

Inżynieria oprogramowania

Michał Śmiałek, Kamil Rybiński

Politechnika Warszawska, 2022

Spis treści

1. WPROWADZENIE DO INŻYNIERII OPROGRAMOWANIA	5
1.1. Co to jest inżynieria oprogramowania?	5
1.2. Podstawowe problemy inżynierii oprogramowania	7
1.3. Przyczyny problemów.....	10
1.4. Najlepsze praktyki inżynierii oprogramowania.....	12
1.5. Przykładowe dziedziny zastosowań inżynierii oprogramowania	14
ZADANIA	19
SŁOWNIK POJĘĆ.....	19
Co trzeba zapamiętać	20
Rozwiązania zadań	22
2. CYKLE WYTWARZANIA OPROGRAMOWANIA	23
2.1. Dyscypliny cyklu wytwarzania oprogramowania.....	23
2.2. Przegląd cykli wytwarzania oprogramowania.....	30
ZADANIA	36
SŁOWNIK POJĘĆ.....	36
Co trzeba zapamiętać	38
3. METODYKI WYTWARZANIA OPROGRAMOWANIA.....	39
3.1. Co to jest metodyka wytwarzania oprogramowania?	39
3.2. Metodyki zwinne (AGILE).....	41
3.3. Metodyki sformalizowane	49
ZADANIA	55
SŁOWNIK POJĘĆ.....	56
Co trzeba zapamiętać	56
4. WPROWADZENIE DO MODELOWANIA OBIEKTOWEGO.....	58
4.1. Podstawowe zasady modelowania.....	58
4.2. Uniwersalny język modelowania	61
4.3. Obiekty jako podstawa modelowania	62
4.4. Klasy obiektów	65
4.5. System jako zbiór współpracujących obiektów	68
4.6. Modele w procesie inżynierii oprogramowania	70
ZADANIA	71
SŁOWNIK POJĘĆ.....	72
Co trzeba zapamiętać	72
5. MODELOWANIE STRUKTURY SYSTEMU.....	74
5.1. Model klas.....	75
5.2. Model komponentów i model wdrożenia	83
ZADANIA	87
SŁOWNIK POJĘĆ.....	87
Co trzeba zapamiętać	90
Rozwiązania zadań	91

6. MODELOWANIE DYNAMIKI SYSTEMU	93
6.1. MODEL PRZYPADKÓW UŻYCIA	93
6.2. MODEL CZYNNOŚCI	99
6.3. MODEL MASZYNY STANÓW	103
6.4. MODEL SEKWENCJI	105
ZADANIA	111
SŁOWNIK POJĘĆ	112
Co TRZEBA ZAPAMIĘTAĆ	114
ROZWIĄZANIA ZADAŃ	116
7. WPROWADZENIE DO INŻYNIERII WYMAGAŃ	119
7.1. ROLA WYMAGAŃ W INŻYNIERII OPROGRAMOWANIA	119
7.2. SPECYFIKOWANIE ŚRODOWISKA SYSTEMU	123
7.3. STRUKTURA SPECYFIKACJI WYMAGAŃ – RODZAJE WYMAGAŃ	127
ZADANIA	132
SŁOWNIK POJĘĆ	133
Co TRZEBA ZAPAMIĘTAĆ	133
ROZWIĄZANIA ZADAŃ	135
8. PODSTAWY SPECYFIKOWANIA WYMAGAŃ.....	136
8.1. SPECYFIKOWANIE WIZJI SYSTEMU.....	136
8.2. SPECYFIKOWANIE WYMAGAŃ UŻYTKOWNIKA	140
8.3. SPECYFIKOWANIE WYMAGAŃ OPROGRAMOWANIA.....	148
ZADANIA	156
SŁOWNIK POJĘĆ	156
Co TRZEBA ZAPAMIĘTAĆ	157
ROZWIĄZANIA ZADAŃ	159
9. WPROWADZENIE DO ARCHITEKTURY OPROGRAMOWANIA.....	163
9.1. ROLA PROJEKTOWANIA ARCHITEKTONICZNEGO	163
9.2. ARCHITEKTURY KOMPONENTOWE I USŁUGOWE	165
9.3. TYPOWE STYLE ARCHITEKTONICZNE.....	170
9.4. PROJEKTOWANIE ARCHITEKTURY NA PODSTAWIE WYMAGAŃ	179
ZADANIA	185
SŁOWNIK POJĘĆ	186
Co TRZEBA ZAPAMIĘTAĆ	187
ROZWIĄZANIA ZADAŃ	189
10. PODSTAWY PROJEKTOWANIA PODSYSTEMÓW.....	192
10.1. PROJEKTOWANIE WARSTW PREZENTACJI I LOGIKI APLIKACJI	192
10.2. PROJEKTOWANIE WARSTWY LOGIKI DZIEDZINOWEJ	197
10.3. PROJEKTOWANIE BAZ DANYCH	201
ZADANIA	208
SŁOWNIK POJĘĆ	210
Co TRZEBA ZAPAMIĘTAĆ	210
ROZWIĄZANIA ZADAŃ	212
11. PODSTAWY IMPLEMENTACJI OPROGRAMOWANIA	214
11.1 KODOWANIE SYSTEMU NA PODSTAWIE PROJEKTU	214
11.2. DOBRE PRAKTYKI W ZAKRESIE KODOWANIA.....	218
11.3. ZARZĄDZANIE WERSJAMI, KONFIGURACJĄ I ZMIANAMI	222
ZADANIA	227
SŁOWNIK POJĘĆ	229

CO TRZEBIA ZAPAMIĘTAĆ	230
ROZWIĄZANIA ZADAŃ	231
12. PODSTAWY TESTOWANIA	234
12.1. ROLA TESTOWANIA W INŻYNIERII OPROGRAMOWANIA	234
12.2. PODSTAWOWE METODY TESTOWANIA	236
12.3. POZIOMY TESTOWANIA	240
12.4. TESTOWANIE PRZYPADKÓW UŻYCIA SYSTEMU.....	242
ZADANIA	244
SŁOWNIK POJĘĆ	244
CO TRZEBIA ZAPAMIĘTAĆ	246
13. NARZĘDZIA I METODY AUTOMATYZACJI INŻYNIERII OPROGRAMOWANIA	248
13.1. NARZĘDZIA AUTOMATYZACJI ANALIZY I PROJEKTOWANIA OPROGRAMOWANIA	248
13.2. NARZĘDZIA WSPARCIA IMPLEMENTACJI I TESTOWANIA OPROGRAMOWANIA.....	252
13.3. METODY AUTOMATYZACJI WYTWARZANIA I EKSPOLOATACJI	257
ZADANIA	262
SŁOWNIK POJĘĆ	263
CO TRZEBIA ZAPAMIĘTAĆ	263

1. Wprowadzenie do inżynierii oprogramowania

1.1. Co to jest inżynieria oprogramowania?

Czym zajmuje się inżynieria oprogramowania?

Termin „inżynieria oprogramowania” został użyty po raz pierwszy w 1967 roku. Wtedy to odbyła się pierwsza konferencja na ten temat zorganizowana przez NATO. Od tego czasu inżynieria oprogramowania przeszła znaczący rozwój. W niniejszym podręczniku przedstawimy podstawy tego obszaru informatyki technicznej.

Ogólnie można powiedzieć, że **inżynieria oprogramowania** zajmuje się inżynierskim podejściem do tworzenia oprogramowania. Inżynieria oprogramowania dąży do ujęcia działań związanych z budową programów komputerowych w ramy typowe dla innych dziedzin inżynierii. W szczególności, inżynieria oprogramowania stosuje wiedzę naukową, techniczną oraz doświadczenie w celu projektowania, implementacji, walidacji i dokumentowania oprogramowania.

Inżynierię oprogramowania można podzielić na kilka zasadniczych dyscyplin. Dyscypliny te związane są z typowymi etapami działań inżynierskich w dowolnej dziedzinie inżynierii. Oczywiście, dyscypliny inżynierii oprogramowania posiadają istotne cechy wyróżniające je spośród dyscyplin sformułowanych ogólnie. Dla ustalenia uwagi, spróbujemy porównać niektóre dyscypliny inżynierii oprogramowania z dyscyplinami inżynierii budowlanej.

Dyscyplina 1: Wymagania

Aby zbudować dom, przyszli jego właściciele lub mieszkańcy muszą określić swoje potrzeby. Te potrzeby należy sformułować, tak, aby wykonawcy domu byli w stanie jak najlepiej je spełnić. Potrzeby mogą dotyczyć funkcjonalności domu (układ funkcjonalny, liczba pokoi itp.), jak i jego cech niefunkcjonalnych (estetyka, kolor, itp.).

Podobnie, w przypadku budowy systemu oprogramowania, należy zebrać dokładne wymagania od jego przyszłych użytkowników. Zadanie to jest znacznie bardziej złożone od zebrania wymagań na dom, lub nawet złożony budynek wielofunkcyjny. Systemy informatyczne realizują coraz bardziej złożoną funkcjonalność dla coraz większej liczby grup użytkowników. Stąd też bardzo istotne jest uzyskanie od tychże użytkowników (jak również od innych grup zainteresowanych systemem) wszystkich niezbędnych informacji dotyczących ich potrzeb. Należy też uważać na to, że potrzeby użytkowników systemów oprogramowania są bardzo zmienne, co zdecydowanie odróżnia te potrzeby od potrzeb właścicieli domów.

Dyscyplina 2: Projektowanie

Na podstawie wymagań zebranych od klientów, architekt przystępuje do projektowania domu. Najpierw musi określić strukturę domu, w szczególności – z jakich dom będzie się składał pomieszczeń, ile będzie miał kondygnacji, jak będzie ukształtowany dach itd. Potem, ekipa inżynierów konstruktorów projektuje detale konstrukcyjne: typy stropów, rodzaj belek konstrukcyjnych, rodzaj elewacji itd.

System oprogramowania też wymaga projektu. Jest to o tyle istotne, że liczba elementów, z jakich skonstruowany jest typowej wielkości system oprogramowania znacznie przekracza liczbę elementów

konstrukcyjnych budynku. Niezbędne jest określenie przez architektów, z jakich składników będzie się składał system i na jakich maszynach (komputerach, procesorach) będzie on wykonywany. Bardzo istotne jest też pokazanie wykonawcom, w jaki sposób system będzie realizował swoją funkcjonalność. Podobnie jak w przypadku projektowania budynków, konieczne jest użycie „rysunków technicznych”. Dla systemów oprogramowania istnieje standardowy język graficznego opisu systemu, który zostanie przedstawiony w dalszych rozdziałach.

Dyscyplina 3: Implementacja

Plany architektoniczne i projekt konstrukcyjny budynku jest podstawą do rozpoczęcia budowy. Zgodnie z projektem należy wytyczyć budynek, zbudować fundamenty, wznieść mury, pokryć dachem i wykonać prace instalacyjno-wykończeniowe.

Wykonanie (implementacja) systemu oprogramowania również wymaga zachowania zgodności z projektem. System oprogramowania konstruowany jest poprzez pisanie programów w odpowiednich językach programowania. W zachowaniu zgodności z projektem mogą pomóc odpowiednie metody przekształcania projektu w kod systemu. Dla dobrze zaprojektowanego systemu, prace związane z implementacją (kodowaniem) sprowadzają się do uzupełnienia elementów kodu wygenerowanych na podstawie projektu szczegółowego. W niniejszym podręczniku nie będziemy omawiać zasad programowania i języków programowania. Podamy natomiast zasady zgodności kodu z projektem.

Dyscyplina 4: Walidacja

W trakcie i po zakończeniu budowy budynku przeprowadzane są niezbędne sprawdzenia (kontrola jakości). Geodeci sprawdzają, czy budynek został postawiony w prawidłowym miejscu. Służby instalacyjne sprawdzają prawidłowość zainstalowania sieci i urządzeń instalacyjnych. Wreszcie, sami mieszkańcy sprawdzają, czy zostały zrealizowane ich wymagania. W razie wykrycia usterek, należy dokonać niezbędnych poprawek. Przy tym, należy uważać, aby usterki wykryć w odpowiednim czasie. Jeżeli np. stwierdzimy, że przecieka sieć hydrauliczna po wykonaniu wszystkich tynków i podłóg, bardzo kosztowne stanie się znalezienie usterki (trzeba rozkuwać ściany i podłogi).

W przypadku systemów oprogramowania, wykrycie usterek jest bardzo złożonym problemem. Wynika to przede wszystkim ze złożoności i zmienności wymagań oraz złożoności systemów. Przede wszystkim należy sprawdzić, czy system został zbudowany zgodnie z potrzebami klienta i użytkowników. Można zauważać, że w zasadzie jedynym kryterium walidacji jest zgodność z tymi potrzebami. System dobrej jakości to system zgodny z nałożonymi na niego wymaganiami. W trakcie walidacji sprawdza się zatem, czy funkcjonalność systemu jest odpowiednia dla potrzeb klienta. Równocześnie sprawdza się inne elementy jakości, takie jak np. niezawodność (w tym odporność na awarie sprzętu) czy wydajność (szybkość działania). Należy podkreślić, że walidacja nie powinna być wykonywana dopiero po zakończeniu implementacji. Należy ją przeprowadzać jak najwcześniej, w celu zmniejszenia ryzyka niepowodzenia. Podobnie jak w przykładzie z naprawą instalacji, często zdarzają się sytuacje, kiedy niewykryte usterki systemu są bardzo kosztowne w naprawie, jeśli zostały wykryte zbyt późno.

Dyscyplina 5: Nadzór

Podczas projektowania i budowy domu, obowiązują pewne procedury związane ze sposobem wykonywania czynności, przepisami administracyjnymi i bezpieczeństwem pracy. Odpowiedni inspektorzy i kierownicy budowy dostosowują te procedury do warunków na budowie oraz kontrolują ich przestrzeganie.

W trakcie budowy systemu oprogramowania też bardzo istotne jest przestrzeganie pewnych procedur i reguł postępowania. W tym celu zostały opracowane odpowiednie standardowe metodyki. Odejście od przestrzegania (lub po prostu brak) metodyk jest bardzo częstą przyczyną niepowodzeń projektów, w które wkrada się chaos organizacyjny. Czynności zawarte w metodykach dotyczą wszystkich etapów

budowy systemu – od wymagań aż do walidacji i oddania systemu do użytku. Nad przestrzeganiem metodyki powinien czuwać odpowiedni kierownik (tzw. „metodyk”).

Dyscyplina 6: Środowisko pracy.

Podczas projektowania i budowy domu bardzo istotne jest środowisko pracy. Współcześni architekci używają odpowiednich systemów CAD (Computer Aided Design), które wyręczają ich w żmudnych pracach kreślarskich. Ekipy budowlane używają odpowiednich narzędzi, które również przyspieszają ich pracę.

Budując system oprogramowania również powinniśmy bardzo uważnie podejść do budowy środowiska pracy. Bardzo istotny jest wybór narzędzi wspomagających projektowanie systemu. Podczas budowy systemu, niezbędne jest posiadanie narzędzi umożliwiających zarządzanie złożonym kodem (dobre środowisko zintegrowane oraz narzędzia do zarządzania konfiguracją i zmianami). Dobrze dobrane środowisko pracy ułatwia również rozbudowę systemu (tzw. ewolucja systemu). W następnych rozdziałach opiszymy najczęściej używane narzędzia.

1.2. Podstawowe problemy inżynierii oprogramowania

Złożoność systemów oprogramowania

Typowe współczesne systemy oprogramowania składają się z setek tysięcy lub nawet milionów wierszy kodu w różnych językach programowania (np. rozmiar systemu operacyjnego Red Hat Linux 7.1 szacowany jest na 30 milionów wierszy kodu źródłowego). Przekłada się to na jeszcze większą liczbę poszczególnych instrukcji. Jest oczywiste, że takie liczby elementów nie jest w stanie opanować nawet najbardziej uzdolniony programista. Konieczne jest zatem zastosowanie metod opanowania tej złożoności.

Ta złożoność wynika z zakresu funkcjonalności, jakie współczesne systemy oprogramowania muszą zapewniać ich użytkownikom. Dla przykładu, typowy, zaawansowany procesor tekstu dostarcza dzisiaj takie opcje menu, wielu operacji sterowanych klawiszami funkcyjnymi, wielu formularzy i okienek do wprowadzania danych. Taki procesor jest składnikiem większego pakietu aplikacji biurowych (m.in. arkusz kalkulacyjny, narzędzie prezentacyjne). Innym, powszechnie znanym przykładem jest system dostępu do usług bankowych za pośrednictwem Internetu. Zauważmy, jak wiele opcji zawiera taki system, ile różnych operacji można wykonać (np. dokonać przelewu, zdefiniować odbiorców, wykonać zestawienie transakcji, aktywować kartę bankową, zmienić kod PIN, itd. itd.).

Systemy oprogramowania są spotykane praktycznie we wszystkich współczesnych systemach ułatwiających nam życie. Dotyczy to zarówno systemów, do których mamy dostęp za pośrednictwem komputerów osobistych, jak i systemów nadzorujących pracę różnego rodzaju urządzeń (od maszynki do golenia, poprzez samolot pasażerski, aż do statku kosmicznego). Z tego punktu widzenia możemy zatem podzielić systemy oprogramowania na:

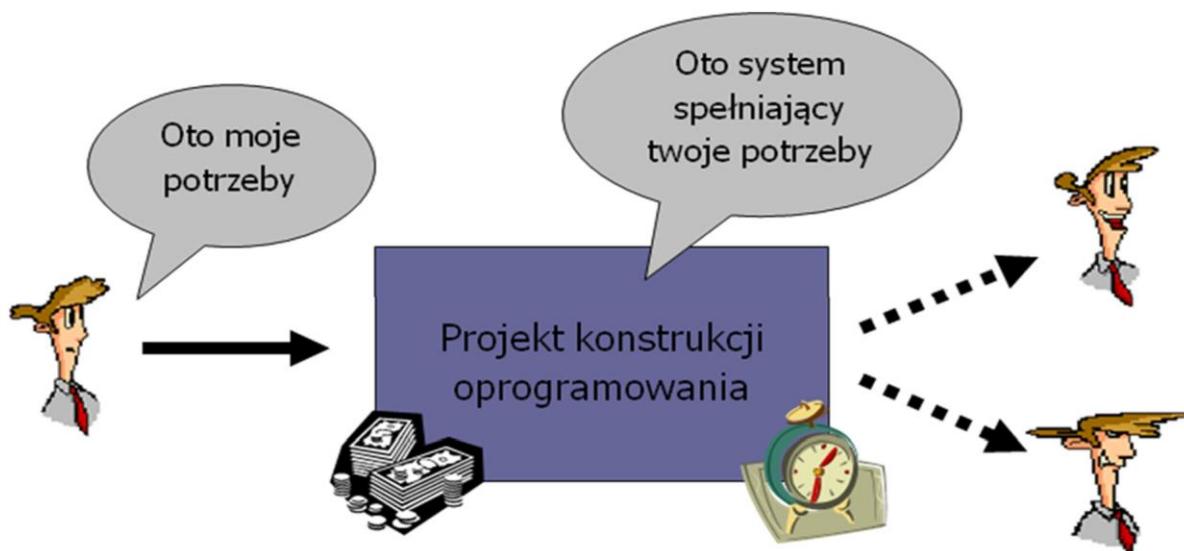
1. **Oprogramowanie aplikacyjne.** Są to systemy działające na typowych komputerach, tabletach czy smartfonach, wspierające różnego rodzaju czynności wykonywane w domu i w pracy. W szczególności mogą to być systemy wspierające pracę organizacji biznesowych (od małych firm handlowych aż do dużych korporacji międzynarodowych). Systemy te działają w różnych dziedzinach gospodarki: przemyśle wytwórczym, handlu, telekomunikacji, usługach finansowych (np. bankowość, ubezpieczenia). Mogą to być również systemy do pracy indywidualnej lub do zastosowań w zakresie rozrywki.
2. **Oprogramowanie wbudowane** (czasu rzeczywistego). Są to systemy oprogramowania kontrolujące pracę urządzeń mechanicznych i/lub elektronicznych. Działają one w czasie

rzeczywistym, tzn. reagują na sygnały i wymagają czasu reakcji dostosowanego do ograniczeń narzuconych przez dane urządzenie. Na przykład, system kontrolujący tor lotu rakiety wymaga korekty lotu tej rakiety na podstawie obliczeń. Korekta musi być wykonana w odpowiednim czasie, gdyż w przeciwnym razie rakieta może ulec uszkodzeniu. Innym przykładem może być system sterowania telewizorem, który wymaga reakcji na odpowiednie polecenia wydawane przez widza.

