleSheet);
31
32          // Get resource bundle
33          resourceBundle = ResourceBundle.getBundle("LabelsBundle",
34              ApplicationInstance.getActive().getLocale());
35
36          // Show main application page
37          window = new Window();
38          showMainPage();
39
40          return window;
41      }
42
43      public void setMainPresenter(MainPresenter mainPresenter) {
44          this.mainPresenter = mainPresenter;
45      }
46
47      public void showMainPage() {
48          if (currentPage != null)
49              pageStack.push(currentPage);
50          MainPage page = new MainPage(mainPresenter);
51          show(page);
52      }
53
54      private void show(ContentPane page) {
55          window.setContent(page);
56          currentPage = page;
57      }
58
59      // RSL-dependent code goes here...
60
61  }
```

Jak zaznaczono w opisie środowiska translacyjnego (rozdział 5.1), klasy prezenterów implementujące logikę przypadków użycia, mają wspólną nadklasę `AbstractUseCasePresenter` zapewniającą ogólny mechanizm przepływu sterowania. Listing 2 przedstawia kod tej klasy wygenerowany przez transformację. Będzie on przydatny

do szczegółowego zrozumienia logiki działania przypadków użycia aplikacji Pet Clinic przedstawionych w Rozdziale 7.4.

Listing 2 - Kod klasy AbstractUseCasePresenter

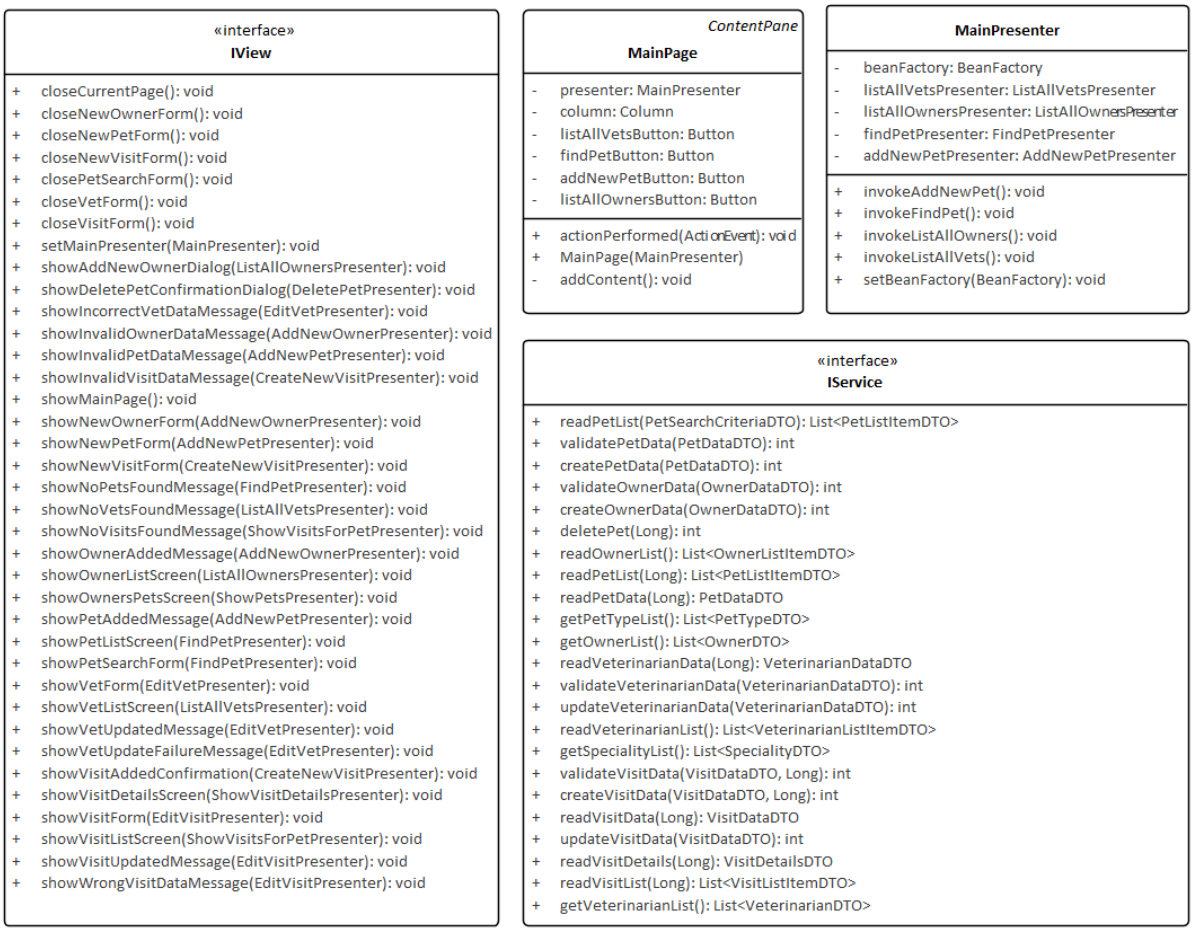
```
public abstract class AbstractUseCasePresenter implements BeanFactoryAware {

    private AbstractUseCasePresenter invokingPresenter = null;
    private int numberOfOpenedPages = 0;
    private int resumeId = -1;
    private int useCaseResult = -1;
    protected BeanFactory beanFactory;
    protected IView view;
    protected IService service;

    public AbstractUseCasePresenter() {
        service = (IService) beanFactory.getBean("Service");
    }

    public void closeCurrentPageAndFinalizeUseCase() {
        if (numberOfOpenedPages > 0) {
            view.closeCurrentPage();
            pageClosed();
        }
        finalizeUseCase();
    }

    public void finalize() throws Throwable {

    }

    protected void finalizeUseCase() {
        if (numberOfOpenedPages == 0)
            if (invokingPresenter != null
                invokingPresenter.resumeUseCase(useCaseResult);
    }

    public int getResumeId() {
        return this.resumeId;
    }

    public int getUseCaseResult() {
        return this.useCaseResult;
    }

    public void invoke(AbstractUseCasePresenter invokingPresenter) {
        this.invokingPresenter = invokingPresenter;
    }

    protected void pageClosed() {
        if (numberOfOpenedPages > 0)
            numberOfOpenedPages--;
    }

    protected void pageOpened() {
        numberOfOpenedPages++;
    }

    public abstract void resumeUseCase(int result);

    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }

    public void setResumeId(int resumeId) {
        this.resumeId = resumeId;
    }

    public void setService(IService service) {
        this.service = service;
    }

    protected void setUseCaseResult(int useCaseResult) {
        this.useCaseResult = useCaseResult;
    }

    public void setView(IView view) {
```

```
73          this.view = view;
74      }
75
76  }
```

7.4 Kod generowany w oparciu o reguły translacji

W strukturze kodu na Rysunek 123 przedstawione zostały wszystkie interfejsy i klasy będące finalnym wynikiem działania transformacji RSL to Java dla modelu źródłowego aplikacji Pet Clinic: podstawowe interfejsy i klasy stanowiące „infrastrukturę" aplikacji, klasy DTO, klasy prezenterów oraz wszystkie klasy w warstwie widoku (ekrany, komunikaty, listy i klasy pomocnicze). Jak wspomniano w poprzednim rozdziale, kod podstawowych interfejsów i klas jest częściowo generowany statycznie a następnie, w toku wykonania transformacji, uzupełniany w oparciu o reguły translacji. Diagram na Rysunek 126 prezentuje podstawowe interfejsy i klasy z kompletem atrybutów i operacji, będących finalnym efektem transformacji RSL to Java dla modelu źródłowego Pet Clinic. Najistotniejsze są tutaj interfejsy `IView` oraz `IService`. W przypadku pierwszego z nich, widzimy zestaw kilkudziesięciu operacji wygenerowanych zgodnie z regułami translacji V1 oraz V2, w oparciu o zdania typu System-to-Screen oraz System-to-Message obecne w scenariuszach przypadków użycia aplikacji Pet Clinic. Kod implementujący te operacje został wygenerowany w klasie `ViewImpl`. W dalszej części przyjrzymy się bliżej wybranym operacjom klasy `ViewImpl`, wywoływanym w ramach przypadku użycia „Find pet" oraz „Create new visit". Operacje w interfejsie `IService` są rezultatem wykonania reguł M1 oraz M2, na podstawie zdań System-to-SimpleView, System-to-ListView oraz System-to-Concept. Implementacje tych operacji w klasie `ServiceImpl` pozostają zasadniczo puste. Jedynie w przypadku operacji, których sygnatury określają typ zwracany, generowana jest instrukcja `return` zwracająca domyślną wartość odpowiedniego typu aby umożliwić poprawną kompilację kodu.

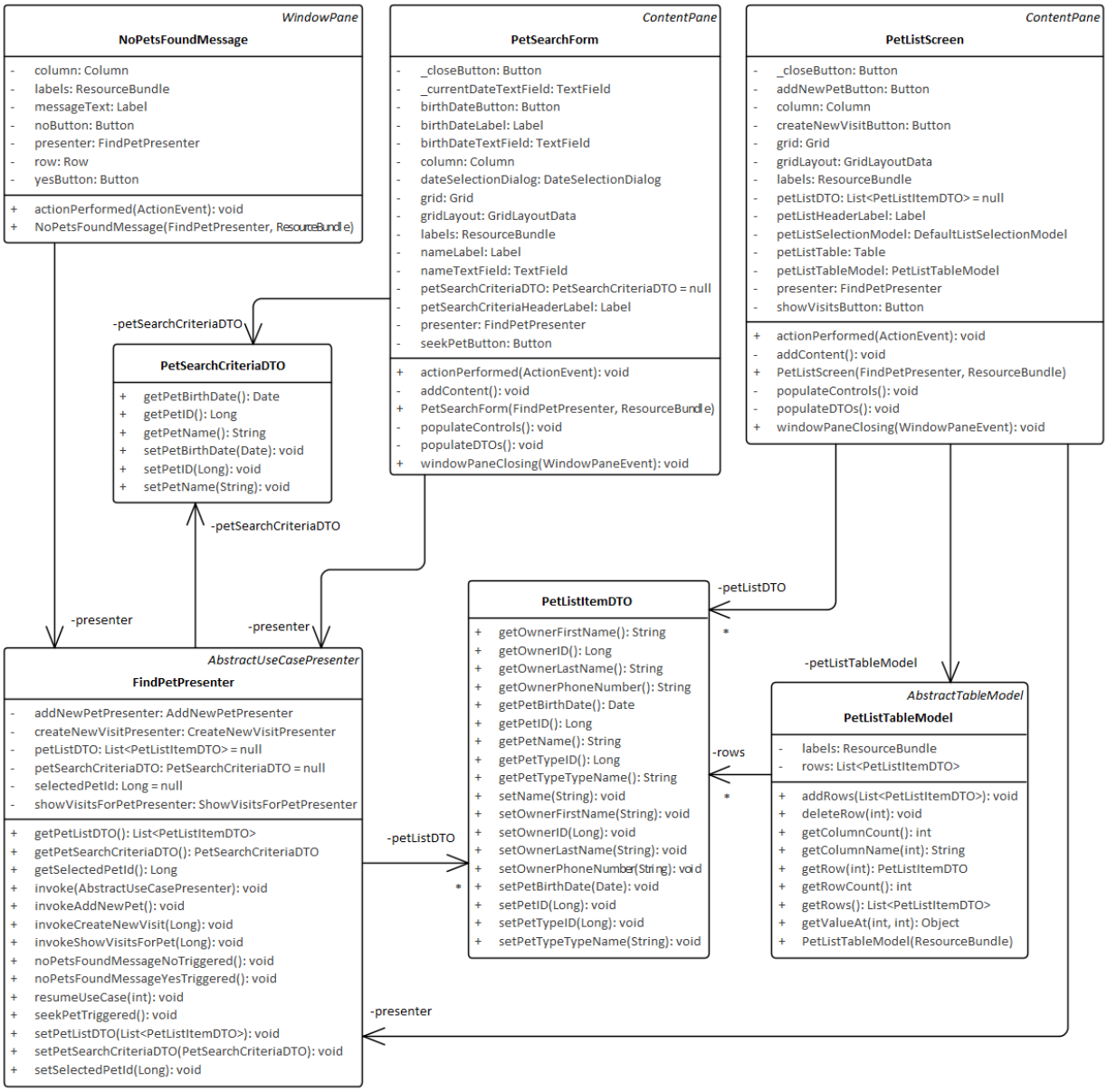
Na diagramie widoczne są również klasy `MainPage` oraz `MainPresenter`, których kod został rozszerzony o elementy związane z wywołaniem czterech przypadków użycia posiadających bezpośrednią relację „use" z aktorem: „Add new pet", „Find pet", „List all owners" oraz „List all vets". Elementy te generowane są zgodnie z regułami G5 oraz G6.



Rysunek 126 – Podstawowe klasy i interfejsy aplikacji Pet Clinic uzupełnione o atrybuty i metody wygenerowane w oparciu o reguły translacji

Rysunek 127 przedstawia klasy realizujące przypadek użycia „Find pet". Najważniejszą klasą realizującą logikę tego przypadku użycia jest `FindPetPresenter`. Podstawowy kod klasy (wraz z dwiema nadpisywanymi metodami z nadklasy `AbstractUseCasePresenter`, tj. `invoke()` oraz `resumeUseCase()`) generowany jest zgodnie z regułą G2. Wszystkie pozostałe atrybuty i metody klasy widoczne na diagramie, są efektem translacji poszczególnych zdań scenariuszy (patrz: Rysunek 121a), zgodnie z odpowiednimi regułami translacji do warstwy prezentera: P2, P4, P8, P11. Kod metod, natomiast, jest generowany według reguł P2, P5, P6, P9, P14, P15, P16. Wynikiem wykonania wszystkich wymienionych reguł jest kompletny kod klasy prezentera dla przypadku użycia „Find pet" widoczny na Listing 3 (pominięto tu jedynie standardowy kod metod dostępowych do atrybutów). Warto zauważyć, że ciało metody `noPetsFoundMessageYesTriggered()` jest puste. Zgodnie ze scenariuszem, powinna się tam znaleźć kod generowany dla zdania „rejoin". Pomiędzy zdaniem wskazywanym przez zdanie „rejoin" („User enters pet searchcriteria") a najbliższym

zdaniem „Actor-to-Trigger” („User selects seek pet”) nie występują jednak żadne zdania, stąd, zgodnie z regułą P15, nie jest generowany żaden kod. Puste jest również ciało metody `resumeUseCase()`, co wynika z faktu, że w scenariuszach przypadku użycia „Find pet” nie występują zdania „invoke” wykonywane przez system. Zgodne z regułą P15, w takiej sytuacji nie jest generowany żaden kod dla zdania „rejoin”.



Rysunek 127 – Diagram klas dla przypadku użycia Find pet

Listing 3 – Kod klasy FindPetPresenter