Problemy w projektach konstrukcji oprogramowania

Zaczyna się kolejne spotkanie komitetu sterującego projektu. Członkowie komitetu z ramienia odbiorcy zgłaszą szereg zastrzeżeń dotyczących jakości dostarczanego systemu oprogramowania oraz niedotrzymywania terminów. Członkowie komitetu z ramienia dostawcy narzekają na kolejną porcję zmian zgłoszonych przez odbiorcę. Projekt zbliża się do końca, a testy akceptacyjne wykazują kompletny brak zgodności produktu z wymaganiami. Projektanci i programiści toczą heroiczne boje o dotrzymanie jakiegokolwiek terminu i zrealizowanie choć części wymagań odbiorcy.

Jest to niestety, dosyć typowy scenariusz. Złożoność współczesnych systemów oprogramowania prowadzi do wielu niepowodzeń w budowie tychże systemów. Z punktu widzenia uczestników projektu budowy oprogramowania można wyróżnić trzy kryteria warunkujące powodzenie, co zostało zilustrowane na rysunku 1.1.



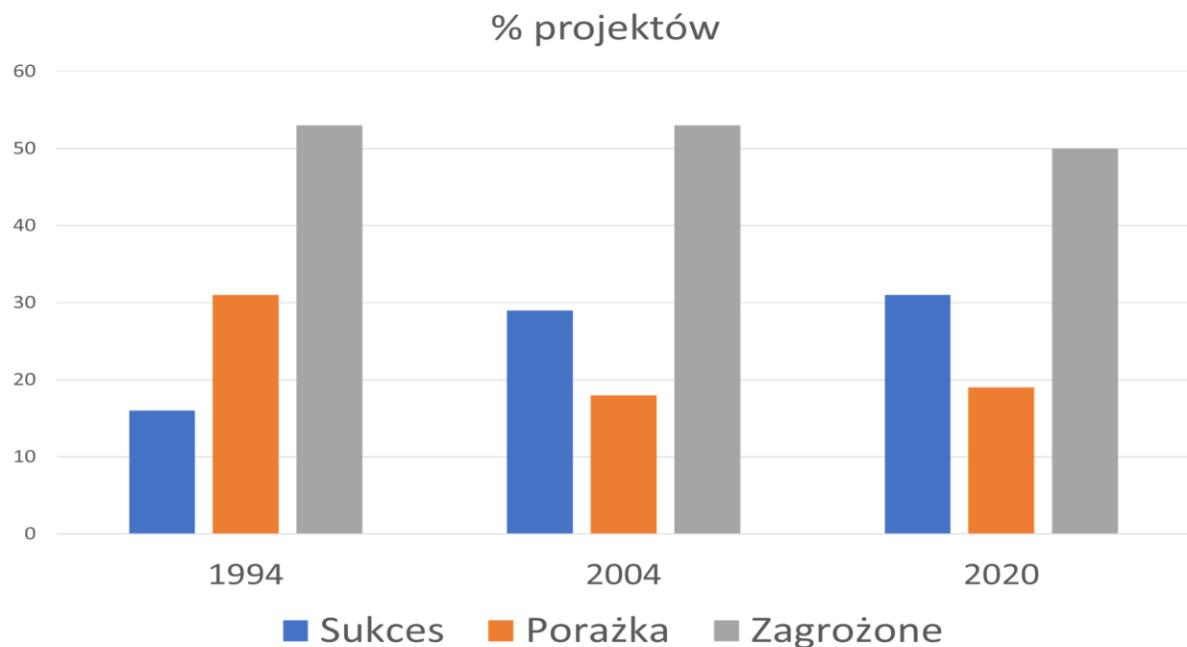
Rysunek 1.1: Kryteria sukcesu projektu konstrukcji oprogramowania

Kryteria te najczęściej określane są na etapie zawierania umowy na budowę systemu. W takiej umowie określa się wymagania odnośnie systemu, termin oddania systemu do użytku i koszt jego wykonania. Zatem można uznać, że projekt skończył się sukcesem, jeżeli:

- Projekt mieści się w budżecie,
- Projekt mieści się w ramach czasowych (termin zakończenia nie jest przekroczony),
- Dostarczony system spełnia rzeczywiste potrzeby klienta (zadowolony klient).

Niestety, bardzo trudno jest uzyskać spełnienie tych trzech kryteriów. Można tutaj przytoczyć statystyki opracowywane co roku przez grupę Standisha w tzw. „raporcie chaosu” (*ang. chaos report*). Rysunek 1.2 przedstawia wyniki dla lat 1994, 2004 i 2020. Statystyki te pokazują bardzo niekorzystną sytuację w przemyśle wytwarzania oprogramowania. Oznaczają one, że znaczący odsetek projektów (71% w roku 2004, 69% w roku 2020) kończy się niepowodzeniem (porażką, lub poważnymi

problemami z dostarczeniem systemu). Oznacza to olbrzymie straty szacowane na miliardy dolarów/euro/złotych rocznie.



Rysunek 1.2: Statystyka rezultatów projektów konstrukcji oprogramowania

Dodatkowo, warto przytoczyć następujące wyniki z roku 2004:

- 29% - odsetek projektów konstrukcji oprogramowania zakończonych sukcesem
- 82% - średnie przekroczenie budżetu projektu w stosunku do planu
- 52% - odsetek spełnienia wymagań

Można zatem powtórzyć za podstawową tezą pierwszej konferencji NATO nt. inżynierii oprogramowania: „jest kryzys”. Ponieważ jednak stan ten utrzymuje się już od kilkudziesięciu lat, należałoby raczej powtórzyć za Rogerem Pressmannem: „to chroniczna choroba”. Choroba ta ma charakterystyczne symptomy, po których można łatwo poznać, czy również nasz projekt jest nią „zarażony”.

Poniżej przedstawimy kilka przykładowych objawów.

1. Niezadowoleni klienci
 - a. „nie taki system mieliśmy na myśli”
 - b. „żaden z naszych urzędników nie będzie w stanie się tego nauczyć”
2. Niezadowoleni dostawcy
 - a. „to czego oni tak naprawdę chcą?”
 - b. „dlaczego ciągle zmieniają zdanie”
 - c. „tak się naprawialiśmy i co?”
3. Kłótnie o zakres
 - a. „chcecie, żebyśmy zbudowali system dwa razy większy niż zapisano w umowie”
 - b. „ale ta funkcjonalność miała być dostarczona dopiero w następnej wersji”

- c. „nie dostarczyliście funkcjonalności zapisanej w paragrafie 24, punkt 5a umowy”, „ależ skąd – dostarczyliśmy”
4. Chaotyczne zarządzanie zmieniającymi się wymaganiami
 - a. „no proszę państwa, dodajcie tutaj taką małą tabelkę w środku ekranu, to wam zajmie tylko chwilkę”
 - b. „myśleliśmy, że ta zmiana nie będzie dla was tak istotna”
 5. Programiści pracujący w trybie 24/7
 - a. „przywieźcie nam pizzę o północy i dajcie dużo napojów energetyzujących”
 - b. „dajcie nam więcej programistów” (nowi programiści – nowe problemy?)
 6. Stres pod koniec projektu
 - a. „ten system pracuje w tempie żółwia”
 - b. „ale jeszcze nie sprawdziliśmy, czy działa ponad połowa funkcjonalności”
 7. Brak stabilności rezultatów między projektami
 - a. „to ile tym razem przekroczymy budżet?”
 - b. „a jak mam napisać te wymagania?” (to właściwie jak pisaliśmy je ostatnio?)
 8. System „prawie gotowy”
 - a. „system był już w 90% gotowy, a tu okazało się, że jest jeszcze tyle do zrobienia”
 - b. „oto gotowy system, z tym, że poprawa jego wydajności wymaga jeszcze pół roku pracy”

Powyższe symptomy zostały zebrane na podstawie obserwacji (w tym obserwacji autorów) przeprowadzonych w rzeczywistych projektach. Warto zastanowić się, czy takie symptomy nie występują w naszych projektach. Tylko bowiem stwierdzenie objawów choroby może nam umożliwić zastosowanie niezbędnej „kuracji”.

1.3. Przyczyny problemów

Jak wynika z przedstawionych wyżej badań, problemy nie dotyczą tylko pojedynczych projektów, ale całego przemysłu dostarczającego oprogramowanie. Jak nazwać ten stan? Czy mamy kryzys? Takie sformułowanie pojawiło się już w 1968 roku na wspomnianej już konferencji NATO na temat inżynierii oprogramowania. Od tamtego czasu, niestety niewiele się zmieniło. Świadczy o tym chociażby to, że klasyczna książka Freda Brooksa („Mityczny osobo-miesiąc”, 1975 r.) pozostaje nadal aktualna i wymagała jedynie krótkiego uwspółcześnienia w swym wydaniu jubileuszowym (1995 r.). W 1994 Wyatt Gibbs nazwał sytuację chronicznym kryzysem oprogramowania. Najbardziej odpowiednim określeniem jest jednak chyba to użyte przez Rogera S. Pressmana – „chroniczna choroba”.

Jak jest przyczyna tej „chronicznej choroby”? Na to pytanie w zasadzie odpowiedź jest jedna: z braku panowania nad wyjątkowo złożonym produktem, jakim jest system oprogramowania. Kluczowymi punktami są tutaj wymagania użytkowników, architektura i kontrola jakości. Wymagań użytkowników dla jednego systemu są zazwyczaj setki. Co więcej, są one sformułowane najczęściej przez odbiorców w taki sposób, że można na ich podstawie zbudować kilka całkowicie różnie działających systemów i nadal być z nimi zgodnym. Dlatego bardzo istotne jest wykonanie analizy wymagań precyzyjnie określającej sposób funkcjonowania systemu. Co więcej, ten opis powinien być zrozumiałym nie tylko dla

użytkowników i analityków, ale również dla projektantów i programistów. Zadanie całkiem karkołomne, ale jeśli go nie wykonamy, to mamy problem! Mamy również problem, jeżeli nie założymy, że wymagania się zmieniają. Brak panowania nad zmieniającymi się wymaganiami to bardzo częsta przyczyna niepowodzeń Drugą przyczyną choroby oprogramowania jest architektura, a raczej jej brak. Architektura jest takim elementem projektu, który zapewnia realizację wymagań użytkownika, zapobiega „radosnej twórczości” programistów i znacznie redukuje złożoność tworzonych modeli. Jeżeli projektanci z dumą pokazują olbrzymie arkusze papieru, na których setki elementów i powiązań między nimi opisują strukturę naszego systemu, to mamy problem. Wreszcie kontrola jakości. Często kojarzymy ją z testowaniem. Rzeczywiście – niedostateczne testowanie to częsty grzech firm produkujących oprogramowanie. Równie poważnym grzechem jest jednak niedostateczna kontrola satysfakcji użytkowników. Jeżeli pokazujemy im działający system po raz pierwszy na kilka tygodni przed końcem projektu, to mamy problem.

Problemy przed jakimi stoi inżynieria oprogramowania wynikają często z zaniechania stosowania ogólnych zasad inżynierii. Bardzo często, zespoły tworzące oprogramowanie stosują raczej zasady pracy chałupniczej. Dlatego bardzo ważne jest, abyśmy zidentyfikowali przyczyny problemów i zaczęli je aktywnie pokonywać stosując zasady wiedzy inżynierskiej z zakresu inżynierii oprogramowania.

Poniżej przedstawimy główne przyczyny występowania symptomów opisanych w poprzedniej sekcji.

1. Nieprecyzyjne specyfikowanie oprogramowania. Język używany podczas formułowania wymagań często przypomina beletrystkę (np. powieści) niż precyzyjny opis techniczny. Przykład: „czy ‘konto’ oznacza ‘konto użytkownika’ czy ‘konto bankowe?’”
2. Zła komunikacja. Poszczególni uczestnicy projektu (analitycy wymagań, architekci, projektanci, użytkownicy) bardzo często rozmawiają ze sobą różnymi językami. Na przykład architekci używają obrazków niezrozumiałych dla analityków, a analitycy spisują wymagania, których nie są w stanie zrozumieć architekci (lub rozumieją opacznie). Często też, obieg informacji w projekcie jest tak długi, że praktycznie uniemożliwia wyjaśnianie wątpliwości. Przykład: „oh, to ten formularz jest taki istotny....”
3. Brak projektowania architektonicznego. Bardzo często pomija się całkowicie dyscyplinę projektowania. Przystępuje się do pisania kodu bezpośrednio po zebraniu wymagań. Wynika to z przeświadczenia, że projektowanie spowalnia prace. Niestety, w przeważającej większości projektów (wyjątkiem są bardzo małe systemy) brak projektu prowadzi do chaosu i częstych nieporozumień między programistami. Zauważmy, że w budownictwie nikt przy zdrowych zmysłach nie zbuduje domu bez projektu (choćż można tak zbudować budę dla psa). Przynosi to bardzo opłakane skutki. Przykład: (na dzień przed oddaniem systemu) „to właściwie na której maszynie umieścimy ten kod?”
4. Brak zarządzania złożonością systemów. Przyczyna ta wiąże się z projektowaniem systemu. Zarządzanie złożonością polega na umiejętności podziały systemu na mniejsze fragmenty (moduły, pakiety, komponenty). Bez takiego podziału przestajemy panować nad systemem – przestaje się on „mieścić w głowie”. Oznacza to, że każda zmiana, poprawka, czy uzupełnienie okupywanie są bardzo żmudnym i kosztownym odkrywaniem „jak to właściwie działa” czy też, „co tak właściwie mieliśmy na myśli”. Przykład: „nasza baza danych ma 1500 tabel i 2000 klas dostępowych, to wszystko w jednym pakietie, a jakoś tam sobie dajemy radę”
5. Bardzo późne odkrywanie poważnych nieporozumień. Złożoność problemu i budowanego systemu powoduje, że często dochodzi do nieporozumień między uczestnikami projektu. Odkrycie takiego nieporozumienia późno w procesie wytwarzania systemu może oznaczać dramatyczny wręcz wzrost kosztów. Szczególnie nieporozumienia dotyczące wymagań są opłakane w skutkach i mogą skutkować koniecznością rozpoczęcia budowy systemu praktycznie od początku. Przykład: „i dopiero teraz mi mówisz, że ta procedura nie ma parametru X?”

6. Brak zarządzania zmianami. Dla większości projektów produkcji oprogramowania można sformułować jeden pewnik: zmiany będą występować. Zmiany te spowodowane są różnymi czynnikami, ale podstawowym ich źródłem są zmieniające się wymagania. Zmianom tym nie można zapobiec (wynikają często z przyczyn obiektywnych), jednak brak zarządzania zmianami prowadzi do chaosu. Bez kontroli zmian nie sposób ocenić, jak duży będzie system po dokonaniu zmian. Prowadzi to do „puchnięcia systemu”, czyli znacznego przekroczenia zakładanego zakresu prac.
7. Nieużywanie narzędzi wspomagających. Bez narzędzi stoiemy w sytuacji architekta, który projekt budynku musi narysować przy pomocy rysika i ekierki. Jest to możliwe do wykonania, ale bardzo nieefektywne w erze narzędzi CAD. Mimo to, typową postawą zespołu twórczego jest: „wszystko, czego nam trzeba, to dobry kompilator”.

Dobrą metodą oceny aktualnego stanu organizacji wytwarzającej oprogramowanie jest przeprowadzenie tzw. badania dojrzałości do produkcji oprogramowania. Zasady takiego badania zostały stworzone już w latach 80-tych XX wieku przez Software Engineering Institute Uniwersytetu Carnegie-Mellon (www.sei.cmu.edu) pod nazwą Capability Maturity Model for Software (SW-CMM).

Model CMM posiada pięć poziomów. Każdy z nich określa coraz wyższą dojrzałość organizacji. Przyporządkowanie do któregoś z poziomów stanowi dobrą diagnozę jej stanu. Pierwszy poziom to poziom początkowy (ang. initial). Jego cechą charakterystyczną jest brak procesu; działania podejmowane są ad-hoc, często w sposób chaotyczny. Na poziomie drugim – powtarzalnym (ang. repeatable) już istnieją (choć często nieformalnie) i są przestrzegane podstawowe standardy prowadzenia projektów. Poziom trzeci zakłada istnienie zatwierzonego standardu procesu twórczego, który jest przestrzegany we wszystkich projektach. Jest to zatem poziom zdefiniowany (ang. defined). Poziomy czwarty (zarządzany) i piąty (optymalizujący) oznaczają coraz bardziej szczegółowe przestrzeganie najlepszych praktyk inżynierii oprogramowania.

1.4. Najlepsze praktyki inżynierii oprogramowania

Coraz więcej organizacji wytwarzających oprogramowanie (chociaż nadal zdecydowanie zbyt mało) zaczyna stosować metody inżynierskie w produkcji oprogramowania. Jakie są zatem te metody? Jakie praktyki należy stosować, aby nie doświadczać symptomów „przewlekłej choroby”? Przedstawiamy tutaj zbiór najlepszych praktyk (ang. best practices) – sprawdzonych w praktyce zasad związanych z tworzeniem oprogramowania. Zasady te, zastosowane w połączeniu, uderzają w podstawowe przyczyny problemów i niepowodzeń procesu. Nazwa „najlepsze praktyki” wynika nie z faktu, że możemy zmierzyć ich poziom „dobroci”. Wynika raczej ze stwierdzenia, że są one najpowszechniej używane w organizacjach, które odnoszą sukcesy w produkcji oprogramowania. Najlepsze praktyki zostały zaadoptowane od tysięcy takich organizacji i były stosowane w tysiącach projektów. Oto one:

- Produkuj iteracyjnie, czyli oddawaj system w sposób przyrostowy. Kolejne przyrosty zapobiegają jednemu „wielkiemu wybuchowi” pod koniec projektu.
- Stosuj architektury komponentowe, czyli panuj nad złożonością systemu, miej spójną koncepcję całości i określ ramy dla radosnej twórczości projektantów i programistów. Panowanie nad złożonością systemu ułatwiają komponenty realizujące zasadę ukrywania informacji. Stosowanie komponentów promuje też zasadę ponownego wykorzystania (ang. reuse). Znamienne jest, że na tej samej konferencji NATO, na której ogłoszono kryzys oprogramowania, M. D. McIlroy opisał ideę komponentów, która dopiero obecnie może stać się panaceum na ten kryzys.

Tomek
2023-05-31 19:55:57

Najlepsze praktyki inżynierii oprogramowania

- Stale kontroluj jakość, przy czym podkreślić należy słowo „stale”. Kontrola jakości nie powinna się ograniczać do testowania. Powinna polegać na ciągłym sprawdzaniu zgodności systemu z wymaganiami i poziomu satysfakcji odbiorców.
- Zarządzaj wymaganiami, czyli traktuj wymagania jak obiekty, które podlegają obróbce. Każde z wymagań powinno posiadać ślad łączący je z wymaganiami szczegółowymi, projektem interfejsu użytkownika, czy komponentami, które je realizują.
- Zarządzaj zmianami, czyli z góry załóż, że zmiany na pewno będą ciągle zgłaszane. Często przekłada się to na istnienie oficjalnego „organu zgłaszania zmian”.
- Modeluj wizualnie, czyli dobry rysunek jest wart więcej niż tysiąc słów. Językiem budowy systemu powinien być język graficzny. Język powinien być wspólny i zrozumiały dla wszystkich włącznie z przyszłymi użytkownikami!