```
public class FindPetPresenter extends AbstractUseCasePresenter {

    private Long selectedPetId = null;
    private List<PetListItemDTO> petListDTO = null;
    private PetSearchCriteriaDTO petSearchCriteriaDTO = null;
    private CreateNewVisitPresenter createNewVisitPresenter;
    private AddNewPetPresenter addNewPetPresenter;
    private ShowVisitsForPetPresenter showVisitsForPetPresenter;

    public void invoke(AbstractUseCasePresenter invokingPresenter){
        super.invoke(invokingPresenter);
        view.showPetSearchForm(this);
        pageOpened();
    }

    public void invokeAddNewPet(){
        addNewPetPresenter = (AddNewPetPresenter) beanFactory.getBean("addNewPetPresenter");
        addNewPetPresenter.invoke(this);
    }

    public void invokeCreateNewVisit(Long selectedPetId){
        createNewVisitPresenter = (CreateNewVisitPresenter) beanFactory.getBean("createNewVisitPresenter");
        this.selectedPetId = selectedPetId;
        createNewVisitPresenter.invoke(this, selectedPetId);
    }

    public void invokeShowVisitsForPet(Long selectedPetId){
        showVisitsForPetPresenter =
            (ShowVisitsForPetPresenter) beanFactory.getBean("showVisitsForPetPresenter");
        this.selectedPetId = selectedPetId;
        showVisitsForPetPresenter.invoke(this, selectedPetId);
    }

    public void noPetsFoundMessageNoTriggered(){
        view.closePetSearchForm();
        pageClosed();
        setUseCaseResult(0);
        finalizeUseCase();
    }

    public void noPetsFoundMessageYesTriggered(){

    }

    public void resumeUseCase(int invocationResult){

    }

    public void seekPetTriggered(){
        petListDTO = service.readPetList(petSearchCriteriaDTO);
        if (!petListDTO.isEmpty()) { /* pets found [1] */
            view.closePetSearchForm();
            pageClosed();
            view.showPetListScreen(this);
            pageOpened();
            setUseCaseResult(1);
            this.finalizeUseCase();
        }
        else if (petListDTO.isEmpty()) { /* no pets found [0] */
            view.showNoPetsFoundMessage(this);
        }
    }

    /* Getters and setters omitted for brevity */

}
```

Rysunek 127 przedstawia również klasy warstwy widoku: `PetSearchForm`, `PetListScreen` oraz `NoPetsFoundMessage`. Zostały one wygenerowane na podstawie pojęć typu „screen” oraz „message”, zgodnie z regułami G3 oraz G4, a następnie uzupełnione o kod będący rezultatem translacji modelu dziedzinowego przedstawionego na Rysunek 121b.

W przypadku trzech omawianych klas widoku, zastosowanie miały wszystkie reguły translacji do warstwy widoku.

W oparciu o reguły V1 oraz V2 nie jest wprawdzie generowany kod samych klas reprezentujących ekrany i komunikaty, lecz metody interfejsu `IView` oraz ich implementacje w klasie `ViewImpl`, wykorzystywane przez klasę prezentera do wyświetlania i ukrywania ekranów i komunikatu, zgodnie z przebiegiem scenariuszy (zdania System-to-Screen oraz System-to-Message). Fragment kodu klasy `ViewImpl` zawierający implementację tych metod przedstawia Listing 4. Zgodnie z założeniami środowiska translacyjnego, metody te wykorzystują stos (`pageStack`) w celu przechowywania aktualnego stanu widoku aplikacji, tj. wszystkich otwartych ekranów i ich kolejności.

Listing 4 – Kod klasy ViewImpl – metody wygenerowane dla przypadku użycia Find pet

```
1  public class ViewImpl extends ApplicationInstance implements IView {
2
3      /* ... */
4
5      public void showPetSearchForm(FindPetPresenter presenter){
6          if (currentPage != null)
7              pageStack.push(currentPage);
8          PetSearchForm page = new PetSearchForm(presenter, resourceBundle);
9          show(page);
10     }
11
12     public void closePetSearchForm(){
13         if (currentPage instanceof PetSearchForm)
14             closeCurrentPage();
15         else
16             for (int i = pageStack.size()-1; i >= 0; i--) {
17                 if(pageStack.get(i) instanceof PetSearchForm) {
18                     pageStack.remove(i);
19                     break;
20                 }
21             }
22     }
23
24     public void showPetListScreen(FindPetPresenter presenter){
25         if (currentPage != null)
26             pageStack.push(currentPage);
27         PetListScreen page = new PetListScreen(presenter, resourceBundle);
28         show(page);
29     }
30
31     /* ... */
32
33 }
```

Pozostałe reguły (V3 – V11) miały zastosowanie przy generowaniu kodu omawianych klas ekranowych. Najważniejsze fragmenty kodu klasy `PetSearchForm` przedstawia Listing 5 a klasy `PetListScreen` – Listing 6. W wygenerowanym kodzie zauważyć można drobną różnicę w stosunku do abstrakcyjnego środowiska translacyjnego. Otóż, kod tworzący i rozmieszczający kontrolki ekranowe nie znajduje się w publicznej metodzie `init()`, lecz w prywatnej metodzie `addContent()` wywoływanej w konstruktorze klasy. Przed jej wywołaniem wykonywany jest kod typowy dla każdej klasy ekranowej w przyjętym

środowisku konkretnym. Po pierwsze, inicjalizowane są atrybuty `presenter` oraz `labels` obiektami przekazanymi przy tworzeniu klasy ekranowej (patrz: wiersz 8 i 27 w kodzie klasy `ViewImpl` na Listing 4). Atrybut `presenter` inicjowany jest oczywiście obiektem klasy `FindPetPresenter`, natomiast `labels` – obiektem typu `ResourceBundle` przechowującym etykiety ekranowe dla całej aplikacji. Po drugie, definiowany jest sposób ułożenia kontrolek na ekranie. W obecnej wersji transformacji wykorzystany został jeden z podstawowych układów w środowisku Echo3 – `nexapp.echo.app.Column`. Po trzecie, dodawany jest przycisk (`_closeButton`) pozwalający użytkownikowi zamknąć w dowolnym momencie aktualnie wyświetlany ekran i powrócić do ekranu wyświetlonego wcześniej. Jak widać w metodzie obsługi zdarzeń `actionPerformed()`, użycie tego przycisku skutkuje wywołaniem metody `closeCurrentPageAndFinalizeUseCase()`, zdefiniowanej w klasie `AbstractUseCasePresenter` (patrz: Listing 2).

Metoda `addContent()` jest najważniejszą metodą klasy ekranowej, gdyż to ona definiuje jej zawartość, będącą efektem translacji widoków danych powiązanych z ekranami. To w tej metodzie generowana jest główna część kodu wynikająca z reguł V3, V4, V5, V8 oraz V9.

Zgodnie z modelem dziedzinowym przypadku użycia „Find pet", klasa `PetSearchForm` musi odzwierciedlać prosty widok danych „pet search criteria" wskazujący dwa atrybuty pojęcia „pet": „name" oraz „birth date". Przekładają się one na dwie edytowalne (ze względu na typ relacji między widokiem danych a pojęciem ekranowym) kontrolki: pole tekstowe (`nameTextField`) oraz pole daty (`birthDateTextField`) wraz z przyciskiem wyboru daty z kalendarza (`birthDateButton`). Obie kontrolki poprzedzone są etykietami ekranowymi `nameLabel` oraz `birthDateLabel`, których tekst pobierany jest z obiektu `labels`. Kod definiujący taki zestaw elementów ekranowych widoczny jest w wierszach 56-96 na Listing 5. Oprócz tego, metoda `addContent()` w klasie `PetSearchForm` definiuje jeszcze przycisk wywołujący akcję wyszukiwania (`seekPetButton`) odpowiadający pojęciu „seek pet" typu „Trigger" (wiersze 98-103). Kod obsługi zdarzenia dla tego przycisku wygenerowany został w metodzie `actionPerformed()` (wiersze 119-124). Warto tutaj zaznaczyć, że – zgodnie z semantyką translacyjną – efektem działania kodu obsługi zdarzenia jest w tym przypadku przekazanie kontroli do obiektu prezentera poprzez wywołanie metody `seekPetTriggered()`. Przedtem, jednak, wywoływana jest metoda `populateDTOs()` (wiersze 127-142) zasilająca obiekt `petSearchCriteriaDTO`

wartościami wprowadzonymi przez użytkownika w polach `nameTextField` oraz `birthDateTextField` oraz przekazanie tego obiektu do prezentera.

Listing 5 – Kod klasy PetSearchForm

```
public class PetSearchForm extends ContentPane implements ActionListener, WindowPaneListener {

    /* ... */

    public PetSearchForm(FindPetPresenter presenter, ResourceBundle labels){

        // Set presenter
        this.presenter = presenter;

        // Set labels resource bundle
        this.labels = labels;

        // Add column layout
        column = new Column();
        column.setInsets(new Insets(30, 15));
        column.setCellSpacing(new Extent(10));
        column.setStyleName("Column.ContentPane");
        add(column);

        // Add Close button
        _closeButton = new Button(labels.getString("Button.Back"));
        _closeButton.setStyleName("Button.Back");
        _closeButton.setActionCommand("_closeButton");
        _closeButton.addActionListener(this);
        column.add(_closeButton);

        // Add page content
        addContent();

        petSearchCriteriaDTO = new PetSearchCriteriaDTO();
    }

    private void addContent(){

        grid = new Grid(2);
        grid.setInsets(new Insets(10, 7));
        grid.setWidth(new Extent(100, Extent.PERCENT));
        grid.setStyleName("Default");
        column.add(grid);

        // Add content for 'pet search criteria' data view

        // Data view header
        petSearchCriteriaHeaderLabel =
            new Label(labels.getString("PetSearchForm.PetSearchCriteria.Header"));
        gridLayout = new GridLayoutData();
        gridLayout.setColumnSpan(2);
        gridLayout.setInsets(new Insets(10));
        gridLayout.setBackground(new Color(105, 89, 205));
        gridLayout.setBackgroundImage(new FillImage(
            new ResourceImageReference("/resource/image/fill/LightBlueLine.png")));
        petSearchCriteriaHeaderLabel.setLayoutData(gridLayout);
        petSearchCriteriaHeaderLabel.setStyleName("Label.Header");
        grid.add(petSearchCriteriaHeaderLabel);

        // Main concept attribute: name
        nameLabel = new Label(labels.getString("PetSearchForm.Label.PetSearchCriteria.Name"));
        gridLayout = new GridLayoutData();
        gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
        nameLabel.setLayoutData(gridLayout);
        grid.add(nameLabel);

        nameTextField = new TextField();
        nameTextField.setStyleName("Default");
        nameTextField.setWidth(new Extent(75, Extent.PERCENT));
        grid.add(nameTextField);

        // Main concept attribute: birth date
        birthDateLabel = new Label(labels.getString("PetSearchForm.Label.PetSearchCriteria.BirthDate"));
        gridLayout = new GridLayoutData();
        gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
        birthDateLabel.setLayoutData(gridLayout);
        grid.add(birthDateLabel);

```

```
75          Row birthDateRow = new Row();
76          birthDateRow.setCellSpacing(new Extent(10));
77          grid.add(birthDateRow);
78
79          Column birthDateCol1 = new Column();
80          birthDateCol1.setCellSpacing(new Extent(10));
81          birthDateRow.add(birthDateCol1);
82
83          Column birthDateCol2 = new Column();
84          birthDateCol2.setCellSpacing(new Extent(10));
85          birthDateRow.add(birthDateCol2);
86
87          birthDateTextField = new TextField();
88          birthDateTextField.setStyleName("Default");
89          birthDateTextField.setWidth(new Extent(300, Extent.PX));
90          birthDateCol1.add(birthDateTextField);
91
92          birthDateButton = new Button(labels.getString("Button.Select"));
93          birthDateButton.setStyleName("Button.Calendar");
94          birthDateButton.setActionCommand("birthDateButton");
95          birthDateButton.addActionListener(this);
96          birthDateCol2.add(birthDateButton);
97
98          // Action buttons
99          seekPetButton = new Button(labels.getString("PetSearchForm.seekPetButton"));
100         seekPetButton.setStyleName("Button.Default");
101         seekPetButton.setActionCommand("seekPetButton");
102         seekPetButton.addActionListener(this);
103         column.add(seekPetButton);
104     }
105
106     public void actionPerformed(ActionEvent e){
107
108         if (e.getActionCommand().equals("_closeButton")) {
109             presenter.closeCurrentPageAndFinalizeUseCase();
110         }
111
112         if (e.getActionCommand().equals("birthDateButton")) {
113             dateSelectionDialog = new DateSelectionDialog(labels);
114             dateSelectionDialog.addWindowPaneListener(this);
115             this.add(dateSelectionDialog);
116             _currentDateTextField = birthDateTextField;
117         }
118
119         if (e.getActionCommand().equals("seekPetButton")) {
120             populateDTOs();
121             presenter.setPetSearchCriteriaDTO(petSearchCriteriaDTO);
122             presenter.seekPetTriggered();
123         }
124     }
125
126     private void populateDTOs() {
127
128         if (nameTextField.getText() != null)
129             petSearchCriteriaDTO.setName(nameTextField.getText());
130
131         if (birthDateTextField.getText() != null) {
132             Date date = null;
133             SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
134             try {
135                 date = dateFormat.parse(birthDateTextField.getText());
136             } catch (ParseException e) {
137                 e.printStackTrace();
138             }
139             petSearchCriteriaDTO.setBirthDate(date);
140         }
141     }
142
143     /* ... */
144 }
145
```

W przypadku drugiej klasy ekranowej dla przypadku „Find pet”, czyli `PetListScreen`, jej głównym elementem prezentowanym użytkownikowi jest lista odpowiadająca pojęciu „pet list” typu „List View”. Generowany kod jest w tym przypadku nieco bardziej złożony. Lista reprezentowana jest przez obekt `petListTable` klasy `nextapp.echo.app.Table`. Jest on tworzony w metodzie `addContent()` (Listing 6, wiersze 15-30), jednak jego pełna

inicjalizacja wymaga trzech dodatkowych klas, które muszą być wygenerowane w toku transformacji. Są to: `TableHeaderRenderer` – klasa definiująca sposób renderowania nagłówków tabeli, `TableRenderer` – klasa definiująca sposób renderowania komórek tabeli dla różnych typów wartości oraz `PetListTableModel` – klasa dostarczająca dane do wyświetlane w poszczególnych kolumnach i wierszach tabeli. Dwie pierwsze wymienione klasy są generowane statycznie dla wszystkich list, natomiast trzecia jest generowana na podstawie konkretnego pojęcia „ListView"; w omawianym przypadku – „pet list". Kod klasy `PetListTableModel` zaprezentowany jest na Listing 7. Jak widać w metodzie `addRows()`, dane dostarczane są w postaci listy obiektów `PetListItemDTO`. Sama metoda wywoływana jest, natomiast, w kodzie metody `populateControls()` klasy `PetListScreen` (Listing 6, wiersze 63-66).

Oprócz listy `petListTable`, ważnymi elementami klasy `PetListScreen` są przyciski do wyzwalania przypadków użycia odpowiadające zdaniom „invoke" w głównym scenariuszu przypadku „Find pet", tj. „Create new visit", „Add new pet" oraz „Show visits for pet". Kod dodający przyciski do ekranu wygenerowany został w metodzie `addContent()`. Jego fragment, dla przycisku `createNewVisitButton`, widoczny jest w wierszach 36-40 na Listing 6. Kod wykonywany w odpowiedzi na kliknięcie tych przycisków, zawarty jest, z kolei, w metodzie `actionPerformed()` (wiersze 46-61). O ile przypadku przycisku `addNewPetButton`, kod ten prowadza się do wywołania bezparametrowej metody prezentera, o tyle, w przypadku dwóch pozostałych przycisków, metoda prezentera wymaga przekazania konkretnej wartości `petId`, gdyż wywoływane przypadki użycia wymagają parametru wejściowego. Stąd, wywołanie metody prezentera może nastąpić tylko wtedy, gdy użytkownik zaznaczy na liście `petListTable` wiersz reprezentujący konkretny obiekt typu `PetListItemDTO` przechowywany w obiekcie `petListTableModel`.

Listing 6 – Kod klasy PetListScreen

```
1  public class PetListScreen extends ContentPane implements ActionListener {
2
3      /* ... */
4
5      private void addContent(){
6
7          /* ... */
8
9          // Add content for 'pet list' data view
10
11         // Data view header
12
13         /* ... */
14
15         // Add table
16         petListTableModel = new PetListTableModel(labels);
17         petListSelectionModel = new DefaultListSelectionModel();
```

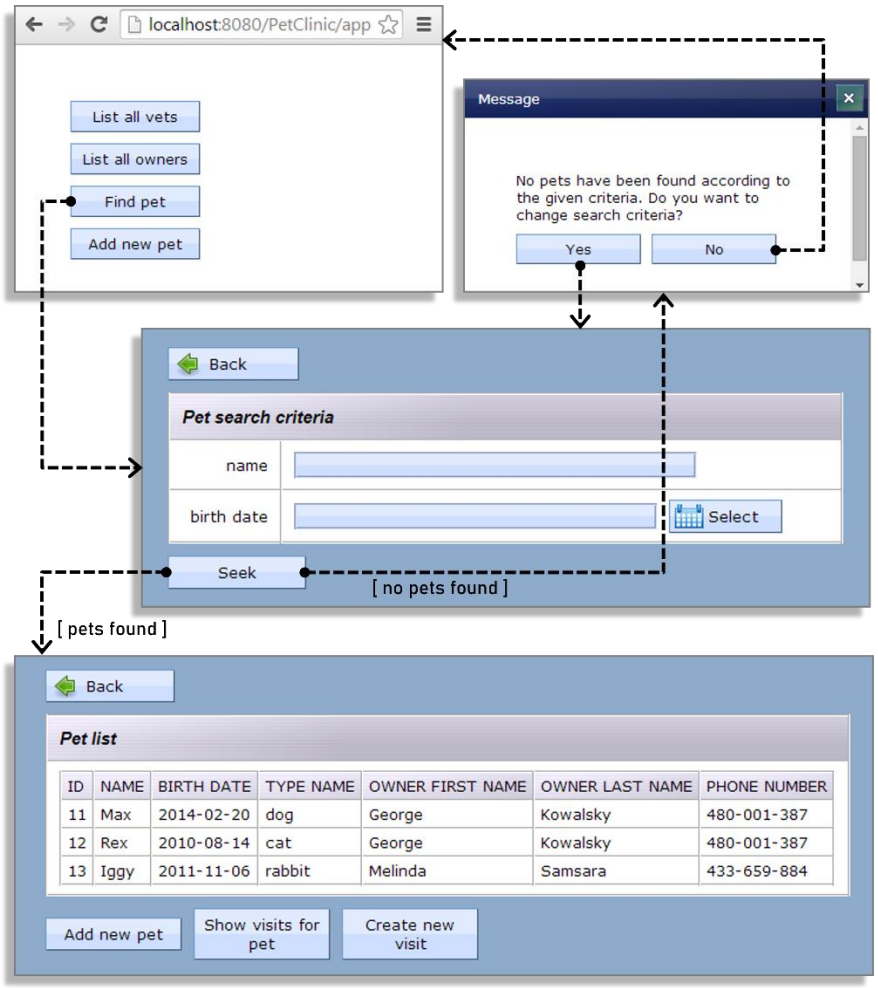
```
        petListSelectionModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        petListTable = new Table(petListTableModel);
        petListTable.setDefaultHeaderRenderer(new TableHeaderRenderer());
        petListTable.setDefaultRenderer(Object.class, new TableRenderer());
        petListTable.setSelectionModel(petListSelectionModel);
        petListTable.setSelectionEnabled(true);
        petListTable.setStyleName("Default");
        petListTable.setRolloverEnabled(true);
        gridLayout = new GridLayoutData();
        gridLayout.setColumnSpan(2);
        petListTable.setLayoutData(gridLayout);
        grid.add(petListTable);

        //// Add invoke buttons
        Row row = new Row();
        row.setCellSpacing(new Extent(10));

        createNewVisitButton = new Button(labels.getString("PetListScreen.createNewVisit"));
        createNewVisitButton.setStyleName("Button.Default");
        createNewVisitButton.setActionCommand("createNewVisitButton");
        createNewVisitButton.addActionListener(this);
        row.add(createNewVisitButton);

        /* ... */

    }

    public void actionPerformed(ActionEvent e){

        if (e.getActionCommand().equals("addNewPetButton")) {
            presenter.invokeAddNewPet();
        }

        if (e.getActionCommand().equals("createNewVisitButton")) {
            int row = petListSelectionModel.getMinSelectedIndex();
            if (row != -1) {
                long petId = petListTableModel.getRow(row).getPetID();
                presenter.invokeCreateNewVisit(petId);
            }
        }

        /* ... */
    }

    private void populateControls() {
        if (petListDTO != null)
            petListTableModel.addRows(petListDTO);
    }

}
```

Listing 7 – Kod klasy PetListTableModel

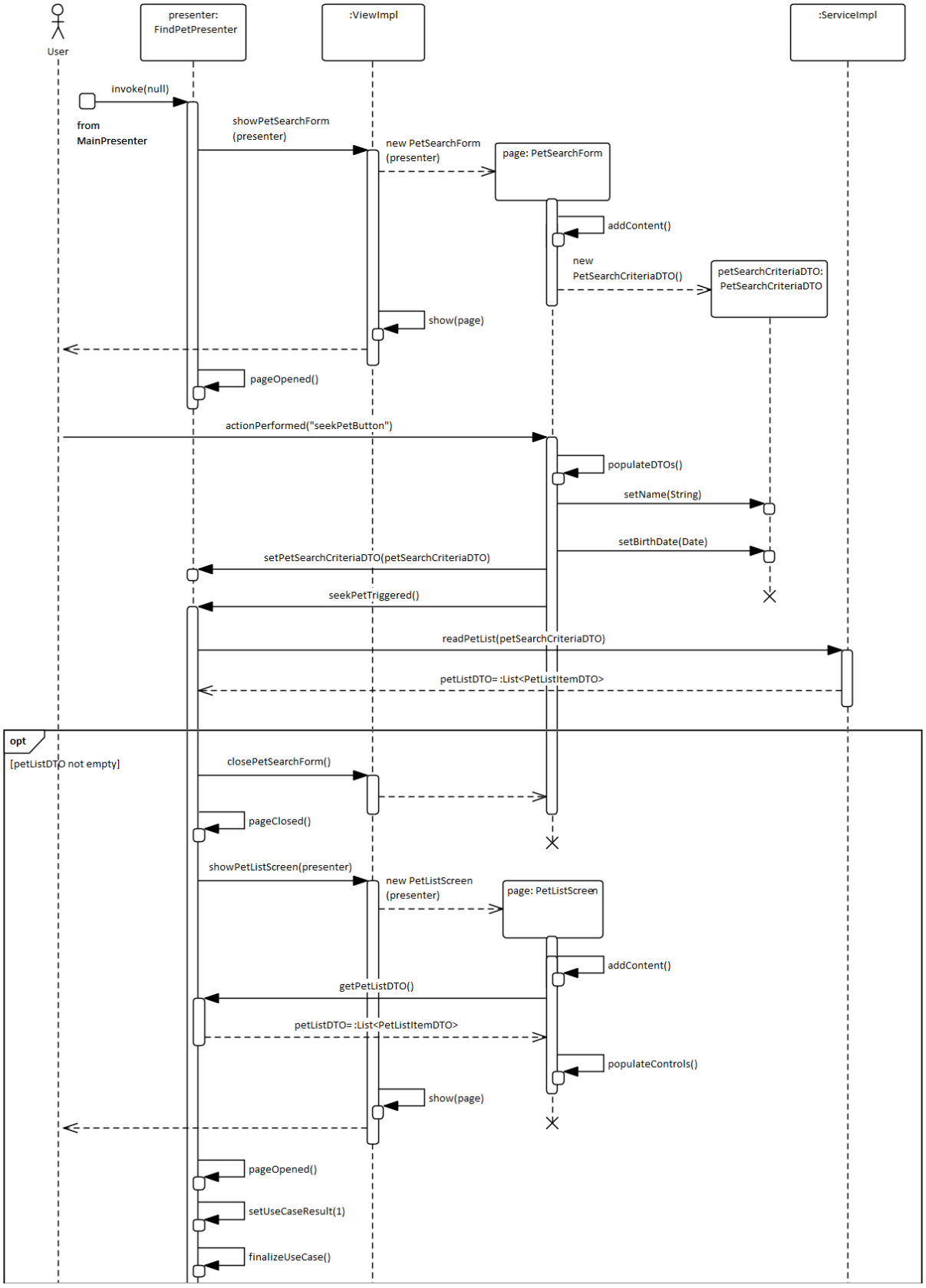
```
public class PetListTableModel extends AbstractTableModel {

    /* ... */

    public void addRows(List<PetListItemDTO> rows){
        this.rows = rows;
        fireTableDataChanged();
    }

    public List<PetListItemDTO> getRows() {
        return rows;
    }

    public PetListItemDTO getRow(int row) {
        return rows.get(row);
    }

    public void deleteRow(int row){
        rows.remove(row);
        fireTableRowsDeleted(row, row);
    }

    public String getColumnName(int column) {
        switch (column) {
            case 0: return labels.getString("PetListTableModel.ColumnName.Id");
            case 1: return labels.getString("PetListTableModel.ColumnName.Name");
            case 2: return labels.getString("PetListTableModel.ColumnName.BirthDate");
            case 3: return labels.getString("PetListTableModel.ColumnName.TypeName");
```