Zaaplikowanie wszystkich tych praktyk pozwoli nam znaleźć się co najmniej na drugim lub trzecim poziomie CMM. Ważne jest jednak, aby nie aplikować ich wszystkich na raz! Doświadczony lekarz prowadzący kurację stopniowo. Dlatego tak ważna jest ocena stanu organizacji. Dopiero na tej podstawie możemy opracować plan działania. Warto zauważyc, że poszczególne praktyki wzajemnie się uzupełniają. Trudno np. sobie wyobrazić ciągłą kontrolę jakości bez cyklu iteracyjnego.

Powysze praktyki koncentrują się przede wszystkim na procesie technicznym i odpowiednim ujęciu go w formalne ramy konkretnych działań w różnych dyscyplinach. Inne podejście do praktyki inżynierii oprogramowania zostało wyrażone w tzw. manifeście zwinnego (ang. agile) wytwarzania oprogramowania (agilemanifesto.org).

„Odkrywamy nowe metody programowania dzięki praktyce w programowaniu i wspieraniu w nim innych. W wyniku naszej pracy, zaczęliśmy bardziej cenić:

- Ludzi i interakcje od procesów i narzędzi,
- Działające oprogramowanie od szczegółowej dokumentacji,
- Współpracę z klientem od negocjacji umów,
- Reagowanie na zmiany od realizacji założonego planu.

Oznacza to, że elementy wypisane po prawej są wartościowe, ale większą wartość mają dla nas te, które wypisano po lewej.”

Manifest – sformułowany przez grupę znanych praktyków wytwarzania oprogramowania – koncentruje się na aspekcie organizacyjnym i konieczności „zwinnego” (stąd nazwa) dostosowywania się do zmieniających się w projekcie warunków. Należy jednak podkreślić, że podejście zwinne nie neguje stosowania odpowiednich czynności technicznych (proces, narzędzia, dokumentacja). Uważa ono jednak, że zdecydowanie ważniejsze są czynniki poprawiające współpracę w zespole i reakcję na zmiany. Bardzo istotnym elementem jest zapewnienie satysfakcji klienta poprzez umożliwienie mu wczesnego (w cyklach kilkutygodniowych) zapoznawania się z ciągle rozbudowywanym oprogramowaniem. Jednocześnie, w projektach zwinnych mile widziane są zmiany wymagań. Takie zmiany najczęściej „wykukują się” w codziennej współpracy zespołu deweloperskiego z przedstawicielami klienta. Główną miarą postępu w projektach zwinnych jest działające oprogramowanie, które jest stale weryfikowane przez klienta. Taka ciągła weryfikacja sprzyja również ciągłe poprawy efektywności pracy zespołu.

Najlepsze praktyki znajdują odzwierciedlenie w metodykach wytwarzania oprogramowania. Metodyki mogą być mniej lub bardziej sformalizowane oraz mogą obejmować różny zakres obszarów (dyscyplin) inżynierii oprogramowania. Metodyki oparte na podejściu zwinnym, zazwyczaj opierają się na zintegrowaniu aspektów organizacji pracy w zespołach wytwarzających oprogramowanie oraz zarządzania zmianami i innymi technikami w obszarze programowania czy testowania. Przykładami takich metod jest Scrum, XP, Lean, DSDM, Kanban, i wiele innych.

Tomek

2023-05-25 20:40:46

metodyki wykorzystujące podejście agile

eXtreme Programming (XP). Metodyki zwinne najczęściej nie formalizują opisu produktów technicznych w projekcie. Metodyki sformalizowane proponują natomiast szczegółowe wytyczne dla różnych czynności w procesie wytwarzania oprogramowania. Przykładem takiej metodyki jest Rational Unified Process (RUP) oraz jej „lekki” wariant – Open Unified Process (OpenUP). Opis każdej metodyki możemy oprzeć na trzech zasadniczych elementach: notacji, technikach i procesie technicznym.

1. Proces techniczny organizuje techniki wykorzystywane w procesie wytwórczym w spójny ciąg czynności prowadzący do celu, jakim jest zbudowanie systemu na czas, w budżecie i spełniającego rzeczywiste potrzeby jego odbiorców.
 - Proces techniczny pozwala zorganizować czynności cyklu życia oprogramowania. O sposobie organizacji cyklu wytwarzania oprogramowania mówi rozdział „Cykle wytwarzania oprogramowania”.
2. Notacja dostarcza jednolitego języka dla porozumiewania się członków zespołu.
 - Notacja pozwala na jednoznaczne dokumentowanie wszystkich decyzji podejmowanych w projekcie. Uwaga: dokumentujmy najlepiej w trakcie prac, a nie po. Dokumentacja powinna zawierać wiele elementów graficznych, które stanowią „plany” systemu. W module „Modelowanie oprogramowania” został zaprezentowany sposób dokumentowania systemów przy pomocy metod obiektowych i jednolitego języka modelowania obiektowego.
3. Techniki przedstawiają sposoby działania, które pokonują zasadnicze przyczyny niepowodzeń.
 - Techniki zarządzania wymaganiami koncentrują się na formułowaniu celów projektu z punktu widzenia odbiorców (klientów, użytkowników) oprogramowania. Zostały one przedstawione w rozdziałach 1 i 2 modułu „Podstawy inżynierii wymagań i projektowania oprogramowania”.
 - Techniki projektowania oprogramowania pozwalały panować nad złożonością systemów oprogramowania poprzez budowę ich planów architektonicznych i szczegółowych modeli kodu. Zostały one przedstawione w rozdziałach 3 i 4 modułu „Podstawy inżynierii wymagań i projektowania oprogramowania”.
 - Techniki programowania przedstawiają zasady formowania kodu oraz zarządzania kodem w ramach zespołów programistów. Zostały one przedstawione w rozdziałach 1 i 3 modułu „Podstawy rozwoju i eksploatacji oprogramowania”.
 - Techniki kontroli jakości dotyczą weryfikacji zgodności powstającego systemu z wymaganiami, czyli ściśle wiążą się z technikami zarządzania wymaganiami. Zostały one opisane w rozdziale 2 modułu „Podstawy rozwoju i eksploatacji oprogramowania”.

W rozdziale „Metodyki wytwarzania oprogramowania” zostały omówione wybrane metodyki wytwarzania oprogramowania wraz ze stosowanymi w nich notacjami, technikami i procesem technicznym.

1.5. Przykładowe dziedziny zastosowań inżynierii oprogramowania

We współczesnym świecie, oprogramowanie stanowi bardzo istotny element znacznej części systemów używanych na co dzień przez miliardy ludzi. Poniżej przedstawiamy przykładowe dziedziny, procesy i systemy, które mogą być wspomagane przez systemy oprogramowania. Krótkie omówienie tych obszarów pozwoli uzmysolić powszechność stosowania metod inżynierii oprogramowania.

Jednocześnie, przedstawione niżej obszary mogą stanowić podstawę do wykonania zadań zamieszczonych w poszczególnych rozdziałach niniejszych materiałów.

Dla każdej przedstawionej dziedziny zastosowań określone zostały przykładowe obszary możliwe do obsłużenia systemem oprogramowania, a także przykładowe procesy, które mogą być wspomagane takim systemem.

Bankowość elektroniczna

- Otwieranie rachunków i dokonywanie transakcji,
- Obsługa kredytów i kart kredytowych,
- Proces przyznawania kredytu (od złożenia wniosku o kredyt do przesłania informacji o decyzji i uruchomienia kredytu).

Biblioteka

- Obsługa księgozbioru oraz wypożyczeń,
- Rejestrowanie czytelników oraz rozliczenia z czytelnikami,
- Proces wypożyczenia książki (od złożenia zamówienia do wydania książki).

Biuro geodezyjne

- Obsługa prac geodezyjnych zlecanego przez różne podmioty,
- Rejestrowanie prac geodezyjnych w systemach informacji geodezyjnej,
- Proces wykonania zlecenia wytyczenia obszaru w terenie (od otrzymania zlecenia do zarejestrowania wyniku wytyczenia).

Biuro podróży

- Obsługa definiowania wycieczek i kontaktów z dostarczycielami usług turystycznych (hotelami, przewoźnikami),
- Obsługa sprzedaży i zakupu wycieczek,
- Proces uczestnictwa w wycieczce (od złożenia zamówienia do powrotu z wycieczki).

Ewidencja pojazdów i kierowców

- Zarządzanie wydziałami komunikacji (struktura organizacyjna, godziny pracy, itp.),
- Rejestracja i udostępnianie danych o prawach jazdy,
- Rejestracja pojazdów,
- Proces rejestracji pojazdów (od złożenia wniosku do otrzymania dowodu rejestracyjnego).

Firma bukmacherska i gier losowych

- Obsługa zakładów na wydarzenia sportowe i inne,
- Obsługa gier losowych, w tym – definiowanie gier losowych,
- Obsługa przeprowadzania gier losowych,
- Proces obsługi zakładu bukmacherskiego (od wykupienia zakładu do wypłacenia wygranej).

Firma kurierska

- Obsługa przesyłek kurierskich (paczki, dokumenty),
- Definiowanie struktury systemu transportowego (oddziały, sortownie) oraz floty transportowej,
- Zarządzanie kurierami (czas pracy, wynagrodzenia),

- Proces obsługi przesyłki kurierskiej (od nadania do dostarczenia).

Firma świadcząca przewozy pasażerskie (kolejowe, autobusowe, samolotowe)

- Obsługa rezerwacji i sprzedaży biletów,
- Definiowanie połączeń oraz rozkładów jazdy,
- Proces podróży środkiem transportu zbiorowego (od zdalnego zakupu biletu do dotarcia do celu podróży)

Firma telekomunikacyjna

- Obsługa sieci telekomunikacyjnej, łącznie z zarządzaniem urządzeniami sieciowymi,
- Obsługa użytkowników sieci – przyłączeń oraz rozliczeń za usługi telekomunikacyjne,
- Proces obsługi awarii (od zgłoszenia do rozwiązania problemu),
- Proces podłączenia abonenta do sieci (od zgłoszenia do uruchomienia urządzenia dostępowego).

Firma transportowa (spedycyjna)

- Obsługa hurtowego transportu samochodowego,
- Rejestracja przewozów oraz klientów,
- Proces transportu towarów (od przyjęcia zlecenia transportu do dostarczenia towaru do odbiorcy).

Klub sportowy (klub fitness)

- Obsługa definiowania obiektów sportowych wraz z obsługą zakupu i instalacji obiektów,
- Obsługa grafiku korzystania z obiektów wraz z obsługą personelu przypisanego do obiektów,
- Proces skorzystania z obiektu sportowego (od dokonania rezerwacji do wyjścia z obiektu).

Komisja wyborcza (krajowa, lokalna)

- Zarządzanie kandydatami, łącznie ze zgłoszaniem kandydatów i tworzeniem kart do głosowania,
- Obsługa procesu wyborczego, w tym – obsługa zliczania głosów i publikowania wyników,
- Proces zgłoszenia listy wyborczej (od zgłoszenia kandydatów do opublikowania karty wyborczej).

Park rozrywki

- Obsługa definiowania atrakcji, w tym obsługa zakupu i instalacji atrakcji,
- Obsługa korzystania z atrakcji przez klientów,
- Proces pobytu w parku rozrywki (od zakupu biletu do opuszczenia terenu parku).

Pomoc drogowa

- Zarządzanie flotą pojazdów pomocy drogowej oraz kierowcami/mechanikami,
- Obsługa zgłoszeń awarii drogowych oraz współpracy z firmami ubezpieczeniowymi,
- Proces pomocy w sytuacji awarii na drodze (od przesłania zgłoszenia awarii do rozliczenia płatności przez firmę ubezpieczeniową).

Serwis sprzętu AGD/RTV

- Definiowanie rodzajów serwisowanych urządzeń, łącznie z obsługą kontaktów z producentami,
- Obsługa kontaktów z klientami składającymi zlecenia naprawy sprzętu,
- Proces obsłużenia reklamacji (od złożenia reklamacji do dostarczenia naprawionego sprzętu).

Sklep internetowy

- Obsługa sprzedaży oraz definiowanie cenników, promocji itp.,
- Obsługa magazynu, w tym – zamówień i współpracy z dostawcami,
- Proces zakupu towarów (od wybrania towarów do dostarczenia towarów do klienta).

Sieć hoteli

- Obsługa definiowania hoteli, w tym – obsługa pozyskiwania nowych hoteli,
- Obsługa rezerwowania noclegów oraz pobytu w hotelach,
- Proces skorzystania z hotelu (od dokonania rezerwacji do wymeldowania po pobycie).

Sieć kasyn

- Zarządzanie kasynami i ich wyposażeniem (dostępne gry), cennikiem i stawkami,
- Zarządzanie rezerwacją gier (np. rezerwacja miejsca przy stole), kontrolą gier i wygranych,
- Zarządzanie przebiegiem gier i wypłat,
- Proces skorzystania z kasyna (od wejścia na teren do wypłacenia wygranej).

Sieć kin

- Zarządzanie kinami i ich wyposażeniem (sale, liczba miejsc, itp.),
- Zarządzanie personelem i obsługa zakupu biletów,
- Obsługa repertuaru (kopie filmów i grafik wyświetlania),
- Proces skorzystania z kina (od zakupu biletu do wyjścia z kina po seansie).

Sieć parkingów strzeżonych

- Definiowanie parkingów oraz obsługa dzierżawy terenów pod parkingi,
- Obsługa abonentów parkingowych oraz korzystania z parkingów przez klientów,
- Proces skorzystania z parkingu (od rezerwacji miejsca do wyjazdu z parkingu po zakończonym parkowaniu).

Sieć przychodni medycznych

- Obsługa wizyt, łącznie z zarządzaniem pacjentami i lekarzami,
- Zarządzanie wyposażeniem przychodni, łącznie z usługą zakupu i instalacji wyposażenia,
- Proces obsługi wizyty (od zapisania się na wizytę do wystawienia dokumentów po wizycie).

Sieć restauracji

- Zarządzanie strukturą sieci – definiowanie restauracji, menu i personelu,
- Realizacja zamówień zdalnych oraz w lokalach,
- Proces obsługi zamówienia zdalnego (od rezerwacji do dostawy potraw do domu)

Sieć salonów samochodowych

- Obsługa sprzedaży samochodów nowych i używanych
- Zarządzanie magazynem samochodów i dostawami od producentów
- Proces sprzedaży samochodu nowego (od zamówienia przez klienta do wydania samochodu)

Sieć stacji narciarskich

- Obsługa definiowania wyposażenia stacji i cennika,
- Obsługa korzystania z wyciągów, w tym obsługa zakupu i zwrotu biletów i karnetów okresowych,
- Proces skorzystania ze stacji narciarskiej (od rezerwacji karnetu do zakończenia pobytu).

Towarzystwo ubezpieczeniowe

- Obsługa definiowania polis oraz ich zakupu za pośrednictwem agentów,
- Obsługa polis – kontynuowanie polis oraz likwidacja szkód,
- Proces likwidacji szkody (od zgłoszenia szkody do przesłania decyzji i wypłacenia odszkodowania).

Uczelnia wyższa

- Obsługa programu studiów oraz planów zajęć,
- Obsługa prowadzenia zajęć, w tym – udostępniania materiałów dydaktycznych oraz wystawiania ocen,
- Proces przyznawania stypendium (od złożenia wniosku do przesłania informacji o decyzji i uruchomienia stypendium).

Urząd skarbowy

- Zarządzanie zeznaniami podatkowymi i rozliczanie podatków,
- Zarządzanie kontrolami skarbowymi,
- Proces kontroli skarbowej (od utworzenia zlecenia kontroli do dostarczenia protokołu pokontrolnego).

Wydział ruchu drogowego policji

- Obsługa mandatów wystawianych na miejscu oraz zdalnie (np. fotoradary)
- Zarządzanie grafikiem patroli drogowych
- Proces obsługi mandatu (od zarejestrowania wykroczenia do opłacenia mandatu przez kierowcę)

Wypożyczalnia samochodów

- Zarządzanie flotą samochodów oraz siecią punktów wypożyczeń,
- Obsługa cennika oraz wypożyczeń,
- Proces wypożyczenia samochodu (od rezerwacji do zwrotu wypożyczonego samochodu)

Zakład produkcyjny

- Obsługa współpracy z klientami (oferty, zamówienia, itp.),
- Obsługa procesu produkcyjnego, w tym magazynu podzespołów,
- Proces realizacji zamówienia (od złożenia zamówienia do przekazania towarów firmie spedycyjnej).

Zarządcy sieci drogowej (autostrad)

- Definiowanie sieci drogowej – dróg i infrastruktury,
- Zarządzanie remontami dróg oraz zdarzeniami drogowymi (np. wypadkami),
- Zbieranie danych z czujników na drogach,
- Proces przydzielenia koncesji na prowadzenie punktu obsługi podróżnych (od ogłoszenia przetargu do uruchomienia punktu).

Zarządzający lasami państwowymi

- Definiowanie struktury organizacyjnej (leśnictwa, granice obszarów leśnych itp.),
- Zarządzanie sprzedażą materiałów leśnych (np. drewna, sadzonek),
- Zbieranie danych z czujników w lasach (temperatura, dym, wilgotność, itp.),
- Proces reakcji na pożar (od informacji z czujnika do zakończenia akcji gaśniczej).

Zadania

Zadanie 1

Zakładamy hipotetyczny duży projekt konstrukcji systemu oprogramowania dla konkretnej instytucji działającej w jednej z dziedzin (patrz sekcja 1.5). Zidentyfikuj i opisz możliwe problemy, jakie mogą wystąpić w takim projekcie. Opisz jedynie objawy problemów.

Zadanie 2

Dla systemu z zadania 1 spróbuj określić w jaki sposób będą zapisywane wymagania oraz jak zostaną sformułowane plany projektowe.

Zadanie 3

Dla systemu z zadania 1 spróbuj określić, w jaki sposób będą wyglądać procedury jego validacji (testowania). Spróbuj określić zarówno testowanie poszczególnych składników kodu (klas, komponentów) jak i całego systemu pod kątem zgodności z wymaganiami.

Zadanie 4

Dla problemów opisanych w rozwiązaniu zadania 1 spróbuj określić przyczyny ich wystąpienia. Zastanów się np. nad problemami związanymi ze złożonością systemu, komunikacją w zespole oraz zastosowanymi technologiami.

Zadanie 5

Dla problemów opisanych w rozwiązaniu zadania 1 spróbuj określić praktyki projektowe, które pozwolą pokonać lub co najmniej zmniejszyć skutki tych problemów.

Zadanie 6

Zastanów się, w jaki sposób można usprawnić komunikację w projekcie z zadania 1. Spróbuj opisać notacje i techniki, które pozwolą na lepsze porozumienie między członkami zespołu projektowego.

Słownik pojęć

CMM

Model (ang. Capability Maturity Model) definiujący pięć poziomów dojrzałości w zakresie zdolności do wytwarzania oprogramowania.

Inżynieria oprogramowania

Inżynieria oprogramowania dąży do ujęcia działań związanych z budową programów komputerowych w ramy typowe dla innych dziedzin inżynierii. W szczególności, inżynieria oprogramowania stosuje wiedę naukową, techniczną oraz doświadczenie w celu projektowania, implementacji, walidacji i dokumentowania oprogramowania.

Oprogramowanie aplikacyjne

System oprogramowania działający na typowych komputerach, tabletach czy smartfonach, wspierający różnego rodzaju czynności wykonywane w domu i w pracy.

Oprogramowanie wbudowane

System oprogramowania kontrolujący pracę urządzeń mechanicznych i/lub elektronicznych, działający w czasie rzeczywistym, tzn. reagujący na sygnały i wymagający czasu reakcji dostosowanego do ograniczeń narzuconych przez sterowane urządzenia.

Co trzeba zapamiętać

Inżynieria oprogramowania jako działalność inżynierska

Inżynieria oprogramowania jest jedną z najważniejszych współcześnie dziedzin wiedzy inżynierskiej. Każda osoba we współczesnym świecie ma do czynienia z dziesiątkami systemów, w których oprogramowanie stanowi zasadniczy składnik. Aby wytworzyć system oprogramowania w sposób efektywny, musimy zastosować typowe zasady metod inżynierskich, dostosowanych do specyfiki oprogramowania. Zasady te muszą uwzględniać olbrzymią złożoność systemów oprogramowania. Niestety, złożoność rodzi wiele problemów i prowadzi do częstych niepowodzeń podczas projektów konstrukcji oprogramowania. Świadczą o tym typowe objawy „chronicznej choroby”, jaka od lat trapi inżynierię oprogramowania. Pokonując zasadnicze przyczyny niepowodzeń należy zastosować najlepsze praktyki inżynierii oprogramowania, które są przedstawione w poszczególnych rozdziałach niniejszego podręcznika.

Dyscypliny inżynierii oprogramowania

Dyscypliny inżynierii oprogramowania związane są z typowymi etapami działań inżynierskich w dowolnej dziedzinie inżynierii, przy czym posiadają one istotne cechy wyróżniające je spośród dyscyplin sformułowanych ogólnie. Najczęściej w inżynierii oprogramowania wyróżniamy takie dyscypliny jak: wymagania, projektowanie, implementacja, walidacja, nadzór i środowisko pracy. W różnych metodykach może występować nieco inny podział na dyscypliny.

Kryteria powodzenia projektu

Z punktu widzenia uczestników projektu budowy oprogramowania można wyróżnić trzy kryteria warunkujące powodzenie. Można uznać, że projekt skończył się sukcesem, jeżeli a) projekt mieści się w budżecie, b) projekt mieści się w ramach czasowych, oraz c) dostarczony system spełnia rzeczywiste potrzeby klienta.

Problemy inżynierii oprogramowania

Inżynierię oprogramowania trapi „przewlekła choroba”, która ma szereg objawów. Najczęściej spotykane objawy to: niezadowoleni klienci, niezadowoleni dostawcy, kłopotnie o zakresu systemu, chaotyczne

zarządzanie zmieniającymi się wymaganiami, programiści pracujący w trybie 24/7, stres pod koniec projektu, brak stabilności rezultatów między projektami oraz syndrom „prawie gotowego” systemu. Przyczynami tych problemów są najczęściej: nieprecyzyjne specyfikowanie, zła komunikacja, brak projektowania architektonicznego, brak zarządzania złożonością systemu, bardzo późne odkrywanie poważnych nieporozumień, brak zarządzania zmianami, nieużywanie narzędzi wspomagających.

Najlepsze praktyki inżynierii oprogramowania

Odpowiedzią na problemy inżynierii oprogramowania jest stosowanie najlepszych praktyk, które są rezultatem doświadczeń z wielu projektów. Podstawowy zestaw takich praktyk obejmuje: produkcję iteracyjną, stosowanie architektur komponentowych, stałą kontrolę jakości, systematyczne zarządzanie wymaganiami, zarządzanie zmianami oraz modelowanie wizualne. Wyrazem najlepszych praktyk w podejściu zwinnym (agile) jest manifest zwinnego wytwarzania oprogramowania, który promuje nastawienie na czynnik ludzki, współpracę z klientem oraz ciągłe dostarczanie działającego oprogramowania. Najlepsze praktyki stanowią zasadniczy element współczesnych metodyk wytwarzania oprogramowania.

Rozwiązania zadań

Rozwiązanie zadania 1

Przykładowy projekt polega na stworzeniu systemu obsługi studiów na uczelni wyższej. Kilka hipotetycznych (przewidywanych lub rzeczywistych) problemów występujących w projekcie:

1. Spory między przedstawicielami kierownictwa uczelni o zakres systemu.
2. Brak akceptacji podstawowych funkcjonalności obsługi planu studiów przez samorząd studentki.
3. Konieczność kosztownej wymiany systemu bazy danych w końcowej fazie projektu z uwagi na niezadowalającą wydajność dostępu do przechowywanych danych.
4. Brak akceptacji przedstawicieli nauczycieli dla interfejsu użytkownika modułu zarządzania przedmiotami i ocenami.
5. W ostatnim miesiącu projektu programiści muszą dokonać znaczących modyfikacji całej struktury systemu z uwagi na brak niezbędnych interfejsów, który został stwierdzony dopiero pod koniec projektu.

Rozwiązanie zadania 5

Przykładowe sposoby na pokonanie problemów z rozwiązań zadania 1 (w kolejności jak w rozwiązańiu):

1. Zorganizowanie wspólnych warsztatów wymagań z udziałem doświadczonego moderatora.
2. Szybkie dostarczenie w pierwszym etapie (iteracji) projektu podstawowych funkcjonalności obsługi planu studiów. Uzyskanie informacji zwrotnej na podstawie obserwacji pracy przedstawicieli samorządu z pierwszym wydaniem systemu.
3. Wykonanie testów wydajnościowych systemu na wczesnym etapie projektu i powtarzanie tych testów w miarę rozwoju systemu. Szybka reakcja na niezadowalające rezultaty testów.
4. Przeprowadzenie testów interfejsu użytkownika z udziałem przedstawicieli nauczycieli wystarczająco wcześnie w trakcie projektu, aby możliwe było dokonanie niezbędnych poprawek.
5. Wykonanie projektu architektonicznego całego systemu, uwzględniającego wszystkie interfejsy oraz uszczegóławianie oraz ew. aktualizacja tego projektu w trakcie kolejnych etapów (iteracji) projektu. Ciągła kontrola zgodności projektu architektonicznego z wymaganiami uczelni.

2. Cykle wytwarzania oprogramowania

Celem procesu wytwarzania oprogramowania jest dostarczenie zamawiającemu, spełniającego wymagania i sprawnego systemu oprogramowania. Z góry narzucone ograniczenia (np. czasowe, finansowe) oraz stopień złożoności problemu i technik jego produkcji wymuszają przedsięwzięcie odpowiednich czynności oraz właściwą ich organizację w planie realizacji projektu. Cechą każdego projektu mającego na celu stworzenie produktu jest ukierunkowanie wszystkich zadań objętych procesem twórczym na realizację wspólnego celu. W przypadku projektu informatycznego wspólnym celem jestowanie systemu oprogramowania.

Zakończenie projektu informatycznego sukcesem nie jest łatwym zadaniem. Różne stopnie skomplikowania problemów, często zmieniające się środowisko i wymagania na system oraz różne zasoby wymuszają indywidualne podejście do każdego projektu. Wytwarzanie oprogramowania można jednak przedstawić jako powtarzalny cykl rozwiązywania poszczególnych problemów, prac technicznych iłączenia rozwiązań. Na podstawie doświadczeń zebranych na przestrzeni lat opracowano wiele modeli procesów twórczych oprogramowania, które przedstawiają różne cykle wytwarzania oprogramowania.

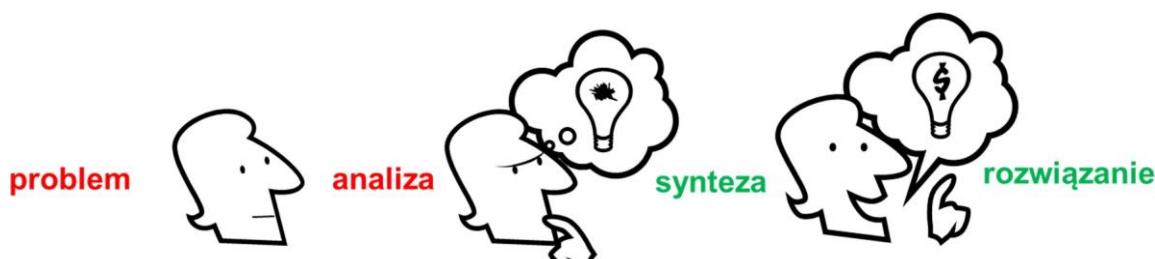
W niniejszym rozdziale przedstawiony zostanie zestaw typowych i powtarzalnych zadań, podzielonych na dyscypliny procesu wytwarzania oprogramowania. Ujęcie ich w trzy główne cykle twórcze pozwoli zrozumieć potrzebę uporządkowania czynności związanych z wytwarzaniem oprogramowania.

2.1. Dyscypliny cyklu wytwarzania oprogramowania

Podstawowe zasady podziału na dyscypliny

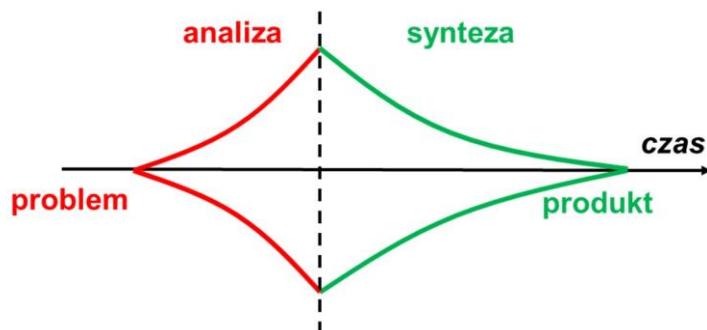
Celem każdego projektu jest wytworzenie pewnego produktu (lub produktów). Głównym produktem przedsięwzięcia informatycznego jest system oprogramowania, który jest uzupełniany innymi produktami, takimi jak: instrukcja użytkownika, specyfikacja wymagań, modele projektowe, itd. Produkty uzupełniające (zwane także pośrednimi) są efektem prac objętych kilkoma dyscyplinami. Podział procesu wytwarzania oprogramowania na dyscypliny zgodnie z zasadą dekompozycji, uwarunkowany jest dużą złożonością problemu i skomplikowanym procesem twórczym.

Czynności procesu wytwarzania oprogramowania można podzielić na dwie grupy: czynności analizy i czynności syntezy. Taki podział jest naturalną dla człowieka metodą radzenia sobie ze skomplikowanymi zadaniami twórczymi: analiza polega na dokładnym zidentyfikowaniu i zrozumieniu problemu, którego rozwiązanie osiąga się poprzez syntezę, czyli realizację i scalanie mniejszych części.



Rysunek 2.1. Analiza i synteza – naturalny sposób rozwiązywania złożonych problemów

Podążając za naturalną dla człowieka ścieżką analizy i syntezy (patrz rysunek 2.1) od problemu do jego rozwiązania, proces wytwarzania oprogramowania definiuje się jako ciąg czynności, podzielonych na dyscypliny, które prowadzą od postawienia problemu do wytworzenia produktu głównego (systemu oprogramowania) i produktów pośrednich. Do czynności analitycznych należą: opisanie środowiska, specyfikowanie wymagań, natomiast do czynności syntetycznych należą: projektowanie, implementacja, wdrożenie. Rysunek 2.2 obrazuje wzajemną zależność czynności analitycznych i syntetycznych. Produkty dyscyplin analitycznych, które prowadzą od ogółu do szczegółu, są podstawą do rozpoczęcia prac syntetycznych, które poprzez realizację i scalanie mniejszych części prowadzą do wytworzenia spójnego produktu.

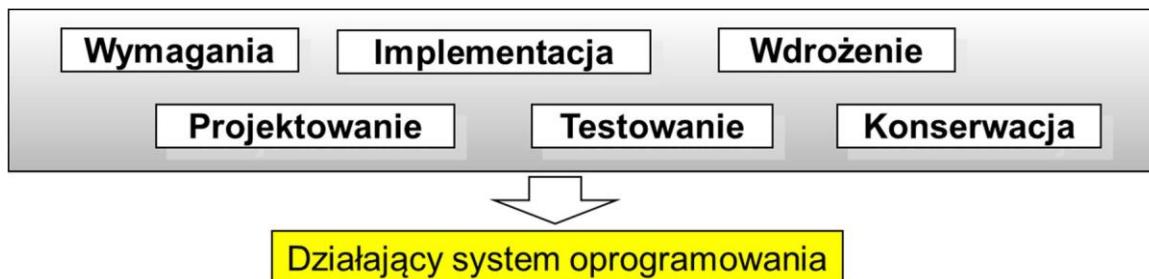


Rysunek 2.2: Analiza i synteza procesie wytwarzania oprogramowania

Przegląd dyscyplin

Ogólne określenie procesu wytwarzania oprogramowania jako sekwencji analizy i syntezy wyznacza jedynie ścieżkę od problemu do produktu. Praktyczne zastosowanie takiego podejścia inżynierskiego prowadzi do różnych rozwiązań w zakresie zdefiniowania i uporządkowania konkretnych zadań. Duża różnorodność projektów informatycznych wymaga bowiem dostosowywania kształtu procesu do indywidualnych potrzeb. W następnej sekcji niniejszego rozdziału zostaną przedstawione główne typy cykli wytwórczych w inżynierii oprogramowania. Najpierw jednak przedstawimy uporządkowanie zadań w ramach różnych cykli życia. Metodą na takie uporządkowanie jest podział na dyscypliny inżynierii oprogramowania. Należy jednak zwrócić uwagę na to, że podział na dyscypliny nie oznacza podziału na etapy, czyli nie decyduje o uporządkowaniu czasowym wykonywania zadań w poszczególnych dyscyplinach. Takie uporządkowanie jest głównym elementem definicji cyklu życia.

Niezależnie od wybranego typu cyklu życia, każdy projekt wytwarzania oprogramowania obejmuje kilka dyscyplin. Każda z dyscyplin dostarcza przynajmniej jeden produkt (pośredni). Produkty pośrednie są punktami wyjścia dla czynności wykonywanych z tej lub innych dyscyplinami. Na rysunku 2.3 przedstawiono główne dyscypliny procesu wytwarzania oprogramowania w kolejności zgodnej ze ścieżką „analiza – synteza”.



Rysunek 2.3: Dyscypliny cyklu wytwarzania oprogramowania

Dyscyplina wymagań dostarcza informacji o kształcie budowanego systemu z punktu widzenia klienta (np. użytkowników). Jest to określenie dość ogólne, albowiem na kształt systemu ma wpływ wiele czynników. Główne z nich to:

- środowisko, w którym system będzie funkcjonował,
- zadania, które system będzie realizował,
- sposób, w jaki system będzie funkcjonował.

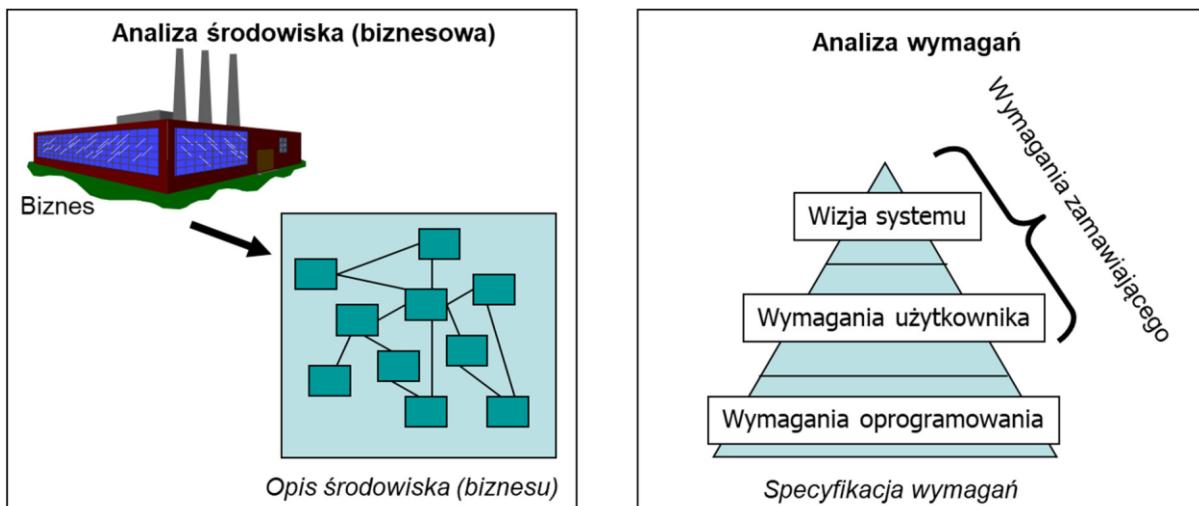
Główne produkty dyscypliny wymagań to opis środowiska (np. środowiska biznesowego) i specyfikacja wymagań. Na podstawie specyfikacji wymagań, dyscyplina projektowania tworzy zbiór modeli projektowych (architektura, projekt bazy danych, projekty szczegółowe komponentów, projekty dynamiki działania komponentów). Dyscyplina implementacji realizuje założenia określone w modelach projektowych. Powstaje działający system, który jest gotowy do kontroli jakości w ramach dyscypliny testowania. Pomyślne przejście testów umożliwia wdrożenie systemu w środowisku produkcyjnym (dyscyplina wdrożenia), co obejmuje również m.in. przeszkolenie użytkowników i przygotowanie dokumentacji technicznej. Ostatecznym produktem jest działający system informatyczny, który powinien podlegać ciągłe utrzymaniu w ramach dyscypliny konserwacji.

Poniżej omawiamy najważniejsze cechy poszczególnych dyscyplin. Szczegóły dotyczące każdej z nich znajdują się w pozostałych rozdziałach tego podręcznika.

Dyscyplina wymagań

Po określeniu problemu podejmowana jest decyzja o rozpoczęciu budowy systemu oprogramowania, który będzie rozwiązywał zadany problem. Podstawą określenia potrzeb klienta wynikających z zadanego problemu jest specyfikacja wymagań. Głównym celem dyscypliny wymagań jest zatem określenie zakresu i kształtu przyszłego systemu. Rysunek 2.4 przedstawia ogólny podział i strukturę dyscypliny wymagań. W ramach proponowanego przez nas podziału, dyscyplina ta dzieli się na dwie zasadnicze dyscypliny składowe – analizę środowiska (np. biznesu) oraz analizę wymagań.

W ramach analizy środowiska opisujemy aktualny oraz docelowy stan środowiska (biznesu). Wykonywane jest to przy współpracy przedstawicieli klienta (np. zamawiającego system). Na tej podstawie definiuje się główne potrzeby użytkowników systemu, które po uwzględnieniu ograniczeń środowiskowych tworzą wymagania zamawiającego. Poprawnie zdefiniowane wymagania zamawiającego zwykle są podstawą do zawarcia kontraktu między zamawiającym i wykonawcą na dostarczenie produktu spełniającego mieszczące się w tym dokumencie wymagania. Kolejną częścią specyfikacji wymagań jest opis wymagań oprogramowania, przedstawiający szczegóły działania systemu (m.in. szczegółowy opis interakcji systemu z użytkownikiem, wygląd okien).



Rysunek 2.4: Główne elementy dyscypliny wymagań

Opis środowiska (biznesu) stanowi pierwszy zasadniczy produkt dyscypliny wymagań. Jest on opisem fragmentu świata, który jest obszarem działań danej organizacji wraz ze środowiskiem, w jakim biznes działa. Formułowanie takiego opisu polega na stworzeniu precyzyjnego i zrozumiałego modelu, który zawiera wszystkie elementy środowiska (biznesowego) oraz zasady oddziaływania ich na siebie. Takie model składają się zazwyczaj z setek lub nawet tysięcy różnych elementów oddziałujących na siebie w określony, złożony sposób. Podstawowe elementy organizacji biznesowej to współpracownicy, pracownicy, jednostki organizacyjne, produkty, surowce, podzespoły, dokumenty, systemy informatyczne. Jednym z tych elementów jest budowany system informatyczny. Modelowanie może przebiegać na różnym poziomie abstrakcji: możemy modelować całą firmę lub tylko jakiś jej fragment (np. dział czy placówkę). Model może opisywać tylko stan docelowy (po zbudowaniu systemu informatycznego), lub być uzupełniony o opis stanu aktualnego.

Opis środowiska zawiera zwykle słownik pojęć (biznesowych) reprezentujący strukturę środowiska (biznesowego) oraz składniki procesów (biznesowych). Procesy opisywane są przy wykorzystaniu pojęć ze słownika pojęć. Procesy biznesowe są serią powiązanych ze sobą działań, które prowadzą do osiągnięcia celu biznesowego. Dokładna ich analiza dostarcza informacji na temat rzeczywistych potrzeb użytkowników, które są podstawą do zdefiniowania tzw. wymagań funkcjonalnych. Analiza środowiska, w którym realizowane są procesy biznesowe, dostarcza informacji na temat uwarunkowań środowiskowych, które są podstawą do sformułowania tzw. wymagań jakościowych (inaczej: pozafunkcjonalnych) na budowany system.