```
29              case 4: return labels.getString("PetListTableModel.ColumnName.OwnerFirstName");
30              case 5: return labels.getString("PetListTableModel.ColumnName.OwnerLastName");
31              case 6: return labels.getString("PetListTableModel.ColumnName.PhoneNumber");
32              default: return null;
33          }
34      }
35
36      @Override
37      public Object getValueAt(int column, int row) {
38          PetListItemDTO tableRow = rows.get(row);
39          switch (column) {
40              case 0: return tableRow.getPetID();
41              case 1: return tableRow.getName();
42              case 2: return tableRow.getBirthDate();
43              case 3: return tableRow.getTypeName();
44              case 4: return tableRow.getOwnerFirstName();
45              case 5: return tableRow.getOwnerLastName();
46              case 6: return tableRow.getPhoneNumber();
47              default: return null;
48          }
49      }
50
51  }
```

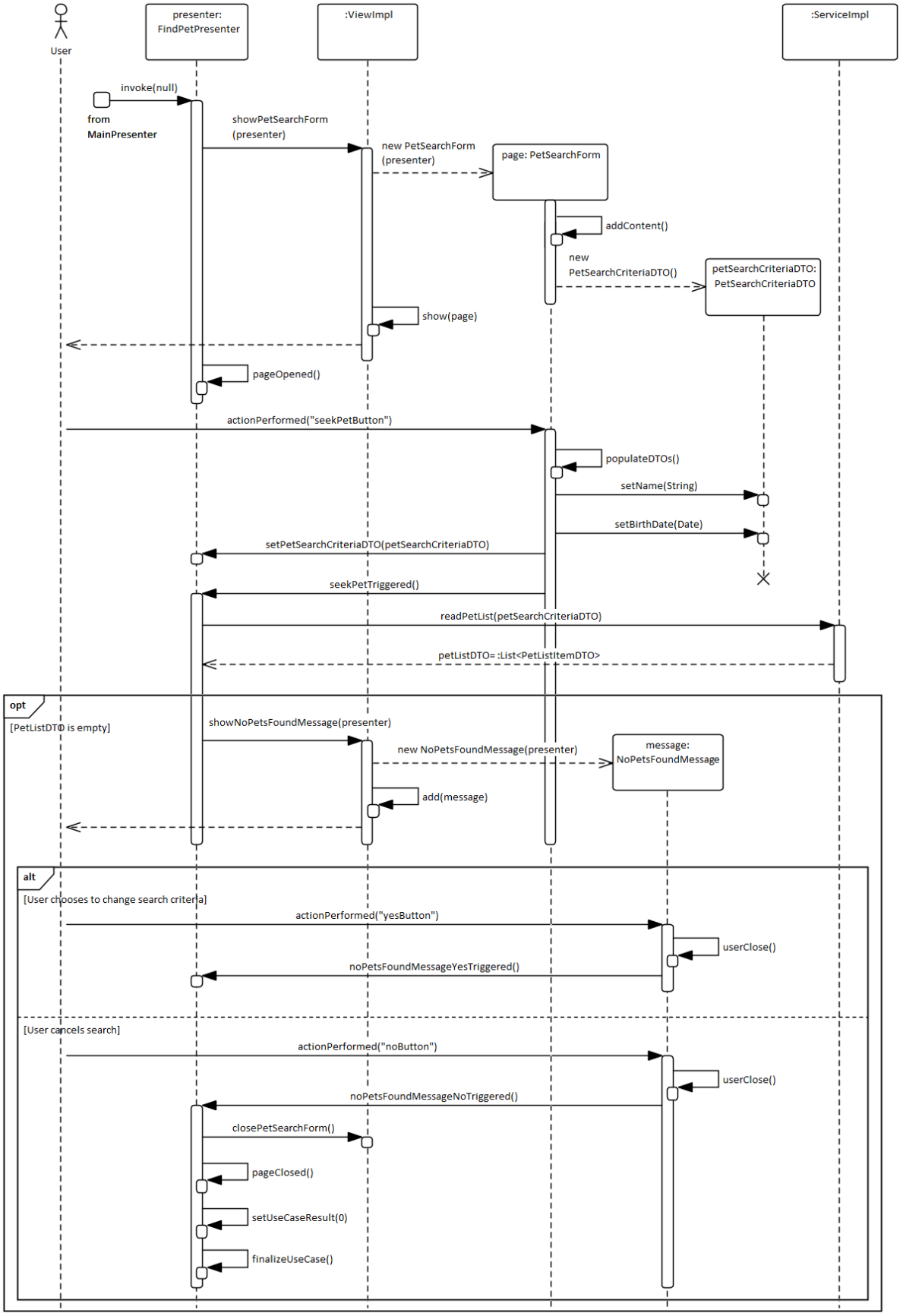
Sposób renderowania wszystkich ekranów dla przypadku użycia „Find pet” przedstawiony jest na Rysunek 128. Przedstawiony jest tu również cały przepływ sterowania pomiędzy poszczególnymi ekranami, zgodnie ze scenariuszami przypadku użycia, począwszy od ekranu głównego, z poziomu którego użytkownik uruchamia przypadek użycia.



Rysunek 128 – Interfejs użytkownika dla przypadku użycia Find pet



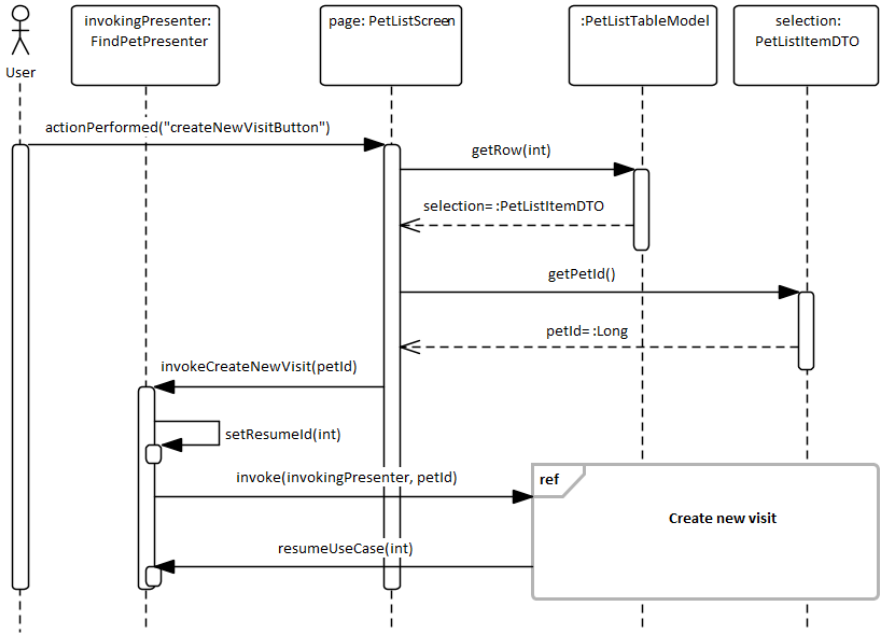
Rysunek 129 – Diagram sekwencji dla przypadku użycia Find pet – scenariusz główny



Rysunek 130 – Diagram sekwencji dla przypadku użycia Find pet – scenariusze alternatywne

Wymiana komunikatów pomiędzy klasami biorącymi udział w realizacji przypadku użycia „Find pet” przedstawiona jest w postaci diagramów sekwencji na Rysunek 129 oraz Rysunek 130. Pierwszy z nich prezentuje przebieg scenariusza głównego („Main scenario”), drugi – przebieg scenariuszy alternatywnych („Alternate – No pets found” oraz „Alternate – Search canceled”). Przedstawiona sekwencja komunikatów jest zasadniczo zgodna z założeniami przyjętymi dla środowiska translacyjnego, omówionymi w Rozdziale 5.1 dla analogicznego przypadku użycia „Search patient” (patrz: Rysunek 51). Drobne różnice wynikają ze sposobu implementacji reguł translacji dla środowiska docelowego.

W przypadku pomyślnego wykonania głównego scenariusza przypadku „Find pet”, użytkownik ma możliwość ręcznego wywołania trzech kolejnych przypadków użycia. Wygenerowane fragmenty kodu realizującego mechanizm wywołania jednego z tych przypadków – „Create new visit” – został omówiony nieco wcześniej. Dynamikę tego wywołania obrazuje, natomiast, diagram sekwencji na Rysunek 131. W odróżnieniu od abstrakcyjnego przykładu przedstawionego w opisie środowiska translacyjnego w Rozdziale 5.1, pokazany jest tu szerszy kontekst takiego wywołania, związany z przekazaniem parametru wejściowego do przypadku wywoływanego.



Rysunek 131 – Diagram sekwencji dla przypadku użycia Find pet – wywołanie przypadku użycia „Create new visit”

Jak widzieliśmy w scenariuszach przypadku użycia „Create new visit” (Rysunek 122a), parametr wejściowy zdefiniowany w zdaniu „precondition” stanowi kontekst dla operacji

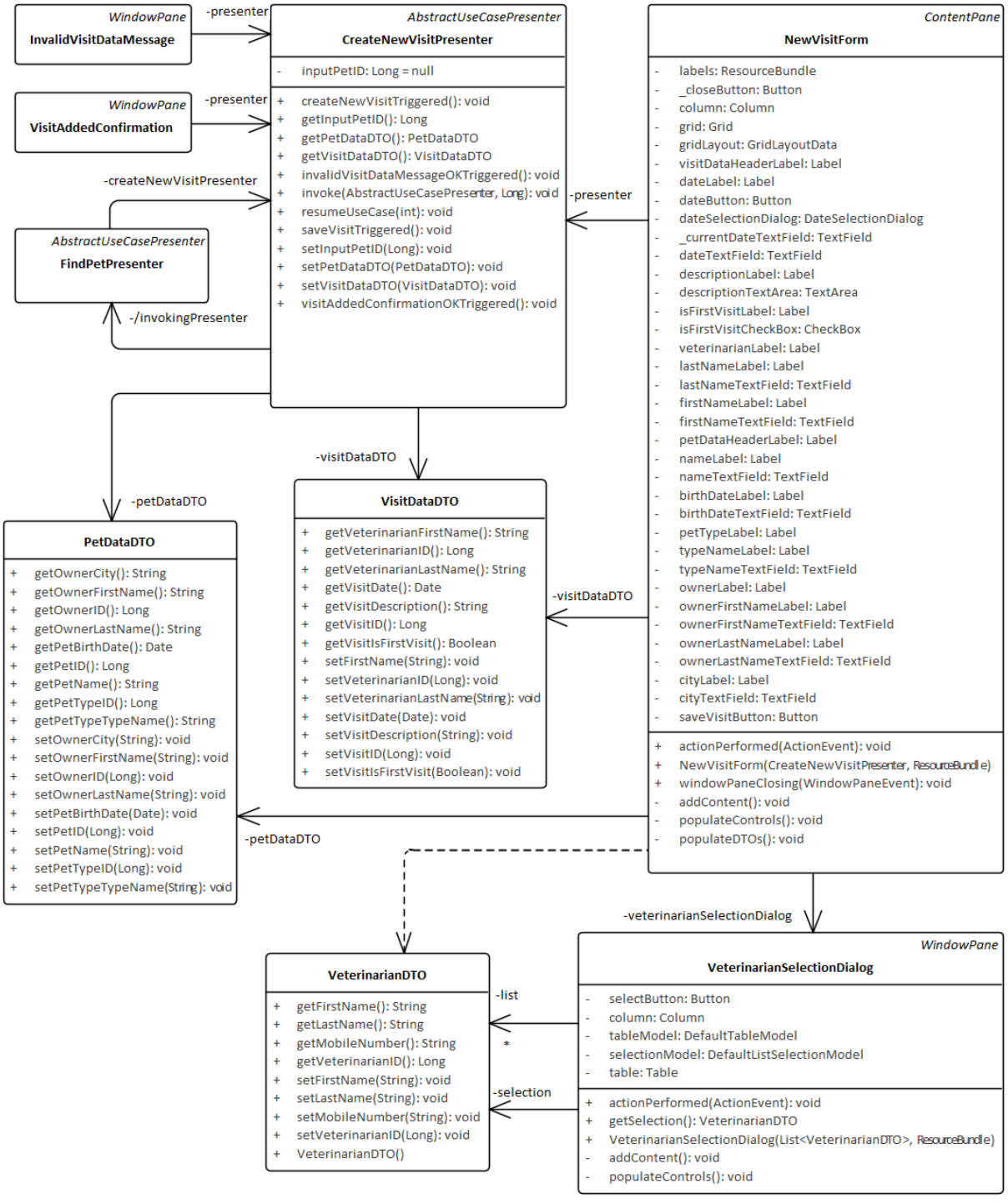
wyrażonych zdaniami System-to-SimpleView. Zdania te przetłumaczone zostały na operacje interfejsu `IService`, których wywołanie z poziomu prezentera wymaga przekazania parametru `petId`. Wartość tego parametru przekazywana jest do obiektu prezentera `CreateNewVisitPresenter` podczas wywołania przypadku użycia jako drugi parametr metody `invoke()` i przypisywana do atrybutu `inputPetId`. Elementy te, wygenerowane zgodnie z regułą P1, widoczne są we fragmencie kodu klasy `CreateNewVisitPresenter` na Listing 8. W zaprezentowanym kodzie widać również wywołania wspomnianych metod logiki biznesowej, do których przekazywany jest parametr petId: `readPetData()` (wiersz 12), `validateVisitData()` (wiersz 19) oraz `createVisitData()` (wiersz 21). Kod tych wywołań jest efektem wykonania reguły P9.

Listing 8 – Kod klasy CreateNewVisitPresenter

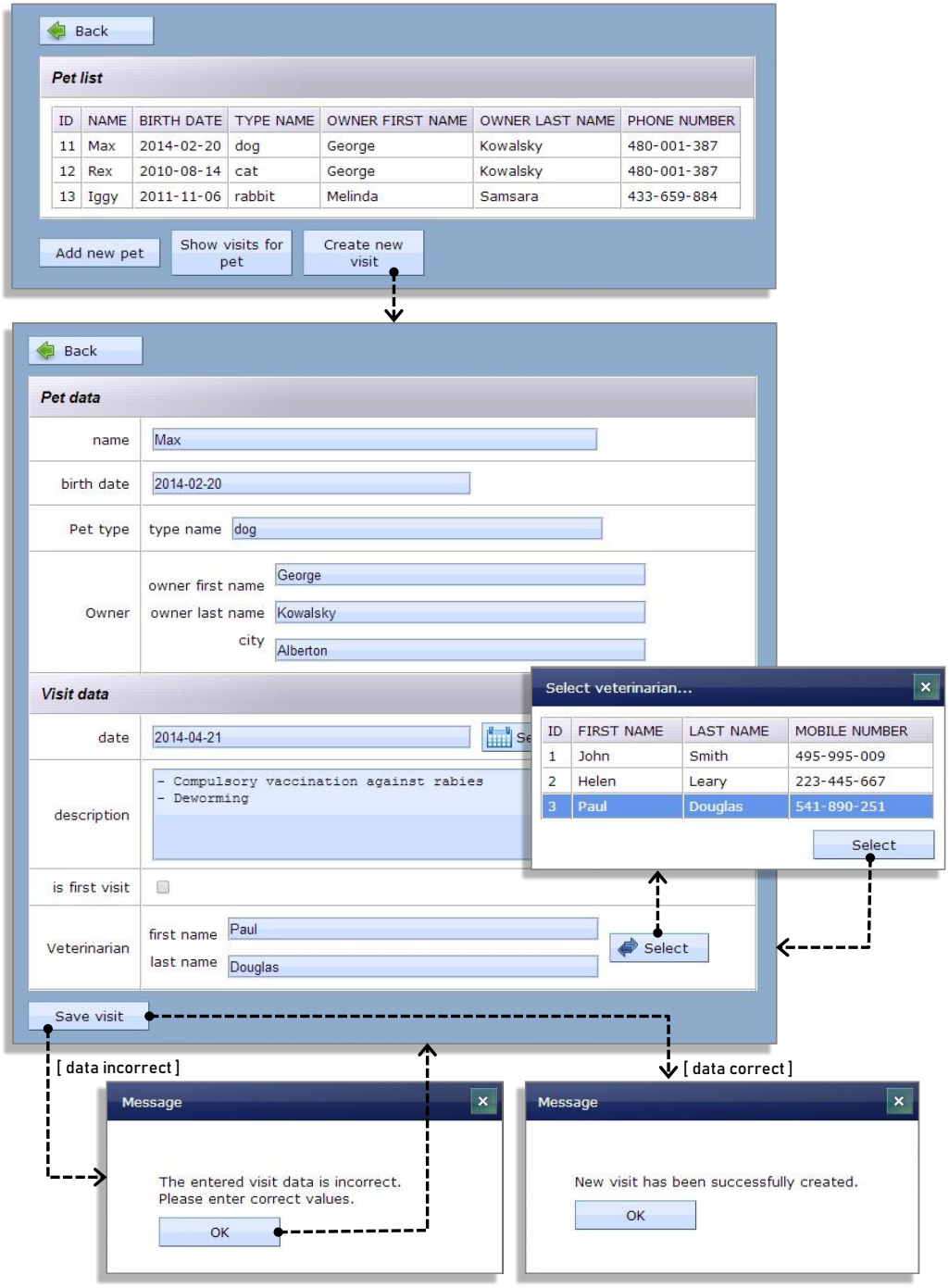
```
1  public class CreateNewVisitPresenter extends AbstractUseCasePresenter {
2
3      private Long inputPetId = null;
4      private PetDataDTO petDataDTO = null;
5      private VisitDataDTO visitDataDTO = null;
6
7      /* ... */
8
9      public void invoke(AbstractUseCasePresenter invokingPresenter, Long inputPetId){
10         super.invoke(invokingPresenter);
11         this.inputPetId = inputPetId;
12         petDataDTO = service.readPetData(this.inputPetId);
13         view.showNewVisitForm(this);
14         pageOpened();
15     }
16
17     public void saveVisitTriggered(){
18         int result6;
19         result6 = service.validateVisitData(visitDataDTO, inputPetId);
20         if (result6 == 1) { /* data correct [1] */
21             service.createVisitData(visitDataDTO, inputPetId);
22             view.showVisitAddedConfirmation(this);
23         }
24         else if (result6 == 0) { /* data incorrect [0] */
25             view.showInvalidVisitDataMessage(this);
26         }
27
28     }
29
30     public List<VeterinarianDTO> getVeterinarianList() {
31         return service.getVeterinarianList();
32     }
33
34     /* ... */
35 }
```

W kodzie wygenerowanym dla przypadku użycia „Create new visit” warto również zwrócić uwagę na klasy warstwy widoku. W modelu dziedzinowym dla omawianego przypadku (Rysunek 122b) występuje tylko jedno pojęcie typu „screen”, któremu odpowiada klasa `NewVisitForm`. Fakt powiązania tego pojęcia z dwoma widokami danych („pet data” oraz „visit data”) oraz charakterystyka widoku „visit data” sprawiają, że kod klasy `NewVisitForm` jest nieco bardziej rozbudowany niż w przypadku klas ekranowych dla przypadku użycia „Find

pet”. Wszystkie klasy ekranowe, wraz z zależnościami, przedstawia Rysunek 132, natomiast sposób renderowania tych elementów przedstawia Rysunek 133.



Rysunek 132 – Diagram klas dla przypadku użycia Create new visit



Rysunek 133 – Interfejs użytkownika dla przypadku użycia Create new visit

Zgodnie z regułą V6, każdemu widokowi danych powiązanemu z pojęciem ekranowym, w klasie odpowiadającej temu pojęciu generowane są atrybuty wskazujące na odpowiednie obiekty transferu danych. W naszym przypadku w klasie `NewVisitForm` wygenerowane zostały dwa atrybuty: `petDataDTO` oraz `visitDataDTO`. Rola obu obiektów jest nieco inna, ze względu na kierunek relacji łączących widoki danych z pojęciem ekranowym w modelu

źródłowym. Rolą obiektu typu `PetDataDTO` jest przekazanie danych zwierzaka, dla którego ma być umówiona wizyta, do wyświetlenia na ekranie. Rola obiektu typu `VisitDataDTO` jest odwrotna – transferuje on do prezentera dane wizyty wprowadzone przez użytkownika. pobranych z warstwy modelu przez obiekt prezentera pierwszego z nich jest przekazanie danych odpowiadających pojęciu „pet data". Mapowanie danych dostarczanych w obiekcie DTO na zawartość kontrolek ekranowych i odwrotnie realizowane jest w metodach `populateControls()` oraz `populateDTOs()` generowanych zgodnie z regułą V7. W omawianym przykładzie, klasa `NewVisitForm` musi implementować obie te metody. Ich kod widoczny jest na Listing 9 (wiersze 5-50).

Analizując kod metody `populateDTOs()`, należy zauważyć, że nie obejmuje on uzupełnienia obiektu DTO wartościami podanymi w kontrolkach wygenerowanych dla atrybutów pojęcia „veterinarian", wskazywanych przez widok danych „visit data". W przyjętym środowisku konkretnym, kontrolki te, a także odpowiadające im atrybuty w obiekcie `visitDataDTO`, uzupełniane są w momencie zamknięcia okna dialogowego `VeterinarianSelectionDialog`, w którym użytkownik wybiera weterynarza z listy. Kod obsługujący zdarzenie zamknięcia okna dialogowego znajduje się w metodzie `windowPaneClosing()` w klasie `NewVisitForm` (Listing 9, wiersze 63-72). Widok okna dialogowego wraz z przykładowymi danymi widoczny jest na Rysunek 133 a kod klasy implementującej to okno przedstawia Listing 10. Kod tej klasy ma bardzo podobną strukturę do klas ekranowych, z tym, że jest rozszerzeniem klasy `nextapp.ech.app.WindowPane`, która w środowisku Echo3 służy do wyświetlania okien dialogowych.

Listing 9 – Kod klasy NewVisitForm