Kompletna **specyfikacja wymagań** wynika bezpośrednio z opisu biznesu i można ją porównać do piramidy (patrz ponownie rysunek 2.4). Na samym szczycie znajduje się wizja systemu, która dostarcza informacji na temat ogólnych cech systemu w ścisłym powiązaniu z potrzebami biznesowymi klienta. Zakres systemu wyznaczają wymagania użytkownika w postaci uporządkowanego zbioru cech i funkcji systemu, które powinny być podstawą do zawarcia kontraktu. Wizja systemu i wymagania użytkownika tworzą wymagania zamawiającego, które są odzwierciedleniem rzeczywistych potrzeb zdefiniowanych przez biznes. Na najniższym poziomie piramidy są wymagania oprogramowania, które opisują szczególne funkcjonalne komunikacji między użytkownikami i systemem, wszystkie wymieniane między nimi dane oraz planowany wygląd systemu.

Specyfikacja wymagań, obok ogólnych cech systemu, zawiera opis jego dynamiki oraz struktury. Słownik pojęć (biznesowych) jest podstawą do stworzenia słownika dziedziny. Słownik dziedziny jest rozbierany w trakcie opisywania cech systemu, wymagań funkcjonalnych i jakościowych. Słownik

przyjmuje ostateczny kształt na koniec tworzenia specyfikacji wymagań. Słownik zapewnia spójność całej specyfikacji wymagań i jest istotnym jej elementem. Na podstawie takiej spójnej i kompletnej specyfikacji wymagań można zbudować system informatyczny, będący naturalnym fragmentem rzeczywistości przedstawionej w opisie biznesu i realizujący rzeczywiste potrzeby zamawiającego.

Dyscyplina projektowania

Syntesa problemu opisanego w ramach dyscypliny wymagań oparta jest na projekcie systemu wykonywanym w ramach dyscypliny projektowania. Jako produkt wejściowy przyjmuje się tutaj specyfikację wymagań i opis środowiska biznesu. Celem działań tej dyscypliny jest stworzenie modeli projektowych na różnych poziomach abstrakcji, według których zostanie zaimplementowany system. Projektowanie często zaczynamy od najbardziej ogólnych (abstrakcyjnych) modeli, a potem - w zależności od złożoności systemu - przechodzimy do bardziej szczegółowych poziomów (podejście top-down). Możliwe jest również podejście odwrotne, w którym zaczynamy od szczegółów, a dopiero potem syntetyzujemy widok ogólny systemu (podejście bottom-up).

Mając do dyspozycji opis środowiska biznesu i specyfikację wymagań można zdefiniować, jakich elementów fizycznych (np. serwer bazy danych, serwer aplikacyjny, maszyny klienckie, urządzenia mobilne) będzie potrzebował budowany system i w jaki sposób te elementy będą się ze sobą komunikować. Jest to najbardziej ogólny, **fizyczny model architektoniczny**. Nie definiuje on logiki działania systemu, która definiowana jest na poziomie **architektury logicznej**. Na tym poziomie definiuje się moduły wykonawcze (komponenty), z jakich będzie się składał budowany system oraz powiązania komunikacyjne (np. interfejsy) między tymi modułami. W często stosowanych technologiach opartych na usługach lub mikrousługach (ang. service, microservice), komponenty w ramach logicznego modelu architektonicznego reprezentują poszczególne usługi. Interfejsy, które są specyfikacją funkcji udostępnianych przez komponenty architektoniczne, pozwalają traktować moduły jako czarne skrzynki. Na tym poziomie abstrakcji nie skupiamy się na tym, „co jest w środku” komponentów. Skupiamy się na tym, jakie funkcje komponenty spełniają.

Istotnym elementem modelu architektonicznego i zależnych od niego modeli szczegółowych są struktury danych, które służą do komunikacji poprzez interfejsy. Ich podstawą może być słownik dziedziny, zdefiniowany w specyfikacji wymagań. Na podstawie słownika możemy zdefiniować tzw. **obiekty transferu danych** (ang. Data Transfer Object). Określają one „paczki danych”, które mogą być przesyłane między poszczególnymi komponentami systemu.

Kolejnym poziomem projektowania jest stworzenie **szczegółowych projektów modułów**. Na tym poziomie definiujemy konkretne szczegóły „czarnych skrzynek”. Dokładnie przedstawiamy strukturę oraz opracowujemy sekwencje wykonywanych operacji, których celem jest realizacja funkcji określonych w poszczególnych interfejsach komponentów. W zależności od złożoności systemu, może być także wymagane zaprojektowanie algorytmów, które realizują złożone funkcjonalności wewnętrz systemu (np. algorytmy szyfrujące, obliczające składki ubezpieczenia czy wykonujące obliczenia na macierzach).

Produktem projektowania jest zbiór spójnych modeli, które dostarczą szczegółowych informacji o składnikach systemu, który ma spełniać wymagania opisane w specyfikacji wymagań i funkcjonować w opisany środowisku. Modele projektowe bezpośrednio odzwierciedlają konstrukcje programistyczne i są bezpośrednio wykorzystywane podczas kodowania systemu, czyli jego implementacji.

Dyscyplina implementacji

W wyniku działań objętych dyscypliną implementacji zostaje wytworzony **kod (program)** realizujący wymagania zdefiniowane podczas analizy i spełniający założenia projektowe, określone w modelach

projektowych. Do wytwarzania działającego systemu wykorzystuje się zbiór technologii, które są zgodne z wymaganiami technicznymi i wpisują się w środowisko działania budowanego systemu.

Proces programowania wymaga odpowiedniego środowiska i organizacji pracy. Aby stworzyć system, który będzie prawidłowo funkcjonował w **środowisku produkcyjnym** klienta, należy takie środowisko zbudować również w wersji testowej (implementacyjnej). Czynności z tym związane mogą obejmować np. instalację serwera i potrzebnych bibliotek w wersjach wykorzystywanych przez organizację, uruchomienie dedykowanych rozwiązań lub tzw. zaślepek potrzebnych do funkcjonowania systemu. Duży wpływ na kształt **środowiska implementacyjnego** mają wymagania jakościowe i ograniczenia środowiska biznesowego.

Implementacja systemu oznacza implementację poszczególnych modułów (komponentów) zaprojektowanych w ramach dyscypliny projektowania. Liczba modułów do zaimplementowania zależy jest oczywiście od stopnia skomplikowania systemu. Każdy moduł może być realizowany niezależnie od pozostałych (przez różnych programistów), ale powinien poprawnie z nimi współpracować. W szczególności, w architekturach komponentowych zadaniem programistów jest zaprogramowanie realizacji interfejsów dostarczanych przez komponenty, zgodnie z założeniami projektowymi.

Każdy zespół programistów realizujących pewną część systemu (zbiór modułów) odpowiedzialny jest za jej prawidłowe funkcjonowanie. Dlatego też, nawet najmniejsze „kawałki” kodu powinny być na bieżąco testowane podczas całego procesu implementacji. Testowaniu podlegają poszczególne elementy kodu (klasy, funkcje), a także moduły (zestawy klas) oraz ich interfejsy (zestawy funkcji). Programiści tworzą tzw. **testy jednostkowe**. Są to zestawy procedur, który celem jest sprawdzenie działania tworzonego kodu dla różnych sytuacji (dla różnych danych wejściowych).

Prace w ramach dyscypliny implementacji wymagają zastosowania odpowiednich narzędzi, tworzących tzw. zintegrowane środowisko deweloperskie (ang. Integrated Development Environment – IDE). Podstawą jest oczywiście dobre narzędzie służące do kompilacji kodu, jego wykonywania w środowisku implementacyjnym (testowym) oraz do przeprowadzania testów jednostkowych. Inne narzędzia wspomagają pracę grupową, kontrolę wersji kodu oraz integrację całego systemu. Niezależnie jednak od stosowania dobrych narzędzi, gwarancją wyprodukowania poprawnego i optymalnego kodu jest wykorzystanie dobrej znajomości technologii i dobrych praktyk programistycznych. Pożądaną cechą kodów źródłowych, oprócz oczywistej zgodności z modelami projektowymi i wymaganiami, powinny być: wspólna konwencja programistyczna, dobra dokumentacja i gwarancja poprawności działania potwierdzona dokumentacją testów jednostkowych.

Po implementacji sprawnie działających (przetestowanych) modułów i ich połączeniu, całość powinna przejść **testy integracyjne**. Poprawność działania zostaje sprawdzona globalnie, dla całego systemu. Tak sprawdzony kod systemu jest głównym produktem dyscypliny implementacji. Pozostałe produkty to dokumentacja testów jednostkowych i integracyjnych oraz np. pliki wykonywalne i instalacyjne.

Dyscyplina testowania

Celem testowania jest sprawdzenie, czy zbudowany system oprogramowania spełnia wymagania określone w specyfikacji wymagań i działa poprawnie w środowisku biznesowym.

Testowanie takie powinno przebiegać w specjalnie przygotowanym środowisku testowym, które służy do uruchamiania systemu w sposób bardzo zbliżony do uruchamiania w środowisku produkcyjnym. W odróżnieniu od testów przeprowadzonych w czasie implementacji, głównymi odbiorcami testów są przedstawiciele zamawiającego. Zwykle są to przyszli użytkownicy systemu, wytypowani i specjalnie przygotowani do nadzorowania i wykonywania czynności związanych z testowaniem. Poza

sprawdzeniem poprawności integracji, które dokonywane jest przez osoby odpowiedzialne za infrastrukturę informatyczną organizacji, sprawdzana jest również zgodność z wymaganiami pochodząymi z biznesu i oczekiwaniami do działania systemu (funkcjonalność, ergonomia).

Przeprowadzenie **testów akceptacyjnych** systemu (ang. *acceptance test*) wymaga **planu testów**. Zazwyczaj plan testów jest dokumentem powstającym na podstawie specyfikacji wymagań. Poszczególne testy ułożone są w odpowiedniej kolejności, która pozwala na prawidłowe sprawdzenie prawidłowości działania poszczególnych funkcjonalności systemu. Testy takie przypominają instrukcje postępowania przygotowane dla użytkowników systemu. Rezultatem jest powstanie kompletnych **scenariuszy testów**. Rezultatem wykonania konkretnego scenariusza testów jest potwierdzenie poprawności działania lub opis błędu. W przypadku niepomyślnego wyniku testów, może nastąpić powrót do działań w ramach innych dyscyplin (projektowanie, implementacja), a potem ponowienie testów po zaimplementowaniu poprawek.

Wynikiem dyscypliny testowania jest dokumentacja testów akceptacyjnych, która jest podstawą do zatwierdzenia systemu i przejścia do czynności objętych dyscypliną wdrożenia.

Dyscyplina wdrożenia

Po pomyślnym przejściu testów i zatwierdzeniu uruchomienia aktualnej wersji budowanego systemu oprogramowania w środowisku produkcyjnym, rozpoczynają się czynności dyscypliny wdrożenia. Dyscyplina ta obejmuje czynności związane z instalacją systemu oraz umożliwieniem korzystania z systemu przez jego użytkowników. Instalacja systemu obejmuje zarówno przygotowanie środowiska produkcyjnego do wdrożenia nowego systemu, jak i samo fizyczne umieszczenie plików wykonywalnych systemu w środowisku docelowym. Głównym produktem dyscypliny wdrożenia jest **zainstalowany system**, działający w docelowym środowisku wykonawczym. Innymi produktami są np. **dokumentacja dla użytkowników** oraz **dokumentacja techniczna**. Strategie działań w ramach dyscypliny wdrożenia zależą od rodzaju budowanego systemu. W dalszym opisie koncentrujemy się na najbardziej typowej sytuacji, tzn. budowie systemu na zamówienie dla klienta (np. dla organizacji biznesowej).

Można wyróżnić kilka najbardziej popularnych strategii wdrożenia nowego systemu do środowiska produkcyjnego:

- Wdrożenie bezpośrednie – nowy system zostaje wdrożony w całości. Jest to strategia niosąca duże ryzyko. Nowy system oprogramowania musi być wysokiej jakości – często usterki są wykrywane w trakcie działania. W niektórych przypadkach, jak pierwsza informatyzacja lub np. wymiana oprogramowania bankomatów, strategia ta jest zalecana.
- Wdrożenie równolegle – obok nowego systemu, przez pewien czas działa również stary. Zaletą tej strategii jest weryfikacja poprawności działania nowego systemu i możliwość wykorzystania starego w przypadku wykrycia błędów w nowym systemie lub jego awarii. Wadą jest natomiast duży koszt utrzymania i obsługi obu systemów.
- Wdrożenie pilotowe – zostaje uruchomiony fragment nowego systemu w celu minimalizacji ryzyka związanego z całkowitym wprowadzeniem nowego systemu.
- Wdrożenie stopniowe – strategia polegająca na wprowadzaniu w działalność organizacji kolejnych fragmentów nowego systemu, uniezależniając pozostałe obszary działalności od nowego systemu. Pozwala to zminimalizować ryzyka związane z zatrzymaniem działalności całej organizacji przez uruchomienie nowego systemu w całości. Wadą jest natomiast rozległe w czasie wdrożenie.

Warto zauważyć, że często wdraża się system według strategii hybrydowej, łączącej wybrane cechy dowolnych strategii. Dzięki temu możliwe jest dostosowanie czynności wdrożeniowych do specyfiki danej organizacji i wdrażanego systemu oprogramowania.

Istotnym elementem dyscypliny wdrożenia jest przygotowanie materiałów dla użytkowników oraz przeprowadzenie **szkoleń** dla przyszłych użytkowników. Forma materiałów oraz szkoleń jest uzależniona od wybranej strategii wdrożenia oraz rodzaju zbudowanego systemu. Podstawowym celem szkolenia jest nauka zasad działania i operowania systemem. Może to być realizowane poprzez dostarczenie użytkownikom podręczników, albo poprzez wbudowane w system materiały szkoleniowe (np. pomocniki kontekstowe). Szkolenia mogą być prowadzone na zasadzie bezpośredniego mentoringu w środowisku produkcyjnym, lub być wykonywane na bazie przygotowanych prezentacji szkoleniowych oraz pracy w osobnym środowisku szkoleniowym.

Efektem wdrożenia systemu jest działający system, sprawnie używany przez jego użytkowników. Wdrożenie powinno również dostarczyć ocenę zbudowanego i wdrożonego systemu (ew. jego wersji pośredniej). Jest to podstawą do dalszych działań w ramach obecnego projektu (kolejne wersje przyrostowe), w następnych projektach (rozbudowa systemu w przyszłości) oraz w ramach dyscypliny konserwacji.

Dyscyplina konserwacji

Czynności konserwacji, zwane także czynnościami utrzymania, stanowią zawsze ostatni etap w cyklu życia oprogramowania. Obejmują one zapewnianie poprawnego funkcjonowanie systemu w środowisku biznesowym. W trakcie pracy z systemem oprogramowania mogą zostać wykryte wcześniej nieznane defekty. Wymagana funkcjonalność lub cechy jakościowe systemu oprogramowania, które zostały zdefiniowane w ramach analizy wymagań, mogą ulec zmianie już po oddaniu systemu do użytku. Może zmienić się także samo środowisko i procesy biznesowe (np. zmiany organizacyjne w firmie, wdrożenie zmian w innym systemie, zmiana otoczenia prawnego). Wszystko te sytuacje wymuszają podjęcie pewnych działań, które można podzielić na cztery grupy:

- czynności naprawcze obejmujące usuwanie defektów oprogramowania;
- czynności adaptacyjne dostosowujące oprogramowanie do zmieniającego się środowiska;
- czynności ulepszające wprowadzające nowe funkcjonalności i zmianę już istniejących;
- czynności prewencyjne przygotowujące oprogramowanie do przyszłych modyfikacji.

W niektórych przypadkach wprowadzenie zmian w funkcjonującym systemie, może wymagać przejścia przez wszystkie dyscypliny, od wymagań do implementacji i wdrożenia. Wszelkie działania konserwacyjne powinny być starannie dokumentowane. Pozwala to unikać błędów we wprowadzaniu zmian i czyni system oprogramowania przejrzystym.

Podjęcie wyżej wymienionych czynności powinna poprzedzić analiza kosztów i ryzyka wprowadzania zmian w funkcjonującym systemie. W wyniku takiej analizy może się okazać, że bardziej opłacalne od utrzymania bieżącego systemu będzie wytworzenie nowego systemu. W takiej sytuacji kończy się cykl życia oprogramowania, prowadząc do rozpoczęcia kolejnego cyklu budowy i życia oprogramowania.

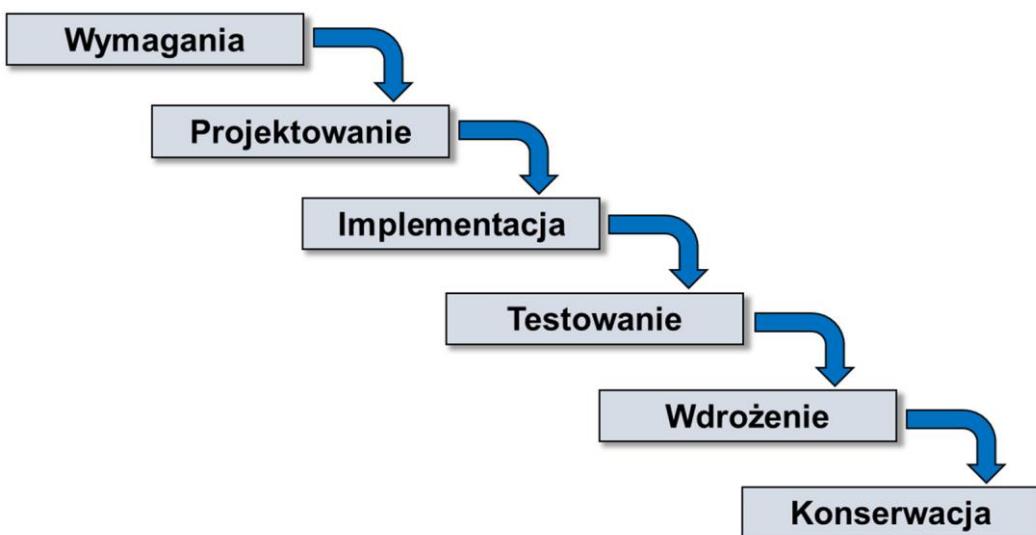
2.2. Przegląd cykli wytwarzania oprogramowania

Przedstawione wyżej dyscypliny określają zakres czynności i produktów wykonywanych w typowym projekcie wytwarzania oprogramowania. Czynności te tworzą pewien ciąg, który powinien być wykonywany w określonej kolejności. Taki ciąg czynności nazywamy procesem wytwarzania oprogramowania. W dobrze zorganizowanym projekcie, proces ten przebiega zgodnie z dobrze określonym cyklem wytwórczym, który jest elementem całego cyklu życia oprogramowania. Cykl wytwórczy definiuje rozłożenie czynności z poszczególnych dyscyplin w czasie.

W większości współczesnych projektów stosuje się jeden z dwóch cykli wytwarzających: cykl wodospadowy (również nazywany kaskadowym) lub cykl iteracyjny. Dodatkowo, rozwinięciem cyklu iteracyjnego jest cykl spiralny. Cykle wytwarzające posiadają warianty, charakterystyczne dla różnych metod i technik wytwarzających, które zostały opisane w kolejnym rozdziale. Z uwagi na różne uwarunkowania i przebieg projektów, zazwyczaj nie udaje się zastosować ściśle do wszystkich reguł przyjętych w wybranym cyklu wytwarzającym. Często eksponuje się pewne elementy kosztem innych, które są pomijane. Cykl wytwarzający jest rodzajem przewodnika, który wskazuje kierunek, który należy obrać, aby osiągnąć zamierzony cel, ale nie pokazuje drogi, którą należy pójść. Szczegóły tej drogi w szczegółach określa natomiast metoda.

Cykł wodospadowy

Definicja wodospadowego cyklu życia oprogramowania, zwanego także kaskadowym lub klasycznym, została sformułowana już w roku 1970. Główna idea tego cyklu jest ujęcie dyscyplin wytwarzania oprogramowania w sekwencję dobrze wyodrębnionych kroków (faz lub etapów), które prowadzą do zbudowania systemu oprogramowania. Rysunek 2.5 przedstawia ogólny schemat cyklu wodospadowego. Kolejne etapy wykonywane są raz w trakcie trwania projektu i następują po sobie. Nazwa tego cyklu wynika z podobieństwa tego schematu do wodospadu lub kaskady.

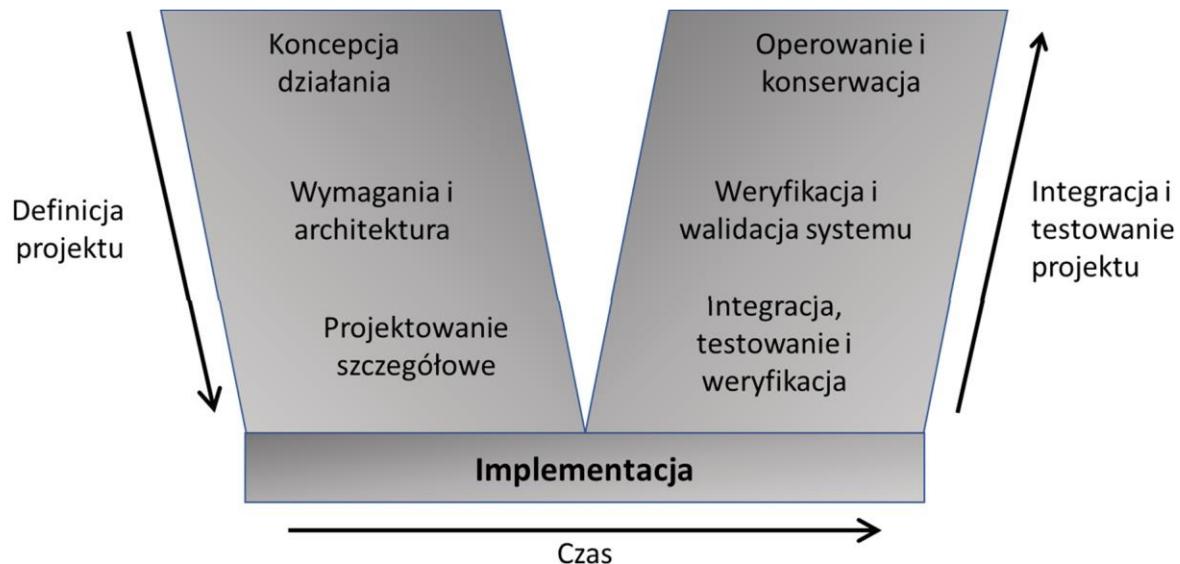


Rysunek 2.5: Wodospadowy cykl wytwarzania oprogramowania

Jak wskazuje rysunek, poszczególne etapy „wodospadu” odpowiadają dyscyplinom cyklu życia oprogramowania przedstawionym w poprzedniej sekcji. Każdy etap dostarcza co najmniej jednego produktu, który podlega weryfikacji i akceptacji. Następny etap może się rozpocząć tylko po zakończeniu poprzedniego i wykorzystuje jego produkty jako punkt wyjścia. W praktyce jednak, sąsiadujące ze sobą etapy zazębiają się i przekazują nawzajem informacje. Oznacza to, że w ramach jednego etapu możemy wrócić do poprzedniego etapu (tzw. wodospad z nawrotami). Można powiedzieć, że powtarzające się nawroty do poprzedniego etapu tworzą „mini-iteracje” (patrz opis cyklu iteracyjnego).

Wariantem cyklu wodospadowego jest tzw. cykl „V”. Jest on zilustrowany na rysunku 2.6. Cykl ten swoją nazwę zawdzięcza graficznemu ułożeniu faz projektu w kształt litery „V”. Pierwsze trzy fazy obejmują czynności definiowania systemu. Nieco inny jest podział dyscyplin. Na przykład, dyscyplina wymagań rozbita jest między fazy „konceptuała działania” (odpowiednik analizy środowiska) oraz „wymagania i architektura”. Po tych fazach następuje faza implementacji, oraz trzy kolejne fazy obejmujące czynności integrowania i testowania systemu. Zauważmy, że trzy ostatnie fazy odpowiadają trzem

pierwszym fazom. Oznacza to, że fazy w drugiej części cyklu służą do walidacji i weryfikacji rezultatów pierwszej części cyklu (w tym – implementacji).



Rysunek 2.6: Cykl „V” dla wytwarzania oprogramowania

Cykł wodospadowy w naturalny sposób odzwierciedla proces rozwiązywania złożonych problemów, opisany wyżej (najpierw analiza, a później synteza). Wszystkie podejmowane czynności prowadzą od potrzeb użytkownika do gotowego systemu w jednym przebiegu przez poszczególne grupy czynności (dyscypliny). Umożliwia to łatwe zarządzanie zasobami (zespoły specjalistów, sprzęt). Przejście do kolejnych faz wymaga jednak zamrożenia wyników prac w fazach poprzednich. Wynika to z konieczności zachowania spójności produktów w ramach całego cyklu. Konieczność powrotu do poprzednich faz oznacza odejście od czystego cyklu wodospadowego.

Cykł wodospadowy stanowi istotny krok w kierunku uporządkowania działań dotyczących wytwarzania oprogramowania. Posiada on niewątpliwe **zalety**, które powodują, że jest stosowany w wielu projektach. Podstawową zaletą jest prosta i naturalna struktura, odpowiadająca naturalnemu procesowi rozwiązywania problemów. Porządkuje on wszystkie czynności i umożliwia łatwe śledzenie postępów prac. Postęp prac mierzony jest kolejnymi produktami zdefiniowanymi w poszczególnych etapach, odpowiadającymi dyscyplinom inżynierii oprogramowania. Zarządzanie projektem realizowanym w cyklu wodospadowym jest stosunkowo proste. Wynika to z tego, że w poszczególnych fazach projektu wykorzystujemy zasoby ludzkie przypisane do określonej dyscypliny (analityków wymagań, projektantów, programistów, testerów). Efekty prac są jasno określone i stanowią podstawę do rozpoczęcia działań w kolejnych fazach projektu.

Niestety, cykl wodospadowy posiada **wady**, które rodzą zasadnicze zagrożenia. Podstawowym problemem jest póżna weryfikacja wyników prac. Problemy z realizacją wymagań klienta odkrywane są zazwyczaj dopiero w fazie testowania, a nawet dopiero w fazie wdrożenia. Prowadzi to często do realizacji syndromu 90%/90%. Pod koniec projektu, jego kierownictwo raportuje wykonanie 90% zadań. W trakcie testowania i wdrożenia okazuje się jednak, że do wykonania pozostaje nadal 90% prac, gdyż wiele funkcjonalności systemu nie jest zgodna z oczekiwaniemi klienta. Skutkuje to znacznym przesunięciem terminu oddania projektu. Efektem są podane w poprzednim rozdziale statystyki niepowodzeń projektów i średniego czasu przekroczenia ich terminów i budżetów. Zauważmy też, że istotnym czynnikiem jest tutaj subiektywizm oceny produktów. Na przykład, jakość specyfikacji wymagań możemy w pełni ocenić dopiero w fazie implementacji. Często dopiero wtedy okazuje się, że programiści

nie mogą wykonać pewnych elementów kodu, gdyż brakuje dokładnej specyfikacji dziedziny problemu. Oznacza to konieczność bardzo dalekiego nawrotu i wykonania dodatkowych czynności w ramach dyscypliny wymagań.

Inną wadą cyklu wodospadowego jest niewielka elastyczność w radzeniu sobie ze zmianami (w tym – zmianami zakresu). Bardzo trudno jest stworzyć poprawną i kompletną specyfikację już na początku projektu. Rzeczywiste potrzeby klienta są często odkrywane lub zmieniane już w trakcie trwania prac nad systemem oprogramowania. Szczególnie dotyczy to faz testowania i wdrożenia. Często dopiero wtedy klient zdaje sobie sprawę z tego, jakie skutki ma przyjęcie określonych założeń dotyczących funkcjonalności systemu lub jego cech jakościowych. Prowadzi to do w rezultacie do dosyć chaotycznych działań w celu „wymuszenia” zgodności z oczekiwaniemi, np. sporów o zakres projektu czy próby istotnych zmian warunków kontraktu. Każda jednak zmiana tego typu oznacza konieczność powrotu do początkowych faz projektu i ponowne przejście przez „wodospad”.

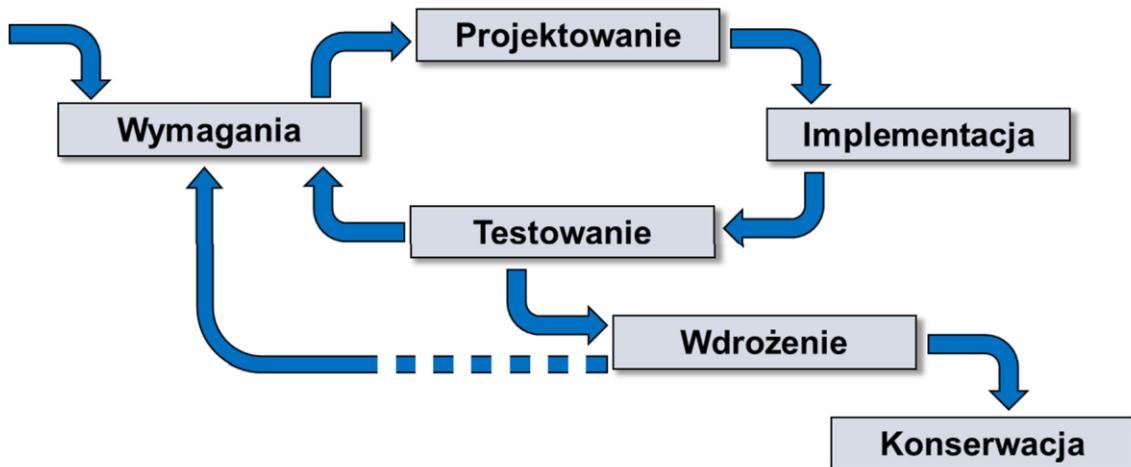
Kolejnym problemem cyklu wodospadowego jest niewielka możliwość optymalizacji prac, w tym – poprawy współdziałania między zespołami działającymi w różnych etapach (tu: dyscyplinach). Przekazanie produktów między etapami – w czystym procesie wodospadowym – następuje tylko raz. Oznacza to, że brakuje możliwości naprawy tych produktów w razie stwierdzenia usterek czy braków w kolejnych etapach. Oczywiście, taka naprawa jest zazwyczaj podejmowana, ale prowadzi to do działań wykraczających poza ustalony plan projektu. Oznacza to zatem często powstanie bałaganu w zarządzaniu projektem.

Wodospadowy cykl wytwarzania oprogramowania, mimo iż jest dosyć leciwy i posiada wiele wad, wciąż jest popularny. Jak wynika z analizy jego wad, powodzenie projektu realizowanego cyklem wodospadowym mocno zależy od stabilności i poprawności specyfikacji wymagań. Dlatego jest polecany do krótkich projektów, w których wymagania są dobrze określone i zrozumiałe, a ich specyfikacja jest w pełni przygotowana. W przypadku dłuższych projektów powinniśmy być pewni, że problem jest bardzo dobrze określony i nie będzie podlegał modyfikacjom.

Cykl iteracyjny

Podstawą dla stworzenia cyklu iteracyjnego była chęć eliminacji zagrożeń związanych z „zamrażaniem” produktów poszczególnych faz cyklu wodospadowego. Zagrożenia takie możemy wyeliminować przez wprowadzenie wielokrotnego przechodzenia przez powiązane ze sobą dyscypliny. To podejście jest główną cechą iteracyjnego cyklu wytwarzania oprogramowania. Ilustruje to rysunek 2.7. Jak widzimy, cykl iteracyjny zachowuje podział na dyscypliny, ale czynności w ramach wszystkich dyscyplin wykonywane są wielokrotnie.

W projekcie planowane jest wiele iteracji, gdzie każda **iteracja** trwa zazwyczaj kilka tygodni (najczęściej od 2 do 6). Poszczególne iteracje rozpoczynają się od wyboru i wyspecyfikowania określonego zakresu wymagań, odpowiadającego przyrostowi funkcjonalności systemu. Następnie następują czynności syntetyzowania problemu – projektowanie, implementacja oraz testowane. Czynności te dotyczą zakresu wymagań wybranych do danej iteracji oraz ew. wymagań, które nie zostały prawidłowo zrealizowane w poprzednich iteracjach. Testowanie przeprowadza się w sposób regresyjny. Dokładnie testuje się funkcjonalność wykonaną w danej iteracji, a także przeprowadza się mniej dokładne testy funkcjonalności zrealizowanych wcześniej. Po zakończeniu testów i akceptacji ich wyników przechodzimy do kolejnej iteracji. W razie stwierdzenia błędów lub uwag kod klienta, odpowiednie zadania są przydzielane do kolejnych iteracji.



Rysunek 2.7: Iteracyjny cykl wytwarzania oprogramowania

Po zakończeniu pewnej liczby iteracji możliwe jest wdrożenie systemu. W niektórych projektach wdrożenie wykonywane jest tylko raz – po ostatniej iteracji. Najczęściej jednak planuje się wdrożenia pośrednie. Takie podejście umożliwia dostarczenie klientowi działającego (choć jeszcze nie w pełni funkcjonalnego) systemu jeszcze w trakcie projektu. Po ostatnim wdrożeniu następuje faza konserwacji, podobnie jak w cyklu wodospadowym.

Zastosowanie cyklu iteracyjnego nie daje oczywiście gwarancji na końcowy sukces procesu wytwarzczego. Zwiększa jednak szanse na jego powodzenie, przy założeniu prawidłowego stosowania jego zasad. Przyczynia się do tego lepsze włączenie zamawiających i przyszłych użytkowników systemu w proces budowy oprogramowania. Dyscyplina wymagań angażuje użytkowników systemu w trakcie całego projektu. W dyscyplinach projektowania i implementacji, tworzony jest system, który następnie jest udostępniony użytkownikom do testowania. Testowanie (często połączone z wdrożeniem), pozwala ocenić poprawność działania systemu, ale przede wszystkim zgodność z założeniami i rzeczywistymi potrzebami klienta. Wszelkie problemy odkrywane są po każdej iteracji. Jeżeli aktualny stan oprogramowania nie odpowiada rzeczywistym potrzebom, lub funkcjonalność nie została w pełni zrealizowana, możemy w naturalny sposób – zgodny z założeniami cyklu – przejść do fazy wymagań, a następnie do faz projektowania, implementacji i testowania.

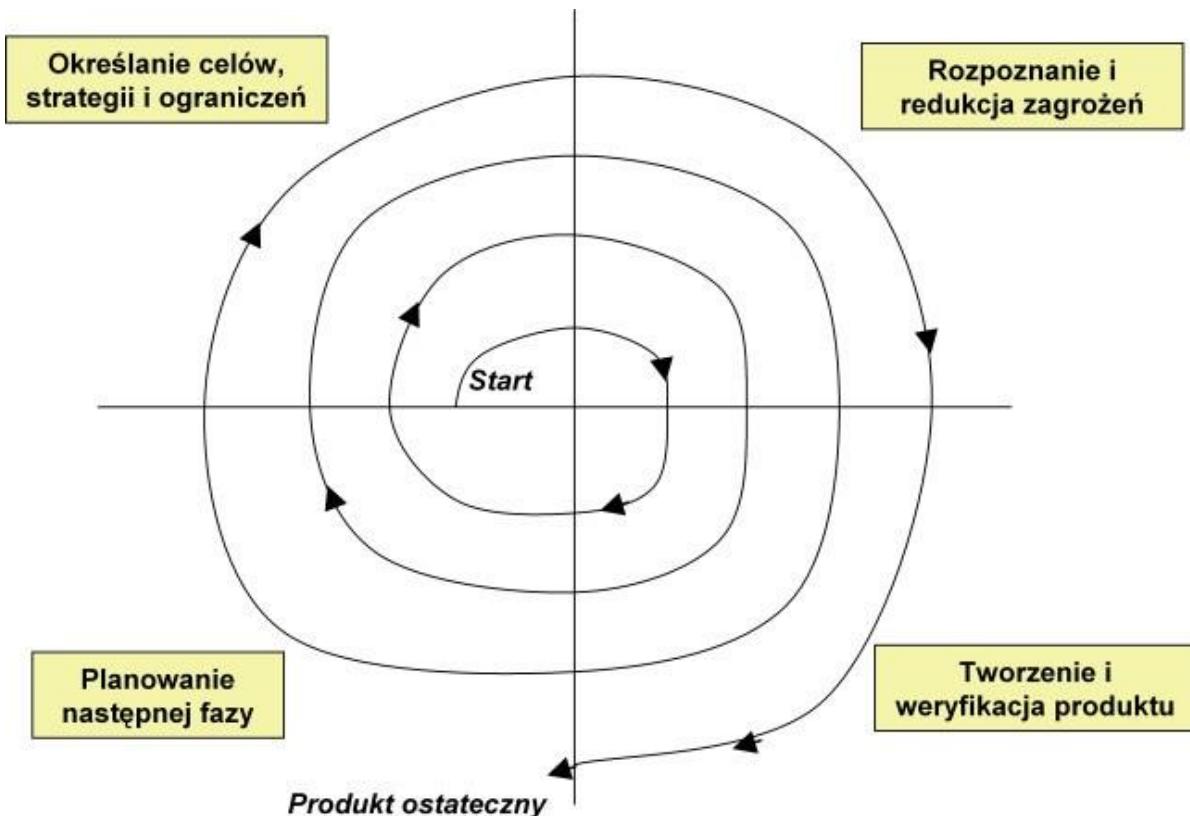
Cykł iteracyjny pozwala na przyrostowe tworzenie systemu oprogramowania. Daje to możliwość ustalenia priorytetów dla określonych funkcjonalności systemu oraz kolejności ich wykonania i wdrożenia. Uproszczeniu ulega także zarządzanie wprowadzaniem zmian w tworzonym oprogramowaniu, które odzwierciedlają zmieniające się ograniczenia środowiska i reguły biznesowe. Produkty poszczególnych faz nie muszą być zamrażane, przez co na koniec cyklu wytwarzania oprogramowania wszystkie specyfikacje i dokumentacje odpowiadają rzeczywistym warunkom.

Kosztem zmniejszenia ryzyka niepowodzenia procesu wytwarzania oprogramowania są bardziej skomplikowane czynności związane z organizacją i zarządzaniem zadaniami w projekcie. Zwykle niesie to ze sobą zwiększenie kosztów wykonania projektu i wydłuża czas od rozpoczęcia prac do ostatecznego ich zakończenia. Ten dodatkowy koszt można jednak porównać do kosztu ubezpieczenia projektu od różnych ryzyk, które występują w cyklu wodospadowym. Jest on zazwyczaj wielokrotnie niższy od kosztów związanych np. z koniecznością dostosowania prawie już gotowego systemu do wymagań klienta.

Iteracyjny cykl wytwarzania oprogramowania najlepiej sprawdza się w projektach budowy średnich i dużych systemów oprogramowania, gdzie nie wszystkie wymagania są dobrze zdefiniowane, a nawet rozpoznane.

Cykl spiralny

Istotnym rozszerzeniem cyklu iteracyjnego jest tzw. cykl spiralny. Główną przyczyną jego powstania była próba powiązania najlepszych cech wodospadowego i iteracyjnego cyklu wytwarzania oprogramowania oraz wzbogacenie go o rozbudowaną analizę ryzyka. Ogólny schemat cyklu spiralnego przedstawia rysunek 2.8. Pokazuje on ścieżkę dla procesu wytwarzania oprogramowania prowadzącą od postawienia problemu (punkt na wykresie na lewo od początku układu współrzędnych) do dostarczenia końcowego produktu. Odbywa się to poprzez realizację czterech zasadniczych etapów ułożonych w formę spirali.