```
1  public class NewVisitForm extends ContentPane implements ActionListener, WindowPaneListener {
2
3      /* ... */
4
5      private void populateControls() {
6
7          if (petDataDTO == null)
8              return;
9
10         if (petDataDTO.getBirthDate() != null) {
11             SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
12             birthDateTextField.setText(dateFormat.format(petDataDTO.getBirthDate()));
13         }
14
15         if (petDataDTO.getName() != null)
16             nameTextField.setText(petDataDTO.getName());
17
18         if (petDataDTO.getTypeName() != null)
19             typeNameTextField.setText(petDataDTO.getTypeName());
20
21         if (petDataDTO.getOwnerFirstName() != null)
22             ownerFirstNameTextField.setText(petDataDTO.getOwnerFirstName());
23
24         if (petDataDTO.getOwnerLastName() != null)
25             ownerLastNameTextField.setText(petDataDTO.getOwnerLastName());
```

```
26
27          if (petDataDTO.getCity() != null)
28              cityTextField.setText(petDataDTO.getCity());
29      }
30
31      private void populateDTOs() {
32
33          if (dateTextField.getText() != null) {
34              Date date = null;
35              SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
36              try {
37                  date = dateFormat.parse(dateTextField.getText());
38              } catch (ParseException e) {
39                  e.printStackTrace();
40              }
41              visitDataDTO.setDate(date);
42          }
43
44          visitDataDTO.setIsFirstVisit(isFirstVisitCheckBox.isSelected());
45
46          if (descriptionTextArea.getText() != null)
47              visitDataDTO.setDescription(descriptionTextArea.getText());
48
49          // Attributes owned by related concepts are populated in DTO while closing selection dialog.
50      }
51
52      public void windowPaneClosing(WindowPaneEvent e) {
53
54          if (e.getSource() instanceof DateSelectionDialog) {
55              Date selection = dateSelectionDialog.getSelection();
56              if (selection != null) {
57                  SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
58                  if (_currentDateTextField != null)
59                      _currentDateTextField.setText(dateFormat.format(selection));
60              }
61          }
62
63          if (e.getSource() instanceof VeterinarianSelectionDialog) {
64              VeterinarianDTO selection = veterinarianSelectionDialog.getSelection();
65              if (selection != null) {
66                  visitDataDTO.setVeterinarianID(selection.getVeterinarianID());
67                  visitDataDTO.setVeterinarianFirstName(selection.getFirstName());
68                  visitDataDTO.setVeterinarianLastName(selection.getLastName());
69                  this.firstNameTextField.setText(selection.getFirstName());
70                  this.lastNameTextField.setText(selection.getLastName());
71              }
72          }
73      }
74
75      /* ... */
76  }
```

Listing 10 – Kod klasy VeterinarianSelectionDialog

```
1   public class VeterinarianSelectionDialog extends WindowPane implements ActionListener {
2
3       /* ... */
4
5       public VeterinarianSelectionDialog(List<VeterinarianDTO> veterinarianList, ResourceBundle labels) {
6
7           // Set style
8           this.setStyleName("Default");
9
10          // Set labels resource bundle
11          this.labels = labels;
12
13          // Add column layout
14          column = new Column();
15          column.setInsets(new Insets(10));
16          column.setCellSpacing(new Extent(10));
17          add(column);
18
19          // Add message content
20          setTitle(labels.getString("VeterinarianSelectionDialog.Title"));
21          setModal(true);
22          addContent();
23
24          // Add Select button
25          selectButton = new Button(labels.getString("Button.Select"));
26          selectButton.setStyleName("Button.Default");
27          selectButton.setActionCommand("selectButton");
28          selectButton.addActionListener(this);
```