Rysunek 2.8: Spiralny model wytwarzania oprogramowania

Każda pętla spirali podzielona jest na cztery części odpowiadające kolejnym ćwiartkom układu współrzędnych i może reprezentować dowolną fazę procesu twórczego. W tym cyklu nie ma stałych faz takich jak analizowanie, albo projektowanie. Cykl ten obejmuje inne procesy, przedstawione na modelu graficznym jako ćwiartki układu współrzędnych [Som03]:

- Ustalanie celów – definiuje się konkretne cele dla danej fazy procesu wytwarzania oprogramowania, wraz z ograniczeniami nałożonymi produkty danej fazy.
- Rozpoznanie i redukcja zagrożeń – znalezione zagrożenia zostają poddane szczegółowej analizie, po czym podejmuje się kroki zmieniające do redukcji tych zagrożeń.
- Tworzenie i zatwierdzanie – na podstawie oceny zagrożeń wybrana zostaje metoda, która w największym stopniu minimalizuje ryzyko.
- Planowanie – produkt zostaje poddany ocenie i na tej podstawie planowane są następne czynności objęte kolejną pętlą spirali.

W jednej pętli, poświęconej wybranej fazie klasycznego procesu wytwarzczego, można wykorzystać do wytwarzania spodziewanego produktu podejście kaskadowe, a w innej pętli – podejście iteracyjne. Na początku każdej pętli spirali wyznacza się cele, strategie i ograniczenia (pierwsza ćwiartka), po czym znajduje się różne sposoby ich osiągnięcia i realizacji (druga ćwiartka). Należy jednocześnie ocenić każdą opcję względem każdego celu. Dzięki tej ocenie możliwe jest oszacowanie źródeł zagrożeń przedsięwzięcia i wybór najlepszej opcji. Następne czynności polegają na wytwarzaniu i weryfikacji następującego przybliżenia produktu (trzecia ćwiartka) konkretnej fazy. Następnie produkt podlega recenzji (czwarta ćwiartka) i na jej podstawie planowana jest następna pętla spirali. Jeżeli wynik recenzji jest pozytywny, następna pętla będzie dotyczyć następnej fazy wytwarzania oprogramowania. W przypadku negatywnego wyniku, następna faza będzie dotyczyć tej samej fazy i opracowania następnego przybliżenia produktu.

Poprzez jawne potraktowanie zagrożeń, cykl ten znacznie zmniejsza ryzyko niepowodzenia projektu. Należy jednak pamiętać, że cykl spiralny oparty o kontrolę ryzyka, wymaga ciągłego dostosowywania kierowania projektem do warunków napotykanych po drodze od problemu do produktu końcowego. Ścisłe trzymanie się planu może uczynić wybranie spiralnego cyklu największym zagrożeniem dla projektu.

Zadania

Zadanie 1

Zakładamy hipotetyczny duży projekt konstrukcji systemu oprogramowania dla konkretnej instytucji działającej w jednej z dziedzin (patrz sekcja 1.5). Zaproponuj cykl wytwarzania oprogramowania oraz uzasadnij swój wybór.

Zadanie 2

Podaj przykłady projektów, w których korzystniejsze będzie zastosowanie cyklu wodospadowego lub iteracyjnego. Zwróć uwagę na dziedzinę zastosowania oraz rozmiar projektu. Uzasadnij swoją odpowiedź.

Słownik pojęć

Architektura fizyczna

Składnik opisu budowanego systemu oprogramowania przedstawiający elementy fizyczne (np. serwer bazy danych, serwer aplikacyjny, maszyny klienckie, urządzenia mobilne), na których będzie działał budowany system oraz sposób komunikacji między tymi elementami.

Architektura logiczna

Składnik opisu budowanego systemu oprogramowania przedstawiający moduły wykonawcze (komponenty), z jakich będzie się składał budowany system oraz powiązania komunikacyjne (np. interfejsy) między tymi modułami.

Dyscyplina inżynierii oprogramowania

Spójny zestaw czynności, które są powiązane z zasadniczym obszarem działania w ramach projektu konstrukcji oprogramowania.

Iteracja

Zestaw czynności, realizowanych w stosunkowo krótkim czasie (np. 2 tygodnie), podczas którego wykonywany jest fragment systemu realizujący zadaną funkcjonalność. Iteracja kończy się wykonaniem działającego, zwalidowanego (przetestowanego) wydania systemu (wewnętrznego lub zewnętrznego).

Obiekt transferu danych

Struktura danych, która służy do wymiany informacji między komponentami w ramach architektury logicznej systemu.

Opis środowiska (biznesu)

Opis fragmentu świata (rzeczywistości), który jest obszarem działań danej organizacji wraz ze środowiskiem, w jakim ta organizacja działa. Formułowanie takiego opisu polega na stworzeniu precyzyjnego i zrozumiałego modelu, który zawiera wszystkie elementy danego fragmentu rzeczywistości (np. biznesowej) oraz zasady oddziaływanie ich na siebie.

Specyfikacja wymagań

Specyfikacja (dokument, model) przedstawiająca potrzeby zamawiającego w stosunku do tworzonego systemu oprogramowania. W ramach tej specyfikacji formułowana jest ogólna wizja systemu oraz opis jego cech i funkcji na różnym poziomie szczegółowości.

Środowisko implementacyjne (testowe)

Wszystkie elementy fizyczne (sprzęt, sieci) oraz oprogramowanie (np. system operacyjny, bazy danych) zainstalowane w miejscu, gdzie system jest poddawany implementacji oraz testom przed jego przeniesieniem do środowiska produkcyjnego.

Środowisko produkcyjne

Wszystkie elementy fizyczne (sprzęt, sieci) oraz oprogramowanie (np. system operacyjny, bazy danych) zainstalowane w miejscu, gdzie system jest eksploatowany na co dzień przez jego użytkowników.

Test jednostkowy

Procedura testowa, której celem jest sprawdzenie działania tworzonego fragmentu kodu (np. klasy) dla różnych sytuacji (dla różnych danych wejściowych).

Test integracyjny

Procedura testowa, której celem jest sprawdzenia działania zestawu połączonych ze sobą (zintegrowanych) modułów, tworzących system.

Test akceptacyjny

Procedura testowa, której celem jest sprawdzenie działania systemu z punktu widzenia jego akceptacji przez użytkowników.

Co trzeba zapamiętać

Produkty cyklu wytwarzania oprogramowania

Niezależnie od wybranego cyklu życia, każdy projekt wytwarzania oprogramowania obejmuje kilka dyscyplin. Każda z dyscyplin dostarcza różne produkty, które są efektem czynności wykonywanych w ramach dyscyplin. Głównym produktem dyscypliny wymagań jest specyfikacja wymagań. W ramach dyscypliny projektowania powstają m.in. architektura fizyczna i architektura logiczna. Efektem dyscypliny implementacji jest przetestowany (jednostkowo i integracyjnie) kod systemu, a w ramach dyscypliny testowania przeprowadzane są testy akceptacyjne. Ostatecznym produktem wdrożenia systemu jest zainstalowany system w środowisku produkcyjnym.

Cykle wytwarzania oprogramowania

Aby zapanować nad złożonością procesu twórczego, czynności w ramach projektu porządkowane są w ramach cyklu twórczego. W przypadku niewielkich systemów, gdzie wszystkie wymagania są znane i dobrze udokumentowane, może sprawdzić się cykl wodospadowy. W przypadku bardziej rozbudowanych systemów, gdzie nie wszystkie wymagania są rozpoznane, powinien być wykorzystany cykl iteracyjny. Cykl spiralny łączy cechy cyklu wodospadowego i iteracyjnego.

3. Metodyki wytwarzania oprogramowania

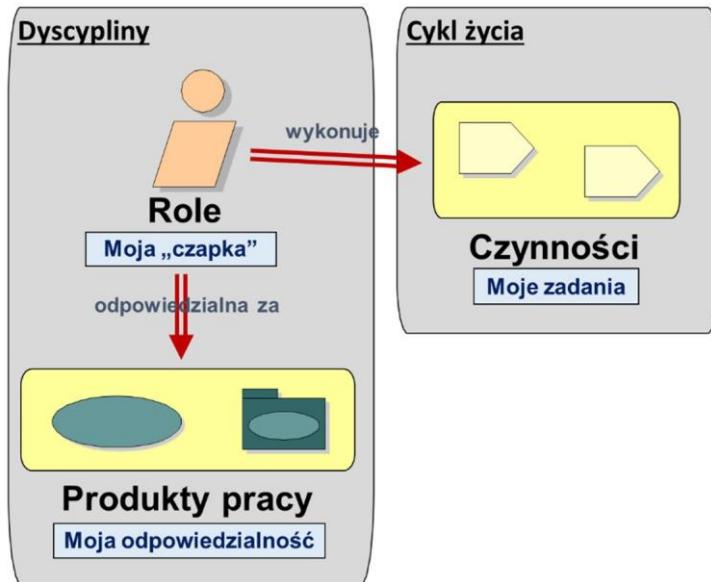
Zastosowanie określonego cyklu twórczego oraz zgrupowanie czynności w dyscypliny może w istotny sposób uporządkować prace w projekcie konstrukcji oprogramowania i obniżyć ryzyko niepowodzenia. Należy jednak podkreślić, że pełna metodyka wytwarzania oprogramowania powinna dostarczać znacznie bardziej dokładnych wskazówek odnośnie całego procesu wytwarzania oprogramowania.

W niniejszym rozdziale przedstawiamy kilka często stosowanych metod wytwórzania oprogramowania. Nasze rozważania oprzemy na definicji **metodyki** jako usystematyzowanym i zorganizowanym zbiorze technik objętych cyklem twórczym, którego produkty wyrażone są za pomocą odpowiedniej notacji. Warto jednak zauważyć, że sam opis metodyki nie stanowi metodyki jako takiej. Metodykę należy bowiem w odpowiedni sposób wdrożyć w organizacji wytwarzającej oprogramowanie oraz w konkretnym projekcie. Działania te obejmują wiele różnych aspektów, takich jak szkolenia, mentoring, wybór i instalacja narzędzi czy tworzenie odpowiedniej kultury pracy. Szczegółowe omówienie procesu wdrażania metodyki wykracza jednak poza zakres niniejszego podręcznika.

Wśród wielu istniejących i stosowanych metod, można wyróżnić dwie główne grupy: metodyki formalne i agilne (zwonne, ang. *agile*). To, która metodyka zostanie wybrana i które jej elementy będą kluczowe dla danego przedsięwzięcia, zależy od specyfiki danego projektu. Przegląd głównych cech obu grup metod, umożliwi zrozumienie idei stosowania metod agilnych i formalnych. Pozwoli to także poznać cechy wspólne i rozłączne obu grup. Warto jednak zauważyć, że większość współcześnie stosowanych metod staje się cykl iteracyjny.

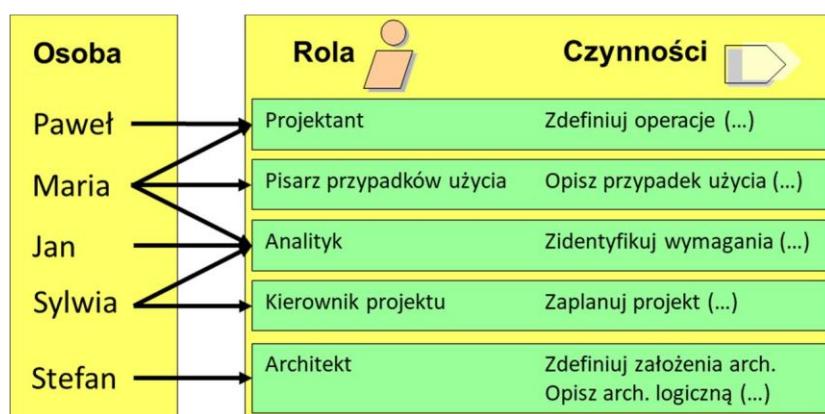
3.1. Co to jest metodyka wytwarzania oprogramowania?

W rozdziale 1 metodyka została przedstawiona jako połączenie trzech elementów: procesu technicznego, notacji oraz technik. W praktyce, zespoły projektowe potrzebują konkretnych wytycznych, które odpowiadają na podstawowe pytanie „jak należy to zrobić?”. Zespół powinien widzieć co należy wykonać, w jaki sposób oraz kto powinien za co odpowiadać. Opis metodyki zazwyczaj zawiera zatem pięć elementów: opis ról, opis czynności, opis produktów pracy, podział na dyscypliny oraz określenie cyklu życia (zazwyczaj ograniczonego do cyklu twórczego). Elementy te zostały zilustrowane na rysunku 3.1.



Rysunek 3.1: Elementy opisu metodyki

Większość metod definiuje **role**, które stanowią abstrakcyjne definicje pewnego zakresu obowiązków (zbioru czynności) wymagających odpowiedniego zakresu kwalifikacji. Role są wypełniane przez wyznaczone osoby lub grupy współpracujących ze sobą osób. Rola może być zatem wypełniana przez kilka (wiele) osób w ramach jednego projektu. Jednocześnie, każdy członek zespołu projektowego może pełnić kilka (wiele) różnych ról. Można powiedzieć, że osoba pełniąca w danej chwili konkretną rolę zakłada odpowiednią „czapkę”, którą zamienia na inną, kiedy wykonuje czynności kolejnej roli. Przykładowy przydział ról (a zatem – również czynności) do konkretnych osób został przedstawiony na rysunku 3.2. Jak widać, role to nie są konkretne osoby. Role stanowią opis tego, jak osoby powinny się zachowywać w ramach procesu tworzenia oprogramowania i za co mają odpowiadać. Większość ról dotyczy organizacji tworzącej oprogramowanie, jednak niektóre role opisują działania przedstawicieli zamawiającego oprogramowanie.



Rysunek 3.2: Przykładowy przydział ról w projekcie

Role są odpowiedzialne za wykonywanie **produkłów pracy**. Produkty pracy stanowią ostateczne lub pośrednie wyniki działań w projekcie, wykorzystywane do podejmowania innych działań. Produktem pracy może być dokument (np. wizja systemu, plan iteracji), model (np. model klas, model komponentów), element modelu (np. klasa, komponent), kod (np. pakiet lub klasa) lub element montażowy (np. plik bazy danych, plik wykonywalny). W celu lepszego zarządzania projektem, produkty pracy oraz wykonyujące je role przydzielamy do dyscyplin. Należy podkreślić, że różne metodyki w różnym stopniu

precyzują produkty pracy. Niezależnie jednak od rodzaju metodyki, każdy dobrze zorganizowany projekt powinien posiadać określone ramy do tworzenia produktów.

Ważnym elementem definicji produktów pracy jest **notacja** używana do ich wytworzenia. Przyjęcie jednolitej notacji jest bardzo istotnym elementem zapewnienia jakości wykonywanych produktów. Umożliwia to łatwiejszą pracę w zespole, wspomaga kreatywność oraz ułatwia porozumienie między różnymi zespołami, częstokroć rozproszonymi geograficznie. Produkty pracy tworzymy używając narzędzi dostosowanych do używanej notacji. Kod oraz elementy montażowe tworzymy w zintegrowanych środowiskach deweloperskich (IDE, ang. Integrated Development Environment) oraz systemach zarządzania wersjami. Modele tworzymy w narzędziach wspierających modelowanie oprogramowania (CASE, Computer Aided Software Engineering). Różnego rodzaju dokumenty i plany tworzymy w narzędziach do edycji tekstu oraz w dedykowanych środowiskach do organizowania pracy w projekcie (często zintegrowanych z narzędziami do zarządzania i wersjonowania kodu). Wiele współczesnych narzędzi funkcjonuje w środowisku chmurowym. W ten sposób możliwa jest praca grupowa w zespołach rozproszonych z możliwością pracy zdalnej.

Produkty pracy są wynikiem **czynności** wykonywanych przez role. Czynności definiują zakres zadań dla odpowiednich ról w projekcie. Czynności korzystają i tworzą produkty pracy. Opis czynności może zawierać różnego rodzaju wskazówki, techniki i instrukcje korzystania z narzędzi. W ramach konkretnej metodyki, czynności są wykonywane w ramach zadanego **cyklu życia**. Czasami definiowany jest odpowiedni proces w postaci diagramu pokazującego przepływ czynności w ramach np. pojedynczej iteracji.

3.2. Metodyki zwinne (agile)

Metodyki zwinne bazują na manifeście zwinności przedstawionym w rozdziale „Wprowadzenie do inżynierii oprogramowania”. Wynikają z tego podstawowe cechy metodyk zwinnych: iteracyjność oraz koncentracja na technikach organizacji zespołu i współpracy z klientem. Bardzo istotna zasadą metodyk zwinnych jest częste dostarczanie działającego oprogramowania. Odbywa się to w stosunkowo krótkich cyklach (iteracjach) trwających od około 2 tygodni do około 2-3 miesięcy (preferowane są cykle krótsze). Dzięki temu na pierwszy plan wysuwa się zasada zapewnienia maksymalnej satysfakcji klienta. Klient ma możliwość ciągłej (w cyklach kilkutygodniowych) weryfikacji spełnienia swoich wymagań w stosunku do budowanego systemu. Klient jest również zachęcany do zgłaszania zmian, nawet na późnym etapie projektu. Dzięki takiemu podejściu zwiększa się szansa na powstanie systemu dostosowanego do rzeczywistych potrzeb zamawiającego oraz o cechach innowacyjnych, zwiększających przewagę konkurencyjną.

Istotnym elementem składowym metodyk zwinnych jest podkreślanie czynnika ludzkiego. Techniki zawarte w metodykach zwinnych pomagają stworzyć spójny zespół oparty na dobrze zmotywowanych jednostkach. Podstawą jest stworzenie dobrego środowiska pracy, zapewnienie niezbędnej pomocy metodycznej oraz dobra komunikacja. Preferowane są bezpośrednie rozmowy, jako najbardziej efektywna metoda komunikacji. Dotyczy to zarówno członków zespołu deweloperskiego, jak i przedstawicieli klienta (osób biznesowych). Kluczowa jest tutaj bezpośrednia i codzienna praca z klientem przez cały czas trwania projektu. Dobra komunikacja sprzyja również doskonaleniu procesu. Zespół w metodykach zwinnych dokonuje systematycznej oceny efektywności swojego działania i w regularnych odstępach przeprowadza działania dostosowawcze.

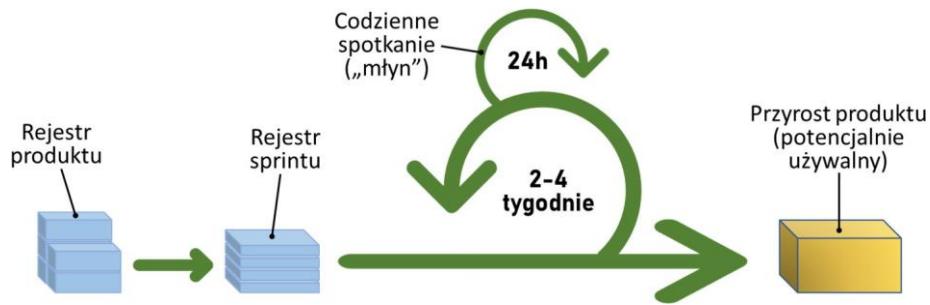
Główną miarą postępu w podejściach zwinnych jest działające oprogramowanie. Mierzono są przede wszystkim jednostki funkcjonalności, które zostały w pełni zrealizowane i zaakceptowane przez klienta. Jednocześnie, dużą wagę przykłada się do zrównoważonego rozwoju oprogramowania. Tempo prac

powinno być stałe, o co powinni dbać wszyscy uczestnicy projektu – deweloperzy, reprezentanci zamawiającego, a przede wszystkim – sponsorzy. Koncentracja na systematycznym dostarczaniu działającego systemu (działającego kodu) nie oznacza jednak braku działań w obszarze projektowania oprogramowania. Metodyki zwinne podkreślają, że dobrze zorganizowany zespół w naturalny sposób dąży do tworzenia dobrze zaprojektowanych systemów. Taki zespół ciągle zwraca uwagę na doskonałość techniczną, co przejawia się kładzeniem nacisku na tworzenie projektów architektonicznych oprogramowania. Takie projekty są ciągle doskonalone, czemu sprzyja ciągła kontrola jakości oraz ciągły po-miar efektywności prac deweloperskich. Jednocześnie, nacisk położony jest na tworzenie jak najprost-szych rozwiązań technicznych, redukujących ilość pracy związanej z implementacją systemu.