```
29          ColumnLayoutData clRight = new ColumnLayoutData();
30          clRight.setAlignment(Alignment.ALIGN_RIGHT);
31          selectButton.setLayoutData(clRight);
32          column.add(selectButton);
33
34          // Set DTO to be displayed
35          this.veterinarianList = veterinarianList;
36          populateControls();
37      }
38
39      private void populateControls() {
40
41          tableModel.setColumnCount(4);
42          tableModel.setColumnName(0, labels.getString("VeterinarianSelectionDialog.ColumnName.Id"));
43          tableModel.setColumnName(1, labels.getString("VeterinarianSelectionDialog.ColumnName.FirsName"));
44          tableModel.setColumnName(2, labels.getString("VeterinarianSelectionDialog.ColumnName.LastName"));
45          tableModel.setColumnName
46              (3, labels.getString("VeterinarianSelectionDialog.ColumnName.MobileNumber"));
47
48          if (veterinarianList != null) {
49              tableModel.setRowCount(veterinarianList.size());
50              for (int i = 0; i < veterinarianList.size(); i++) {
51                  tableModel.setValueAt(veterinarianList.get(i).getVeterinarianID(), 0, i);
52                  tableModel.setValueAt(veterinarianList.get(i).getFirstName(), 1, i);
53                  tableModel.setValueAt(veterinarianList.get(i).getLastName(), 2, i);
54                  tableModel.setValueAt(veterinarianList.get(i).getMobileNumber(), 3, i);
55              }
56          }
57      }
58
59      private void addContent() {
60          // Add table
61          tableModel = new DefaultTableModel();
62          selectionModel = new DefaultListSelectionModel();
63          selectionModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
64          table = new Table(tableModel);
65          table.setStyleName("Default");
66          table.setSelectionModel(selectionModel);
67          table.setSelectionEnabled(true);
68          table.setDefaultHeaderRenderer(new TableHeaderRenderer());
69          table.setDefaultRenderer(Object.class, new TableRenderer());
70          table.setRolloverEnabled(true);
71          table.setWidth(new Extent(100, Extent.PERCENT));
72          column.add(table);
73      }
74
75      @Override
76      public void actionPerformed(ActionEvent e) {
77
78          if (e.getActionCommand().equals("selectButton")) {
79              int row = selectionModel.getMinSelectedIndex();
80              if (row != -1) {
81                  selection = veterinarianList.get(row);
82              }
83              userClose();
84          }
85      }
86
87      public VeterinarianDTO getSelection() {
88          return selection;
89      }
90
91  }
```

8 Ewaluacja rozwiązania

W celu dokonania oceny działania opracowanego podejścia oraz zbadania jego efektywności w porównaniu z tradycyjnymi metodami wytwarzania oprogramowania, wykonano studium przypadku a także przeprowadzono kontrolowany eksperyment z udziałem studentów.

Głównym celem przeprowadzonego studium przypadku (opisanego w rozdziale 7) była praktyczna weryfikacja poprawności zdefiniowanych reguł semantycznych oraz ich implementacji w postaci transformacji „RSL to Java" – w szczególności weryfikacja czy spełnione zostały założenia projektowe dla transformacji (opisane w rozdziale 6). W tym celu stworzona została pełna specyfikacja wymagań w języku RSL dla przykładowego systemu Pet Clinic. Istotnym założeniem było wykorzystanie wszystkich konstrukcji języka RSL, dla których zdefiniowana została semantyka translacyjna. Dla przygotowanej specyfikacji wymagań wykonana została transformacja „RSL to Java". Wynikowy kod źródłowy został przeniesiony do przygotowanego wcześniej projektu w środowisku programistycznym z zaimportowanymi bibliotekami wymaganymi przez docelowe środowisko aplikacyjne. Zgodnie z założeniami, efektem działania transformacji „RSL to Java" jest kod w warstwie prezentera (logika aplikacji) oraz w warstwie widoku. Stąd, weryfikacja poprawności działania docelowej aplikacji wymagała ręcznego uzupełnienia kodu w warstwie modelu, w tym kodu metod dostępowych do warstwy persystencji danych; w tym przypadku – relacyjnej bazy danych. Tak przygotowany kod został skompilowany w celu uruchomienia aplikacji. W celu weryfikacji poprawności jej działania, wykonany został szereg scenariuszy testowych odpowiadających wszystkim zdefiniowanym w specyfikacji scenariuszom przypadków użycia, z wykorzystaniem odpowiednio spreparowanych danych testowych. Pozwoliło to potwierdzić zgodność wygenerowanej aplikacji z wymaganiami wyrażonymi modelem źródłowym.

Studium przypadku pokazało również istotną zmianę w poziomie złożoności artefaktów jakie należy utworzyć w celu uzyskania produktu finalnego w postaci działającego systemu. Przykładowo, specyfikacja przypadku użycia „Find pet" (Rysunek 121) składa się z 20 zdań scenariuszy, 16 elementów modelu dziedziny oraz 22 łączących je relacji. Kod wygenerowany dla tego przypadku użycia, nie licząc kodu wygenerowanego statycznie, składa się z 7 klas zawierających ponad 100 atrybutów oraz metod (Rysunek 127). Każda z metod zawiera dodatkowo od kilku do kilkudziesięciu wierszy kodu. Można na tej podstawie zakładać, że zaprezentowane podejście pozwala zwiększyć efektywność procesu wytwarzania oprogramowania w porównaniu z podejściem tradycyjnym, w którym kod realizujący

wymagania tworzony jest ręcznie. Aby ująć to ilościowo w sposób bardziej precyzyjny, przyjęliśmy metrykę będącą stosunkiem liczby konstrukcji w języku RSL do liczby odpowiadających mu logicznych linii kodu (Logical Lines of Code – LLOC)[33] w języku Java. Do wyznaczenia liczby konstrukcji w języku RSL dla systemu Pet Clinic, za pojedynczą konstrukcję języka przyjęliśmy:

- utworzenie aktora,
- utworzenie przypadku użycia,
- utworzenie relacji pomiędzy przypadkami użycia,
- utworzenie relacji pomiędzy aktorem i przypadkiem użycia,
- utworzenie zdania dowolnego typu w scenariuszu,
- utworzenie pojęcia dziedzinowego dowolnego typu,
- utworzenie relacji pomiędzy pojęciami dziedzinowymi,
- utworzenie atrybutu pojęcia dziedzinowego,
- utworzenie relacji pomiędzy pojęciem dziedzinowym i atrybutem,
- dodanie właściwości pojęcia dziedzinowego (np. dodanie treści komunikatu dla pojęcia typu Message).

Jako konstrukcji języka nie uwzględniono dodawania stwierdzeń dziedzinowych do pojęć dziedzinowych, gdyż są one dodawane automatycznie w edytorze języka RSL w trakcie tworzenia zdań scenariuszy. Nie uwzględniono również tworzenia pakietów stanowiących strukturę specyfikacji wymagań, podobnie jak w przypadku wyznaczania liczby konstrukcji w języku Java nie uwzględniono tworzenia pakietów kodu stanowiących strukturę projektu. Do wyznaczenia wartości LLOC w wygenerowanym kodzie użyto narzędzia Project Code Meter[34]. Uwzględniono przy tym całość kodu będącego rezultatem działania transformacji „RSL to Java" (tj. kod wygenerowany statycznie oraz dynamicznie, w oparciu o reguły translacji), pominięto natomiast kod uzupełniony ręcznie w warstwie modelu. Obliczona tym sposobem liczba konstrukcji języka RSL dla kompletnej specyfikacji wymagań systemu Pet Clinic wyniosła 405, natomiast wartość LLOC dla wygenerowanego kodu wyniosła 4187. Daje to istotną – o rząd wielkości – redukcję złożoności dla artefaktów jakie należy utworzyć w celu uzyskania finalnego produktu procesu wytwarzania oprogramowania. Jest to redukcja podobna

[33] LLOC – metryka rozmiaru oprogramowania wyrażona liczbą instrukcji danego języka programowania

[34] https://www.projectcodemeter.com

do tej jaką zapewniają języki programowania trzeciej generacji w porównaniu do języka maszynowego [167]. Przykładowo, średnia liczba instrukcji języka Java niezbędna do zaimplementowania jednego punktu funkcyjnego[35] wynosi 53, podczas gdy w przypadku języka maszynowego potrzeba średnio 640 instrukcji.

Dodatkową ewaluację zaproponowanego podejścia wykonano w ramach kontrolowanego eksperymentu podczas zajęć laboratoryjnych ze studentami. W eksperymencie udział wzięły dwie grupy studentów studiów stacjonarnych II stopnia na kierunku informatyka, uczęszczających na zajęcia Wytwarzanie oprogramowania sterowane modelami – łącznie 48 osób. Przed przystąpieniem do eksperymentu, studenci zdobyli wiedzę z zakresu MDSD oraz MDRE, w szczególności zapoznali się z językiem RSL oraz środowiskiem ReDSeeDS. Wszyscy studenci posiadali również ugruntowaną wiedzę z zakresu projektowania oraz implementacji aplikacji internetowych zgodnych ze wzorcem MVP/MVC z wykorzystaniem notacji UML oraz języka Java.

Eksperyment był realizowany tylko w trakcie trwania zajęć laboratoryjnych w przeciągu całego semestru. Najpierw studenci zostali podzieleni na 12 czteroosobowych zespołów. Każdy zespół otrzymał wstępne wymagania wobec systemu, który miał powstać w trakcie eksperymentu. Były to dwie proste aplikacje o zbliżonym poziomie złożoności: system obsługi przychodni lekarskiej oraz system zapisu na zajęcia w klubie fitness. W przypadku obu systemów wymagania miały postać modelu przypadków użycia składającego się z 2 aktorów oraz 10 przypadków użycia wraz z relacjami między nimi. Przypadki użycia nie zawierały scenariuszy a jedynie krótki opis oczekiwanego sposobu działania, z perspektywy użytkownika końcowego. Opisy te uwzględniały najważniejsze pojęcia dziedzinowe (encje biznesowe) dla każdego systemu oraz proste reguły biznesowe niezbędne do zaimplementowania logiki biznesowej systemu.

Pierwszym zadaniem było stworzenie kompletnej specyfikacji funkcjonalnej zawierającej szczegółowe scenariusze przypadków użycia oraz pełny model dziedzinowy. Sześć zespołów wykonywała to zadanie z wykorzystaniem języka RSL (z uwzględnieniem konwencji wymaganych przez transformację „RSL to Java") oraz edytora tego języka dostępnego

[35] Punkt funkcyjny – metryka złożoności oprogramowania podawana jako liczba bezwymiarowa określająca efektywną względną miarę wartości funkcji oferowanej przez program użytkownikowi. Najczęściej jest podstawą do oszacowania m.in. pracochłonności wytworzenia danego oprogramowania. Pojęcie to zaproponował pierwotnie Allen Albrecht z firmy IBM [170]. Zmodyfikowana wersja jest obecnie międzynarodowym standardem ISO/IEC 20926:2009 [171].

w narzędziu ReDSeeDS. Kolejne sześć zespołów posługiwało się notacją UML z wykorzystaniem standardowego narzędzia do modelowania – Enterprise Architect (EA), stosując powszechne dobre praktyk pisania scenariuszy przypadków użycia [168] [131]. W tym przypadku narzędzie nie narzucało żadnej konkretnej składni scenariuszy a pojęcia dziedzinowe zostały zamodelowane z wykorzystaniem diagramów klas. Stworzone klasy zostały powiązane ze zdaniami scenariuszy za pomocą mechanizmu hiperłączy dostępnych w EA.

Przygotowane specyfikacje podlegały odbiorowi przez prowadzącego zajęcia, który wcielał się w rolę zamawiającego system. Wszystkie zespoły były w stanie wytworzyć podobnej jakości specyfikacje stanowiące projekty funkcjonalne realizujące inicjalne wymagania. Sposób realizacji przypadków użycia został ujęty w postaci scenariuszy (głównego oraz alternatywnych) w liczbie od 20 do 23, w zależności od zespołu (od 1 do 4 scenariuszy dla pojedynczego przypadku użycia). Liczba zdań scenariuszy stworzonych przez każdy zespół oscylowała pomiędzy 119 a 146 (średnio 13 zdań dla każdego przypadku użycia). Średnia liczba zdań stworzonych przez zespoły korzystające z EA była nieznacznie większa niż w przypadku zespołów korzystających z edytora języka RSL. Może to wynikać większej swobody przy pisaniu scenariuszy w narzędziu EA, które nie narzuca konkretnej składni zdań w odróżnieniu od języka RSL. Ponadto, specyfikacje w języku RSL wymagały utworzenia szeregu dodatkowych relacji między pojęciami z dziedziny systemu oraz pojęciami z dziedziny problemu (np. relacji „main concept” czy „action param”), aby zapewnić poprawne działanie transformacji „RSL to Java”.

Kolejnym zadaniem była implementacja aplikacji działającej zgodnie z przygotowaną specyfikacją wymagań. Zespoły tworzące specyfikację wymagań w języku RSL miały wykorzystać transformację „RSL to Java” w celu wygenerowania kodu w warstwie logiki aplikacji oraz interfejsu użytkownika, a następnie ręcznie uzupełnić kod w warstwie modelu włącznie z kodem dostępu do warstwy persystencji danych. Zadaniem pozostałych zespołów była ręczna implementacja aplikacji z wykorzystaniem takiego samego stosu technologicznego oraz podstawowego wzorca architektonicznego (MVP) jak w przypadku transformacji „RSL to Java”. Zespoły te miały przy tym dowolność w zakresie wyboru konkretnych wzorców projektowych czy sposób implementacji poszczególnych fragmentów aplikacji. Wymogiem było natomiast stosowanie dobrych praktyk formatowania kodu w celu zapewnienia jego czytelności. Zespoły korzystające z EA mogły używać dostępnego w tym narzędziu generatora kodu. Uwzględniając powyższe założenia, celem każdego zespołu było zaimplementowanie jak

największej liczby scenariuszy przypadków użycia. Taka miara została podyktowana faktem, że z perspektywy zamawiającego oprogramowanie, produktywność zespołu deweloperskiego mierzona jest z reguły zdolnością do implementacji funkcjonalności dostarczającej pewną minimalną ale kompletną wartość biznesową dla użytkowników końcowych, nawet jeżeli nie wyczerpuje ona wszystkich możliwych alternatyw. Można zatem uznać, że najmniejszą jednostką funkcjonalną jest scenariusz (w przypadku metod zwinnych zwany historyjką użytkownika), który reprezentuje jeden ze sposobów realizacji przypadku użycia. Przy ocenie wyników pracy zespołów, scenariusz był uznawany za zrealizowany jeżeli poprawnie przechodził test akceptacyjny polegający na obserwacji zgodności działania systemu ze scenariuszem, przy zastosowaniu konkretnych danych testowych. Zespoły korzystające z transformacji „RSL to Java" zaimplementowały poprawnie ponad 70% scenariuszy, tj. 85 ze 128 scenariuszy zdefiniowanych w specyfikacjach wymagań. Najsłabszy zespół zaimplementował przy tym 12 a najlepszy 16 scenariuszy. Zespoły implementujące aplikacje w sposób tradycyjny zdołały zaimplementować 20% scenariuszy, tj. 26 ze 130 scenariuszy zdefiniowanych w specyfikacjach wymagań przygotowanych przez te zespoły. W tej grupie, najsłabszy zespół zaimplementował 2 scenariusze a najlepszy 6.

Ten prosty eksperyment pokazał wyraźny wzrost efektywności wytwarzania oprogramowania przy użyciu zaprezentowanego podejścia, w porównaniu z podejściem tradycyjnym. Wzrost mógłby być zapewne jeszcze lepszy w przypadku generowania kompletnego kodu we wszystkich warstwach aplikacji, np. z zastosowaniem rozszerzenia RSL-DL [53] pozwalającego generować kod logiki biznesowej. Należy także zwrócić na uwagę na inne czynniki mogące mieć wpływ na uzyskane wyniki. Po pierwsze, dobór składu zespołów powinien zapewniać jak najbardziej wyrównany poziom umiejętności związanych z wytwarzaniem oprogramowania. Wydaje się, że wpływ tego czynnika został częściowo zredukowany dzięki temu, że liderzy każdego z zespołów zostali wskazani na podstawie wyników osiągniętych w poprzednim semestrze studiów. Liderzy dobierali następnie pozostałych członków zespołów naprzemiennie spośród całej grupy studentów. Po drugie, wpływ na efektywność pracy mógł mieć poziom dojrzałości wykorzystanych narzędzi i notacji. Podczas, gdy EA oraz środowiska deweloperskie używane przez zespoły implementujące system manualnie, są narzędziami komercyjnymi o wysokiej stabilności i użyteczności, o tyle środowisko ReDSeeDS (w tym zakres konstrukcji języka RSL, dla którego zdefiniowana została semantyka translacyjna, jak również sama implementacja transformacji „RSL to Java") zostały stworzone na potrzeby badań i mają charakter prototypowy. Znalazło to wyraz

w uwagach zespołów zebranych w formie ankiet po zakończeniu eksperymentu. Jedynie zespoły używające środowiska ReDSeeDS zgłaszały uwagi dotyczące ich wydajności oraz problemów technicznych. Żaden z zespołów nie zgłosił przy tym, że mogło to mieć istotny wpływ na ogólną wydajność zespołu; wpływ ten oceniali jako niski.

Ocena pełnego wpływu przedstawionego rozwiązania na efektywność wytwarzania oprogramowania jest znacznie bardziej złożona. Dotyczy to między innymi takich kwestii, jak całościowe uwzględnienie metody wytwórczej używanej w kontekście opracowanego rozwiązania w projekcie wytwarzania oprogramowania oraz całym późniejszym cyklu życia oprogramowania, ekspresywność i łatwość opanowania języka RSL oraz powiązanych narzędzi, ocena sprawności posługiwania się nimi przez uczestników projektu, ocena jakości generowanego kodu – zarówno pod kątem jego późniejszego utrzymania i rozwoju, jak też pod kątem poziomu spełnienia faktycznych potrzeb końcowych użytkowników i ich satysfakcji z finalnego oprogramowania (badania użyteczność oprogramowania). Ocena takich aspektów powinna być przedmiotem przyszłych badań.

Konkludując, należy zauważyć, że niniejsza praca nie zawiera formalnego czy pełnego eksperymentalnego dowodu prawdziwości postawionej hipotezy badawczej, które wychodzą poza zakres tej pracy mającej charakter badawczy. Praktyczna weryfikacja przedstawionego rozwiązania i porównanie jej z ugruntowanymi i powszechnie stosowanymi metodami wytwarzania oprogramowania, wymagałaby istotnej rozbudowy języka RSL i semantyki translacyjnej oraz stworzenia narzędzia o jakości komercyjnej. Niemniej, przeprowadzone studium przypadku oraz eksperyment z udziałem studentów stanowi istotną przesłankę wskazującą, że możliwa jest znacząca poprawa efektywności procesu wytwarzania oprogramowania poprzez automatyczną generację kodu aplikacji bezpośrednio z modelu wymagań, z użyciem jednoetapowej transformacji.

9 Podsumowanie

W pracy zostało przedstawione rozwiązanie będące realizację idei MDRE. W opisanym podejściu wymagania w procesie wytwarzania oprogramowania traktowane są jako artefakty pierwszej kategorii, mające bezpośrednie powiązanie z kodem docelowego systemu i tym samym niwelują lukę semantyczną pomiędzy wymaganiami a implementującym je kodem. Formułowane są one w postaci precyzyjnych modeli w rozszerzonym języku RSL. Modele takie z jednej strony zapewniają odpowiedni poziom komunikatywności w procesie analizy wymagań, z drugiej strony – posiadają dobrze zdefiniowaną semantykę translacyjną, umożliwiającą ich bezpośrednie przekształcenie do kodu docelowej aplikacji dla konkretnej platformy technologicznej. Semantyka translacyjna została zdefiniowana w postaci zestawu reguł określających sposób mapowania konstrukcji języka RSL na konstrukcje języka Java stanowiące pełną implementację logiki aplikacji oraz interfejsu użytkownika docelowej aplikacji internetowej zgodnej z typowym wzorcem architektury warstwowej – MVP. Reguły translacji zostały zaimplementowane w postaci transformacji „RSL to Java" działającej środowisku ReDSeeDS, co pozwoliło na przeprowadzenie wstępnej ewaluacji rozwiązania. Uzyskane wyniki stanowią istotną przesłankę wskazującą, że opracowane rozwiązanie umożliwia istotną poprawę efektywności procesu wytwarzania oprogramowania w porównaniu do tradycyjnego procesu wytwórczego. Pełna ewaluacja rozwiązania w kontekście rzeczywistych projektów wymaga jednak dalszych badań i prac rozwojowych obejmujących zarówno język RSL, semantykę translacyjną dla tego języka oraz środowisko narzędziowe ReDSeeDS.

Biorąc po uwagę zdobyte doświadczenia i obserwacje poczynione w trakcie opracowywania i ewaluacji zaprezentowanego rozwiązania, można wskazać kilka najważniejszych kierunków jego dalszego rozwoju.

Po pierwsze – uwzględnienie wymagań pozafunkcjonalnych. Obecne rozwiązanie skupia się na wymaganiach funkcjonalnych opisujących oczekiwany sposób działania systemu. Wymagania pozafunkcjonalne mogą mieć, natomiast, istotny wpływ na finalny kod docelowego systemu. W procesie wytwarzania oprogramowania, na podstawie tego typu wymagań podejmowanych jest szereg decyzji mających wpływ na docelową architekturę systemu, wybór konkretnych technologii, kwestii związanych z wydajnością systemu, bezpieczeństwem przetwarzanych danych czy dostępnością cyfrową (np. zgodność ze

standardem WCAG[36]). Język RSL powinien umożliwiać definiowanie wymagań pozafunkcjonalnych, które mogłyby służyć jako parametry sterujące regułami translacji wymagań funkcjonalnych do kodu oraz jako parametry sterujące wykonaniem samej transformacji, decydujące o aspektach związanych ze złożonością incydentalną.

Po drugie – generacja kompletnego kodu dla wszystkich warstw aplikacji. Obecne rozwiązanie pozwala generować kod w warstwie widoku oraz warstwie logiki aplikacji. Powstały wprawdzie rozszerzenia dla języka RSL umożliwiające generowanie kodu logiki biznesowej [53] oraz kodu realizującego mapowanie obiektowo-relacyjne [169], które należałoby rozważyć i zintegrować w jedno spójne rozwiązanie w toku dalszych prac rozwojowych przedstawionego rozwiązania.

Po trzecie – rozszerzenie języka RSL oraz jego edytora, zwiększające możliwości tworzenia bardziej rozbudowanych graficznych interfejsów użytkownika. Obecne rozwiązanie, przygotowane w ramach niniejszej pracy, dostarcza ograniczone możliwości w tym zakresie – dostępny jest widok prostego formularza oraz listy a rozmieszczenie poszczególnych elementów interfejsu użytkownika w ramach tych widoków jest z góry narzucone. Nie ma też możliwości definiowania reguł wstępnej walidacji danych wprowadzanych z poziomu interfejsu użytkownika aplikacji. Dalszy rozwój rozwiązania powinien uwzględniać rozbudowę konstrukcji języka RSL oraz semantyki translacyjnej w tym zakresie. Wskazana byłaby też rozbudowa edytora języka RSL dostępnego w narzędziu ReDSeeDS o możliwość graficznego prototypowania interfejsu użytkownika.

Po czwarte – możliwość definiowania uprawnień użytkowników do wykonywania poszczególnych funkcji systemu i dostępu do danych. Semantyka translacyjna języka RSL (być może również składnia) powinna być rozszerzona w taki sposób, aby uwzględnić w wygenerowanym kodzie określone uprawnienia dla użytkowników aplikacji. Dodanie generycznego mechanizmu ról i uprawnień do generowanego kodu jest kwestią złożoności incydentalnej i wymagała rozbudowy samej transformacji w części generującej kod statycznie. Inicjalna konfiguracje tego mechanizmu powinna, natomiast, wynikać z modelu wymagań w języku RSL, np. w oparciu o hierarchię aktorów oraz relacje pomiędzy aktorami a przypadkami użycia.

[36] Web Content Accessibility Guidelines (https://www.w3.org/WAI/standards-guidelines/wcag/)

Powyżej wskazane kierunki nie wyczerpują oczywiście wszystkich możliwości dalszego rozwoju opisanego rozwiązania. Zebrane w trakcie realizacji pracy doświadczenia i obserwacje pozwalają zakładać, że dalsze, szerzej zakrojone badania i prace rozwojowe mogą zaowocować stworzeniem kompleksowego rozwiązania typu „No-Code", pozwalającego przenieść całkowicie ciężar tworzenia oprogramowania na poziom modeli wymagań. Efektem tego byłaby istotna redukcja złożoność incydentalnej związaną z wykorzystaniem tradycyjnych języków programowania oraz platform technologicznych i – tym samym – znaczący wzrost efektywności procesu wytwarzania oprogramowania.

Bibliografia

[1] I. F. Alexander i R. Stevens, Writing Better Requirements, Addison Wesley Professional, 2002.

[2] J. Beatty i A. Chen, Visual Models for Software Requirements: An RML Handbook, Microsoft Press, 2012.

[3] M. Chemuturi, Requirements Engineering and Management for Software Development Projects, Springer, 2012.

[4] G. Kotonya i I. Sommerville, Requirements engineering: processes and techniques, Wiley, 1998.

[5] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques, Springer, 2010.

[6] S. Robertson i J. Robertson, Mastering the Requirements Process: Getting Requirements Right, Addison Wesley, 2012.

[7] A. Sutcliffe, User-Centred Requirements Engineering, Springer, 2002.

[8] A. van Lamsweerde, Requirements Engineering: From System Goals to UML Models to Software Specifications, Wiley, 2009.

[9] K. Wiegers i J. Beatty, Software Requirements, 3 red., Microsoft Press, 2013.

[10] D. Leffingwell i D. Widrig, Managing Software Requirements: A Unified Approach, Addison-Wesley, 2000.

[11] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley Professional, 2009.

[12] D. Leffingwell, Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise, Pearson Education, 2010.

[13] K. Schwaber i M. Beedle, Agile Software Development with Scrum, 1st red., USA: Prentice Hall PTR, 2001.

[14] B. H. C. Cheng i J. M. Atlee, „Research Directions in Requirements Engineering," w *Future of Software Engineering, 2007. FOSE '07*, 2007.

[15] M. Kassab, C. Neill i P. Laplante, „State of practice in requirements engineering: contemporary data," *Innovations in Systems and Software Engineering*, tom 10, p. 235–241, April 2014.

[16] S. J. Mellor, K. Scott, A. Uhl i D. Weise, MDA Distilled: Principles of Model Driven Architecture, Addison-Wesley, 2004.

[17] S. Mellor, K. Scott, A. Uhl and D. Weise, "Model-Driven Architecture," in *Bruel, JM., Bellahsene, Z. (eds) Advances in Object-Oriented Information Systems. OOIS 2002. Lecture Notes in Computer Science, vol 2426.*, 2002.

[18] M. Guttman i J. Parodi, Real-Life MDA: Solving Business Problems with Model Driven Architecture, Morgan Kaufman, 2006.

[19] A. G. Kleppe, J. B. Warmer i B. Wim, MDA Explained, The Model Driven Architecture: Practice and Promise, Boston: Addison-Wesley, 2003.

[20] O. Pastor i J. C. Molina, Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling, Springer, 2007.

[21] D. Thomas, „MDA: Revenge of the Modelers or UML Utopia?," *IEEE Software*, tom 21, p. 22–24, 2004.

[22] M. Völter, T. Stahl, J. Bettin, A. Haase i S. Helsen, Model-Driven Software Development: Technology, Engineering, Management, Wiley, 2013.

[23] S. Beydeda, M. Book i V. Gruhn, Redaktorzy, Model-Driven Software Development, Springer, 2005.

[24] M. Brambilla, J. Cabot i M. Wimmer, Model-driven Software Engineering in Practice, Morgan & Claypool, 2012.

[25] D. Gasevic, D. Djuric i V. Devedzic, Model Driven Engineering and Ontology Development (2. ed.), Springer, 2009.

[26] V. G. Díaz, J. M. C. Lovelle, B. C. P. García-Bustelo i O. S. Martinez, Redaktorzy, Advances and Applications in Model-Driven Engineering, IGI Global, 2014.

[27] S. Cook, G. Jones, S. Kent i A. C. Wills, Domain-Specific Development with Visual Studio DSL Tools, Addison Wesley Professional, 2007.

[28] M. Fowler, Domain-Specific Languages, Pearson Education, 2010.

[29] S. Kelly i J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008.

[30] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1 red., Addison-Wesley Professional, 2008.

[31] I. Zikra, J. Stirna and J. Zdravkovic, "Analyzing the Integration between Requirements and Models in Model Driven Development," in *Enterprise, Business-Process and Information Systems Modeling*, vol. 81, T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt and I. Bider, Eds., Springer, 2011, pp. 342-356.

[32] R. Hirschfeld, M. Perscheid i M. Haupt, „Explicit use-case representation in object-oriented programming languages," w *Proceedings of the 7th symposium on Dynamic languages*, Portland, 2011.

[33] B. A. Berenbach, „Comparison of UML and Text Based Requirements Engineering," w *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2004.

[34] K. S. Mukasa, A. Jedlitschka, C. Graf, K. Klöckner, M. Eisenbarth i S. Steinbach-Nordmann, „Requirements Specification Language Validation Report," 2007.

[35] A. Jedlitschka, K. S. Mukasa i S. Weber, „Case Creation Verification and Validation," 2009.

[36] H. Kaindl, M. Smialek, P. Wagner, D. Svetinovic, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, H. Schwarz, D. Bildhauer, J. P. Brogan, K. S. Mukasa, K. Wolter i T. Krebs, „Requirements Specification Language Definition," 2009.

[37] M. Śmiałek, J. Bojarski, W. Nowakowski, A. Ambroziewicz i T. Straszak, „Complementary Use Case Scenario Representations Based on Domain Vocabularies," *Lecture Notes in Computer Science*, tom 4735, p. 544–558, 2007.

[38] M. Śmiałek, J. Bojarski, W. Nowakowski i T. Straszak, „Scenario Construction Tool based on Extended UML Metamodel," *Lecture Notes in Computer Science*, tom 3713, p. 414–429, 2005.

[39] A. Kalnins, A. Sostaks, E. Celms, E. Kalnina, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, V. Riediger, H. Schwarz, D. Bildhauer, S. Kavaldjian, R. Popp i F. Juergen, „Final Reuse-Oriented Modeling and Transformation Language Definition," 2009.

[40] A. Kalnins, E. Kalnina, E. Celms, A. Sostaks, H. Schwarz, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, S. Kavaldjian i F. Juergen, „Reusable Case Transformation Rule Specification," 2007.

[41] T. Krebs, W. Nowakowski, A. Kalnins i E. Kalnina, „Modeling and Transformation Language Validation Report," 2007.

[42] A. Ambroziewicz, T. Straszak, H. Schwarz, W. Nowakowski, J. P. Brogan, E. Kalnina, E. Celms, A. Sostaks, D. Bildhauer, M. Nick, S. Schneickert, A. Kalnins, J. Bojarski i L. Hotz, „Research-Oriented Software Artifact," 2007.

[43] M. Rein, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, A. Kalnins, E. Celms, E. Kalnina, D. Bildhauer, T. Szymasński, K. S. Mukasa i O. Unalan, „Final ReDSeeDS Prototype," 2009.

[44] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski i T. Straszak, „Methodology and Tool for Systematic Software Development with Requirements-Based Cases," *Electrical Review,* tom 86, nr 9, pp. 216-220, 2010.

[45] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, K. Wolter, L. Hotz, K. S. Mukasa, A. Jedlitschka, D. Bildhauer, K. Falkowski, J. Haas, T. Horn, V. Riediger, H. Schwarz, A. Kalnins, E. Kalnina, A. Sostaks, E. Celms, M. Rein, S. Drejewicz, S. Knab, J. Falb, O. Tufekci i I. Cokkececi, „Case-Driven Software Development. Comprehensive Approach to Produce and Reuse Model-Based Software Cases," 2009.

[46] P. Mohagheghi, F. Barbier, A. Berre, B. Morin, A. Sadovykh, T. Saether, A. Henry, A. Abherve, T. Ritter, C. Heim i M. Smialek, „Migrating Legacy Applications to the Service Cloud Paradigm: The REMICS Project," w *European Research Activities in Cloud Computing*, Cambridge Scholars Publishing, 2012, p. 97–122.

[47] W. Nowakowski, M. Smialek, A. Ambroziewicz, N. Jarzebowski i T. Straszak, „Recovery and Migration of Application Logic from Legacy Systems," *Computer Science,* tom 13, p. 53–70, 2012.

[48] W. Nowakowski, M. Śmiałek, A. Ambroziewicz i T. Straszak, „Requirements-Level Language and Tools for Capturing Software System Essence," *Computer Science and Information Systems,* tom 10, p. 1499–1524, 2013.

[49] M. Smialek, N. Jarzebowski i W. Nowakowski, „Translation of Use Case Scenarios to Java Code," *Computer Science,* tom 13, p. 35–52, 2012.

[50] M. Śmiałek, N. Jarzebowski i W. Nowakowski, „Runtime semantics of use case stories," w *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, 2012.

[51] M. Smialek, W. Nowakowski, N. Jarzebowski i A. Ambroziewicz, „From Use Cases and Their Relationships to Code," w *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012*, 2012.

[52] M. Śmiałek i W. Nowakowski, From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice, Springer International Publishing, 2015.

[53] K. Rybiński, *Reprezentacja wiedzy dziedzinowej na potrzeby generacji kodu z poziomu wymagań oprogramowania,* Oficyna Wydawnicza Politechniki Warszawskiej, 2019.

[54] A. R. S. Correia, J. M. Iyoda i C. T. L. L. Silva, „From Requirements to Ready to Run Software: A Brief Thought on How to Mechanize The Software Development Process," *International Journal in Foundations of Computer Science & Technology,* tom 4, p. 17–26, 2014.

[55] M. Śmiałek, „From User Stories to Code in One Day?," *Lecture Notes in Computer Science,* tom 3556, p. 38–47, 2005.

[56] V. Šimko, P. Hnětynka i T. Bureš, „From Textual Use-Cases to Component-Based Applications," *Studies in Computational Intelligence,* tom 295, p. 23–37, 2010.

[57] A. V. Aho, Compilers: Principles, Techniques, & Tools, Pearson/Addison Wesley, 2007.

[58] F. P. Brooks, „No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, tom 20, p. 10–19, April 1987.

[59] F. P. Brooks, The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, Pearson Education, 1995.

[60] J. Carter i W. B. Gardner, „Converting scenarios to CSP traces with Mise en Scene for requirements-based programming," *Innovations in Systems and Software Engineering*, tom 4, p. 45–70, 01 April 2008.

[61] M. G. Hinchey i J. L. Rash, „A Formal Approach to Requirements-Based Programming," w *Proc. 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, 2005.

[62] M. Smialek, „Requirements-Level Programming for Rapid Software Evolution," w *Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, J. Barzdins i M. Kirikova, Redaktorzy, IOS Press, 2011, pp. 37-51.

[63] J. Bézivin, „On the unification power of models," *Software {\&}amp$\mathsemicolon$ Systems Modeling*, tom 4, p. 171–188, May 2005.

[64] D. Djuric, D. Gaševi i V. Devedžic, „The Tao of Modeling Spaces.," *The Journal of Object Technology*, tom 5, p. 125, 2006.

[65] E. Seidewitz, „What models mean," *IEEE Software*, tom 20, pp. 26-32, September 2003.

[66] M. Śmiałek, „Accommodating Informality with Necessary Precision in Use Case Scenarios," *Journal of Object Technology*, tom 4, p. 59–67, August 2005.

[67] J. Sanchez Cuadrado, E. Guerra and J. de Lara, "Generic model transformations: Write Once, Reuse Everywhere," in *Theory and Practice of Model Transformations: 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011*.

[68] E. Guerra, J. de Lara, D. Kolovos, R. Paige and O. dos Santos, "transML: A Family of Languages to Model Model Transformations," in *Model Driven Engineering Languages and Systems. MODELS 2010. Lecture Notes in Computer Science, vol 6394.*, 2010.

[69] T. Horn and J. Ebert, "The GReTL transformation language," in *Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 6707*, 2011.

[70] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Satellite Events at the MoDELS 2005 Conference*, vol. 3844, J. Bruel, Ed., Springer, 2006, p. 128–138.

[71] A. Agrawal, G. Karsai and F. Shi, "Graph transformations on domain-specific models," *Journal on Software and Systems Modeling*, vol. 37, no. 11, 2003.

[72] R. Geiß, G. V. Batz, D. Grund, S. Hack i A. Szalkowski, „GrGen: A fast SPO-based graph rewriting tool," w *Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings 3*, 2006.

[73] D. Kolovos, R. Paige and F. Polack, "The Epsilon transformation language," in *Vallecillo, A., Gray, J., Pierantonio, A. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 5063, pp. 46–60*, 2008.

[74] „MOF 2.0 Query / View / Transformation Specification, v. 1.1, formal/2011-01-01," 2011.

[75] A. Kalnins, J. Barzdins i E. Celms, „Model Transformation Language MOLA," *Lecture Notes in Computer Science*, tom 3599, p. 62–76, 2004.

[76] E. Rencis, "Model transformation languages L1, L2, L3 and their implementation," *Computer Science and Information Technologies 733, Scientific Papers, University of Latvia,* 2008.

[77] J. Ebert i T. Horn, „GReTL: An Extensible, Operational, Graph-based Transformation Language," *Software and Systems Modeling,* tom 13, p. 301–321, February 2014.

[78] G. Booch, J. Rumbaugh i I. Jacobson, Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series), Addison-Wesley Professional, 2005.

[79] „OMG Unified Modeling Language, version 2.5.1, formal/17-12-05," 2017.

[80] P. Swithinbank, M. Chessell, T. Gardner, C. Griffin, J. Man, H. Wylie and L. Yusuf, Patterns: Model-Driven Development Using IBM Rational Software Architect, IBM Redbooks, 2005.

[81] D. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, USA: John Wiley & Sons, Inc., 2002.

[82] J. Miller i J. Mukerji, Redaktorzy, MDA Guide Version 1.0.1, omg/03-06-01, Object Management Group, 2003.

[83] H. Hussmann, G. Meixner i D. Zuehlke, Model-Driven Development of Advanced User Interfaces, Springer, 2011.

[84] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modeling, Addison-Wesley Object Technology Series, 1998.

[85] O. Lopez, F. Hayes and S. Bear, "Oasis: An object-oriented specification language," in *Loucopoulos, P. (eds) Advanced Information Systems Engineering. CAiSE 1992. Lecture Notes in Computer Science, vol 593*, 1992.

[86] M. Völter and J. Bettin, "Patterns for Model-Driven Software-Development," in *Proceedings of the 9th European Conference on Pattern Languages of Programms (EuroPLoP '2004), Irsee, Germany, July 7-11, 2004*.

[87] K. Czarnecki i S. Helsen, „Feature-based survey of model transformation approaches," *IBM Systems Journal,* tom 45, p. 621–645, 2006.

[88] P. Mohagheghi, W. Gilani, A. Stefanescu and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering,* vol. 18, p. 89–116, 2013.

[89] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen and M. Fritzsche, "Where does model-driven engineering help? Experiences from three industrial cases," *Software & Systems Modeling,* vol. 12, p. 619–639, 2011.

[90] A. S. Koch, Agile Software Development, Artech House Publishers, 2004, p. 280.

[91] O. Dieste, N. Juristo i F. Shull, „Understanding the Customer: What Do We Know about Requirements Elicitation?," *IEEE Software,* tom 25, p. 11–13, March 2008.

[92] T. Dyba i T. Dingsoyr, „What Do We Know about Agile Software Development?," *IEEE Software,* tom 26, p. 6–9, September 2009.

[93] P. Flinn, „Improving Product Development Performance," w *Managing Technology and Product Development Programmes,* John Wiley & Sons Ltd, 2019, p. 219–233.

[94] S. W. Ambler, Agile Modeling: Effective Practices for Extreme Programming and the Unified Process, USA: John Wiley & Sons, Inc., 2002.

[95] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan i V. Leppänen, „Technical debt and agile software development practices and processes: An industry practitioner survey," *Information and Software Technology,* tom 96, pp. 141-160, 2018.

[96] P. S. M. dos Santos, A. Varella, C. R. Dantas i D. B. Borges, „Visualizing and Managing Technical Debt in Agile Development: An Experience Report," w *Lecture Notes in Business Information Processing*, Springer Berlin Heidelberg, 2013, p. 121–134.

[97] J. Greenfield i K. Short, Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools, Indianapolis, Indiana: Wiley, 2004.

[98] J. S. Cuadrado i J. G. Molina, „A Model-Based Approach to Families of Embedded Domain-Specific Languages," *IEEE Transactions on Software Engineering*, tom 35, p. 825–840, 2009.

[99] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison Wesley, 2004.

[100] K. Pohl, G. Böckle i F. J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.

[101] J.-C. Royer i H. Arboleda, Model-Driven and Software Product Line Engineering, Wiley, 2013.

[102] F. J. van der Linden, K. Schmid i E. Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, 2007.

[103] B. Berenbach, „A 25 year retrospective on model-driven requirements engineering," w *Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE*, 2012.

[104] A. Moreira, G. Mussbacher, J. Araújo, N. Bencomo i P. Sánchez, Redaktorzy, International Workshop on Model-Driven Requirements Engineering, MoDRE 2013, Rio de Janeiro, Brasil: IEEE Computer Society, 2013.

[105] A. Moreira, G. Mussbacher, J. Araujo i P. Sanchez, Redaktorzy, First Model-Driven Requirements Engineering Workshop, MoDRE 2011, Trento, Italy: IEEE Computer Society, 2011.

[106] G. Mussbacher, J. Araujo i P. Sanchez, Redaktorzy, Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012, Chicago, IL, USA: IEEE Computer Society, 2012.

[107] S. Assar, „Model Driven Requirements Engineering: Mapping the field and beyond," w *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, 2014.

[108] M. Śmiałek, A. Kalnins, A. Ambroziewicz, T. Straszak i K. Wolter, „Comprehensive System for Systematic Case-Driven Software Reuse," *Lecture Notes in Computer Science*, tom 5901, p. 697–708, 2010.

[109] M. Smialek i T. Straszak, „Facilitating transition from requirements to code with the ReDSeeDS tool," w *Requirements Engineering Conference (RE), 2012 20th IEEE International*, 2012.

[110] M. Śmiałek, Software Development with Reusable Requirements-Based Cases, Publishing House of the Warsaw University of Technology, 2007.

[111] University of Latvia, „The MOLA Language, Reference Manual, Version 2.0 final," Riga, 2007.

[112] D. Rubel, "The Heart of Eclipse: A look inside an extensible plug-in architecture," *Queue*, vol. 4, no. 8, pp. 36-44, October 2006.

[113] J. McAffer and J.-M. Lemieux, Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications, Addison-Wesley, 2006.

[114] B. Moore, Eclipse Development: Using the Graphical Editing Framework and the Eclipse Modeling Framework, IBM, 2004.

[115] J. Ebert, V. Riediger, H. Schwarz i B. Daniel, „Using the TGraph Approach for Model Fact Repositories," w *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008)*, 2008.

[116] H. Kaindl, M. Smialek i W. Nowakowski, „Case-based Reuse with Partial Requirements Specifications," w *Requirements Engineering Conference (RE), 2010 18th IEEE International*, 2010.

[117] A. Ambroziewicz i M. Smialek, „Application Logic Patterns – Reusable Elements of User-System Interaction," *Lecture Notes in Computer Science*, tom 6394, p. 241–255, 2010.

[118] A. Ambroziewicz, *Application logic patterns - description language, implementation and usage*, Wojskowa Akademia Techniczna, 2016.

[119] V. Riediger, D. Bildhauer, H. Schwarz, A. Kalnins, A. Sostaks, E. Celms, M. Śmiałek i A. Ambroziewicz, „Software Case Meta-Model Definition," 2007.

[120] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski i T. Straszak, „Introducing a unified Requirements Specification Language," w *Proc. CEE-SET'2007, Software Engineering in Progress*, 2007.

[121] O. M. Group, „Meta Object Facility (MOF) Core Specification, version 2.5.1, formal/2016-11-01," 2016.

[122] „IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1-40, 1998.

[123] M. Ridao, J. Doorn and J. C. S. d. P. Leite, "Domain Independent Regularities in Scenarios," *Requirements Engineering*, 2001.

[124] J. C. S. d. P. Leite, G. D. S. Hadad, J. H. Doorn i G. N. Kaplan, „A Scenario Construction Process," w *Proceedings of the RE '00*, 2000.

[125] D. Bjorner, "Role of Domain Engineering in Software Development - Why Current Requirements Engineering is Flawed!," in *Lecture Notes in Computer Science, vol. 5947, pp. 2-34*, 2010.

[126] M. Fowler, Analysis patterns: reusable objects models, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[127] ISO/IEC 25010, *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, 2011.

[128] I. Jacobson, M. Christerson, P. Jonsson i G. Övergaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Reading: Addison-Wesley, 1992.

[129] S. Adolph, P. Bramble, A. Cockburn i A. Pols, Patterns for Effective Use Cases, Addison Wesley, 2002.

[130] H. Astudillo, G. Génova, M. Śmiałek, J. Llorens Morillo, P. Metz i R. Prieto-Diáz, „Use Cases in Model-Driven Software Engineering," *Lecture Notes in Computer Science*, tom 3844, p. 262–271, 2006.

[131] A. Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000.

[132] M. El-Attar i J. Miller, „Constructing high quality use case models: a systematic review of current practices," *Requirements Engineering*, pp. 1-15, 2011.

[133] D. Kulak i E. Guiney, Use Cases: Requirements in Context, 2 red., Addison Wesley, 2012.

[134] D. Rosenberg i K. Scott, Use Case Driven Object Modeling with UML: Theory and Practice, Apress, 2008.

[135] G. Schneider i J. P. Winters, Applying Use Cases: A Practical Guide, Second Edition, Addison-Wesley, 2001.

[136] G. Génova, J. Llorens i V. Quintana, „Digging into use case relationships," w *International Conference on the Unified Modeling Language*, 2002.

[137] M. A. Laguna, J. M. Marqués i Y. Crespo, „On the Semantics of the Extend Relationship in Use Case Models: Open-Closed Principle or Clairvoyance?," *Lecture Notes in Computer Science*, tom 6051, p. 409–423, 2010.

[138] P. Metz, J. O'Brien i W. Weber, „Against Use Case Interleaving," *Lecture Notes in Computer Science*, tom 2185, p. 472–486, 2001.

[139] P. Metz, J. O'Brien i W. Weber, „Specifying Use Case Interaction: Types of Alternative Courses," *Journal of Object Technology*, tom 2, p. 111–131, March 2003.

[140] P. Metz, J. O'Brien i W. Weber, „Specifying Use Case Interaction: Clarifying Extension Points and Rejoin Points," *Journal of Object Technology*, tom 3, p. 87–102, May 2004.

[141] A. J. H. Simons, „Use cases considered harmful," w *Proceedings of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe'99*, Nancy, 1999.

[142] A. Fatwanto, „Specifying translatable software requirements using constrained natural language," w *2012 7th International Conference on Computer Science Education (ICCSE)*, 2012.

[143] S. Boyd, D. Zowghi i A. Farroukh, „Measuring the expressiveness of a constrained natural language: an empirical study," w *13th IEEE International Conference on Requirements Engineering*, 2005.

[144] N. Chomsky, „Three models for the description of language," *IRE Transactions on Information Theory*, tom 2, pp. 113-124, 1956.

[145] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger i P. Naur, „Report on the Algorithmic Language ALGOL 60," *Commun. ACM*, tom 3, p. 299–314, May 1960.

[146] *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*, 1996.

[147] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, pp. 529-565, 2010.

[148] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, November 1995.

[149] K. Lano, UML 2 Semantics and Applications, K. Lano, Red., Wiley, 2009.

[150] K. Slonneger i B. L. Kurtz, Formal Syntax and Semantics of Programming Languages, Addison-Wesley, 1995.

[151] Y. Zhang i B. Xu, „A Survey of Semantic Description Frameworks for Programming Languages," *SIGPLAN Not.*, tom 39, nr 3, 2004.

[152] D. S. Scott, „Outline of a Mathematical Theory of Computation," Oxford University Computing Laboratory, Oxford, 1970.

[153] D. Scott i C. Strachey, „TOWARD A MATHEMATICAL SEMANTICS FOR COMPUTER LANGUAGES," 1971.

[154] J. van Wijngaarden i E. Visser, „Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems,"

Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[155] J. Evermann i Y. Wand, „Toward formalizing domain modeling semantics in language syntax," *IEEE Transactions on Software Engineering,* tom 31, p. 21–37, January 2005.

[156] J. Gosling, B. Joy, G. L. Steele, G. Bracha i A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st red., Addison-Wesley Professional, 2014.

[157] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software, Volume 1: A System of Patterns, Chichester, UK: Wiley, 1996.

[158] M. Shaw, "Heterogeneous design idioms for software architecture," in *Proceedings of the 6th international workshop on Software specification and design*, Como, Italy, 1991.

[159] T. Reenskaug, „Models-views-controllers," Xerox PARC, 1979.

[160] M. Potel, "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java & Taligent Inc," 1996.

[161] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002, p. 560.

[162] J. Martin, Managing the Data Base Environment, Savant Res. Inst., 1981.

[163] G. Fink i I. Flatow, „Introducing Single Page Applications," w *Pro Single Page Application Development: Using Backbone.js and ASP.NET*, Berkeley, CA: Apress, 2014, p. 3–13.

[164] Y. Sun, „Single-Page Applications," w *Practical Application Development with AppRun: Building Reliable, High-Performance Web Apps Using Elm-Inspired Architecture, Event Pub-Sub, and Components*, Berkeley, CA: Apress, 2019, p. 141–162.

[165] F. Gutierrez, Introducing Spring Framework: A Primer, 1st red., USA: Apress, 2014.

[166] A. Sarin i J. Sharma, Getting Started with Spring Framework: A Hands-On Guide to Begin Developing Applications Using Spring Framework, CreateSpace Independent Publishing Platform, 2016.

[167] C. Jones, „Programming Languages Table, Release 8.2," Software Productivity Research, Inc., 1997.

[168] M. Śmiałek, „Zrozumieć UML 2.0. Metody modelowania obiektowego," Helion, 2005.

[169] N. Yamouni Khelifi, M. Śmiałek i M. Rachida, „Generating database access code from domain models," w *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2015.

[170] A. Albrecht, „Measuring application development productivity," w *Proceedings IBM Applications Development*, Monterey, California, 1979.

[171] ISO/IEC20926, „2009 Software and Systems Engineering—Software Measurement—IFPUG Functional Size Measurement Method, 2nd Edition," ISO, Geneva, 2009.