Istnieje wiele metodyk zwinnych. Z najważniejszych warto wymienić Scrum, Extreme Programming, Crystal, Lean Development i Kanban. Metodyka Scrum została opracowana przez Kena Schwabera i Jeffa Sunderlanda w okolicach roku 1995. Oficjalna definicja metodyki („Przewodnik po Scrumie”) jest bardzo zwięzła. Doczekała się ona kilku aktualizacji (ostatnia w 2020 roku). W podobnym czasie co Scrum została również opracowana metodyka Extreme Programming (XP). Głównym twórcą tej metodyki jest Kent Beck, który w 1999 roku opublikował pierwszą książkę opisującą jej reguły. Nieoficjalną specyfikację metodyki XP można znaleźć na stronie extremeprogramming.org. Również w połowie lat 1990 powstała metodyka Crystal, stworzona przez Alistaira Cockburna. Ważną cechą tej metodyki (a właściwie – rodziny metodyk) jest skalowalność – kolejne „poziomy” metodyki są dostosowane do coraz większych zespołów, realizujących coraz większe systemu oprogramowania. Wraz ze wzrostem złożoności zespołu rośnie również poziom formalizacji wykonywanych w ramach metodyki czynności.

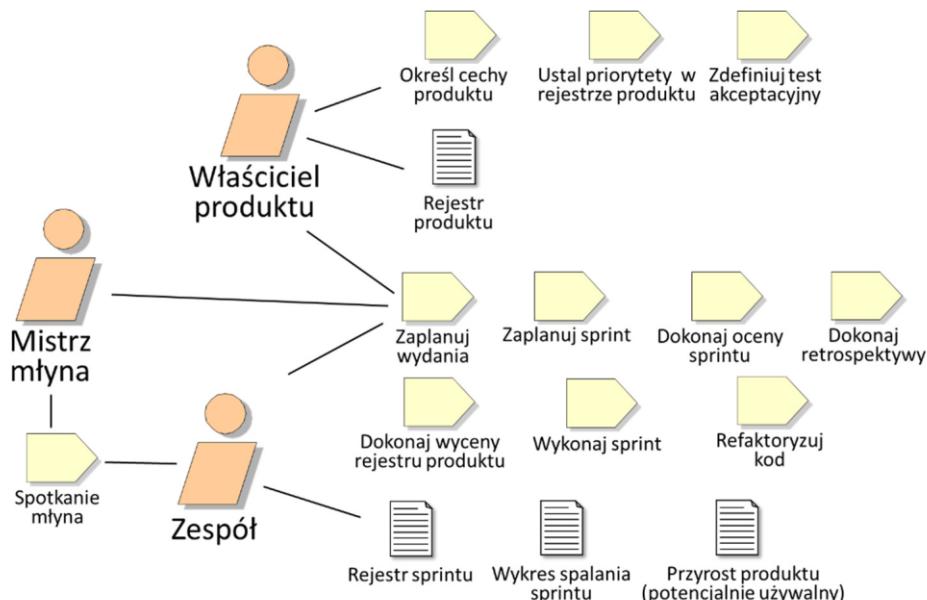
Nieco później niż Scrum i XP została sformułowana metodyka Lean Development (Lean). Została ona przedstawiona w 2003 roku przez Mary i Toma Poppendieck jako rozwinięcie i dostosowanie do warunków projektów software'owych metod wytwórczych stosowanych w firmie Toyota. Metodyka stosowana do produkcji samochodów Toyota została opracowana przez Taiichi Ohno i zyskała nazwę Kanban (tablica, billboard). Tej samej nazwy użył David Anderson, który w latach 2004-2010 dokonał adaptacji Kanbana do warunków projektów produkcji oprogramowania. Należy podkreślić, że metodyki Lean i Kanban, chociaż wymieniane w gronie metodyk zwinnych, nie stanowią metodyk w sensie definicji sformułowanej na początku niniejszego rozdziału. Stanowią one raczej zbiór praktyk poprawiających organizację pracy zespołu deweloperskiego.

Aby przybliżyć zasady rządzące metodykami zwinymi opiszemy bliżej dwie metodyki – Scrum i XP. Rysunek 3.3 przedstawia cykl wytwórczy oraz podstawowe produkty metodyki **Scrum** („młyn”, nazwa pochodzi z terminologii gry w rugby). Pojedyncza iteracja (tzw. sprint) trwa zazwyczaj kilka tygodni. W ramach sprintu postęp prac kontrolowany jest codziennie. Główną techniką kontroli postępów jest tzw. codzienne spotkanie młyna. Na początku sprintu definiowany jest tzw. rejestr sprintu (ang. Sprint Backlog), który jest na bieżąco aktualizowany i służy do kontroli wykonania zaplanowanych zadań. Efektem każdego sprintu jest przyrost produktu (ang. Product Increment), który może być w określonym zakresie używany przez klienta. Kolejne sprints (a tym samym – przyrosty systemu) planowane są w ramach tzw. rejestrzu produktu (ang. Product Backlog).



Rysunek 3.3: Cykl wytwórczy metodyki Scrum

Rysunek 3.4 pokazuje podstawowe role, produkty pracy i czynności metodyki Scrum. Jak można zauważać, zakres definicji metodyki jest dosyć ograniczony i głównie koncentruje się na organizacji pracy zespołu deweloperskiego oraz planowaniu zadań. Metoda definiuje 3 role – mistrz młyna (ang. Scrum Master), właściciel produktu (ang. Product Owner) oraz zespół deweloperski. Cały zespół w projekcie prowadzonym metodą Scrum powinien liczyć kilka osób (do około 10). W jego skład wchodzi jeden mistrz młyna i jeden właściciel produktu. Metoda nie wyróżnia bardziej szczegółowych ról w ramach zespołu deweloperskiego.



Rysunek 3.4: Role w metodyce Scrum

Zespół deweloperski powinien składać się z osób o różnych, uzupełniających się kompetencjach. Metoda nie określa tych kompetencji, ale można uznać, że są to typowe kompetencje obejmujące wszystkie dyscypliny techniczne inżynierii oprogramowania (wymagania, projektowanie, implementacja, testowanie, wdrożenie). Zespół powinien sam się organizować i Scrum nie narzuca żadnych konkretnych ram organizacyjnych zespołu. Podstawowym celem zespołu jest zapewnienie odpowiedniej jakości kolejnych przyrostów produktu (włącznie z przyrostem końcowym). Podstawowe czynności wykonywane przez zespół deweloperski to zaplanowanie sprintu oraz wykonanie sprintu.

Planowanie sprintu polega na ustaleniu celu sprintu oraz szczegółowej listy zadań do wykonania w ramach sprintu, przydzielenie tych zadań do wykonania członkom zespołu oraz określenie harmonogramu ich wykonania. Rezultatem planowania sprintu jest **rejestr sprintu**. Rejestr ten stanowi ciągle aktualizowaną „tablicę” z zadaniemi do wykonania oraz ich statusem. Zadania są definiowane w sposób szczegółowy i powinny dotyczyć wszystkich dyscyplin inżynierii oprogramowania (od wymagań do

wdrożenia). Zadania powinny prowadzić do wykonania celu sprintu, czyli do zbioru cech produktu, które powinny zostać zrealizowane w ramach sprintu. Cechy produktu są wybierane do wykonania z rejestru produktowego.

Ciekawą techniką służącą do szacowania pracochłonności i przydzielu cech produktu do danego sprintu jest tzw. **poker scrumowy** (ang. Scrum Poker). Technika ta polega na dokonaniu wyceny cech systemu w formie sesji pracy grupowej. Zespół deweloperski wstępnie wybiera zestaw cech do oceny. Następnie, stosowany jest specjalny zestaw kart. Karty w zestawie posiadają odpowiednie liczby, które odpowiadają pracochłonności wykonania danej cechy. Często stosowane są ciągi liczbowe, które nie są liniowe (np. ciąg Fibonacciego – 1, 2, 3, 5, 8, 13, 21, 34, 55, ...). Wynika to potrzeby podkreślenia, że liczby są szacunkowe, szczególnie podczas określania pracochłonności wykonania dla bardziej złożonych cech systemu. Podczas sesji pokera scrumowego, poszczególni członkowie zespołu dokonują szacowania pracochłonności danej cechy poprzez wybranie jednej z kart. Następnie karty są jednocześnie odkrywane i odbywa się dyskusja. W trakcie dyskusji oceniane są wartości skrajne i podejmowana decyzja w drodze konsensusu. Efektem pokera scrumowego jest zestaw cech systemu z przypisanymi punktami. Na podstawie liczby punktów i wcześniejszych doświadczeń zespół wybiera cechy, które jest w stanie zrealizować w kolejnej iteracji (sprincie).

Należy podkreślić, że planowanie sprintu jest czynnością ciągłą. Rejestr sprintu jest aktualizowany na bieżąco w miarę dokonywania oceny postępów prac. Istotnym narzędziem do kontroli postępów jest tzw. **wykres spalania** sprintu (ang. sprint burndown chart). Wykres ten pokazuje w sposób graficzny liczbę zadań pozostających do wykonania. W ten sposób, zespół na bieżąco może śledzić tempo prac i oceniać możliwość ich wykonania w ramach sprintu.

O ile rejestr sprintu służy do planowania jednej iteracji, o tyle **rejestr produktu** służy do planowania całego projektu. Za rejestr produktu odpowiada **właściciel produktu**. Określa on cechy produktu, ustala ich priorytety, a także planuje wydania (przyrosty produktu gotowe do użycia przez klienta w rzeczywistym środowisku). Metodyka Scrum nie definiuje w sposób jednoznaczny, czym są cechy produktu. Często przyjmuje się, że cechami produktu są historie użytkownika (ang. User Story), które są elementem metodyki Extreme Programming. Cechami produktu mogą też być przypadki użycia pochodzące z metodyki Unified Process (RUP i OpenUP). Ogólnie, przyjmuje się, że sprinty w metodyce Scrum powinny być sterowane cechami, które odpowiadają wymaganiom funkcjonalnym (funkcjonalności systemu). Cechami uzupełniającymi są cechy jakościowe (np. niezawodność, wydajność, użyteczność). Należy jednak podkreślić, że cechy jakościowe zazwyczaj przebiegają przez cały produkt (system oprogramowania), zatem nie jest możliwe przydzielenie ich realizacji do konkretnej iteracji (sprintu).

Przedstawione powyżej czynności związane z planowaniem i wykonywaniem sprintów uzupełniają czynności dotyczące zapewnienia jakości. Właściciel produktu specyfikuje testy akceptacyjne, które mają na celu zdefiniowanie kryteriów ukończenia sprintu. Zapewnienie jakości produktu przez zespół deweloperski jest oparte na dążeniu do zakończenia wszystkich testów akceptacyjnych w sposób pozytywny. Innym sposobem zapewnienia jakości systemu przez zespół jest dokonywanie refaktoryzacji kodu. Czynność ta pochodzi z metodyki Extreme Programming i ma na celu poprawę jakości struktury kodu, a przez to również – serwisowalności całego systemu.

Rolą kluczową w zapewnieniu jakości jest **mistrz młyna**. Jego zadaniem jest pomóc zespołowi w przestrzeganiu reguł metodyki. Mistrz młyna powinien stwarzać zespołowi warunki odpowiednie do poprawy efektywności pracy oraz poprawy jakości budowanego systemu. Mistrz młyna uczestniczy we wszystkich czynnościach zespołu deweloperskiego i właściciela produktu. Pomaga w stosowaniu odpowiednich technik pracy z zespołem, planowaniu sprintów oraz całego projektu.

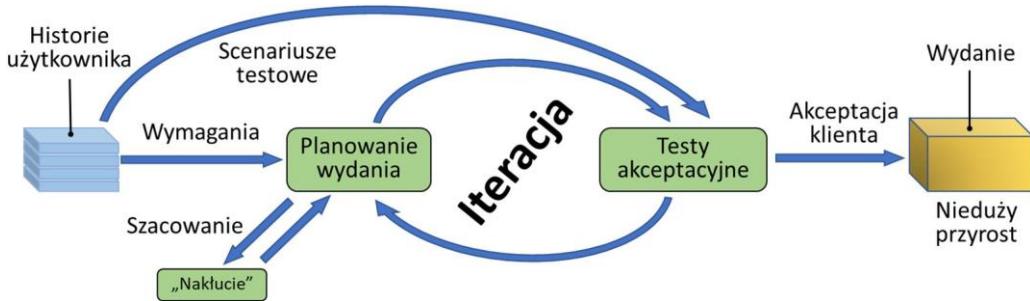
Podstawowymi czynnościami zapewnienia jakości pracy zespołu są **spotkanie młyna** oraz **dokonanie retrospektwy**. Spotkanie młyna powinno się odbywać codziennie. Jego celem jest przedstawienie postępu prac w dniu poprzednim oraz planów na dzień bieżący. Takie codzienne spotkanie umożliwia lepszą identyfikację problemów bieżących oraz umożliwia ciągłą ocenę tempa prac poszczególnych deweloperów. Co jest istotne, spotkanie powinno odbywać się na stojąco i trwać nie więcej niż kilkanaście minut. Podczas spotkania młyna nie dyskutuje się rozwiązań, a jedynie przedstawia problemy i ustala zadania do wykonania. Formuła szybkiego spotkania na stojąco ma sprzyjać zwięzości wypowiedzi i podejmowaniu konkretnych decyzji.

Rozwinięciem codziennych spotkań młyna są spotkania retrospektwy. Spotkania takie organizuje się na koniec każdego sprintu. Ich celem jest ustalenie problemów w funkcjonowaniu zespołu w ramach danego projektu oraz wypracowanie rozwiązań. Członkowie zespołu wspólnie z mistrzem młyna zastanawiają się jakie elementy procesu twórczego działają prawidłowo, a jakie sprawiają problemy. W efekcie zespół podejmuje decyzje dotyczące tego, jakich czynności należy zaprzestać, jakie czynności należy zacząć wykonywać, a jakie kontynuować bez zmian lub z odpowiednimi zmianami.

Oprócz retrospektwy, na koniec każdego sprintu dokonuje się **oceny sprintu**. Podczas takiego spotkania zespół przedstawia postęp prac – co zostało osiągnięte w trakcie ostatnich kilku tygodni projektu. Często, taka ocena polega na przeprowadzeniu demonstracji nowych funkcjonalności budowanego systemu. Kluczowa jest tutaj obecność właściciela produktu, który jest w stanie ocenić jakość produktu (oprogramowania) z punktu widzenia potrzeb zamawiającego. Ocena sprintu dokonywana jest na podstawie oceny realizacji celów danego sprintu. W szczególności, oceniane jest spełnienie realizacji cech systemu przydzielonych do danego sprintu w rejestrze sprintu. Ważne jest, aby spotkanie oceny sprintu nie było zbytnio sformalizowane. Nie powinno ono odciągać zespołu od podstawowych działań deweloperskich., a jedynie umożliwić szybką ocenę postępów prac w projekcie oraz ocenę jakości dotychczas zbudowanego systemu z punktu widzenia jego przyszłych użytkowników.

Na koniec omówienia metodyki Scrum warto wspomnieć o możliwości skalowania metodyki. Jej podstawowa wersja jest przeznaczona dla niewielkich zespołów (do ok. 10 osób). W przypadku większych projektów rozwiązaniem jest zastosowanie metody „Scrum of Scrums” (młyn młynów). Polega ona na budowie większego zespołu jako zbioru kilku zespołów scrumowych. Każdy zespół składowy rządzi się regułami metodyki Scrum i odpowiada za wybrany fragment funkcjonalności całego systemu. Na poziomie całego projektu organizowane są spotkania „młyna młynów”, które mają na celu koordynację prac poszczególnych zespołów. Takie spotkania odbywają się w formule podobnej do codziennego spotkania młyna. Odbywają się one jednak co kilka dni (do tygodnia) i angażują jedynie przedstawicieli poszczególnych zespołów.

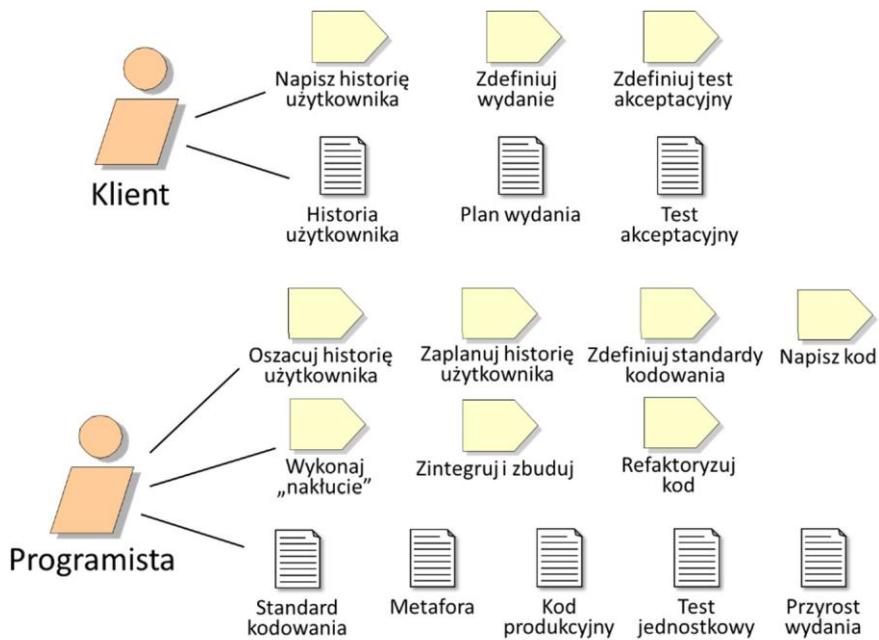
Metoda **Extreme Programming (XP)** oferuje podobny, iteracyjny cykl twórczy jak metoda Scrum. Nieco inaczej są tutaj zdefiniowane role oraz zdefiniowane są konkretne czynności techniczne. Iteracja w metodzie XP jest sterowana wymaganiami oraz testami, co ilustruje rysunek 3.5. Wymagania formułowane są w postaci historii użytkownika, które są oryginalną notacją wprowadzoną przez XP. Zestaw historii użytkownika służy do planowania poszczególnych wydań (przyrostów) systemu, które stanowią rezultaty kolejnych iteracji. Jednocześnie z wymaganiami tworzone są testy akceptacyjne, które mają sprawdzić spełnienie tych wymagań. Jest to zgodne z jedną z podstawowych cech metodyki XP, jaką jest wytwarzanie sterowane testami (ang. Test Driven Development).



Rysunek 3.5: Cykl wytwórczy metodyki Extreme Programming

Metodyka XP definiuje dwie role podstawowe: Klient i Programista, oraz trzy role uzupełniające: Tester, Tropiciel i Trener. Ważną zasadą jest praca całego zespołu (wszystkich ról) we **wspólnej przestrzeni pracy**, ułatwiającej bieżącą komunikację. Przy tym bardzo istotna jest możliwość ciągłego kontaktu z klientem. Klient musi być ciągle na miejscu, czyli powinien mieć swoje miejsce pracy we wspomnianej wspólnej przestrzeni. Wszyscy deweloperzy (programiści, zgodnie z terminologią XP) powinni mieć natychmiastowy dostęp do klienta w celu ciągłej analizy wymaganej funkcjonalności oraz weryfikacji powstającego systemu. Komunikacji i mierzeniu postępu prac sprzyjają codzienne **spotkania na stojąco**, podobne w swej formule do codziennych spotkań młyna w metodyce Scrum. Ważne jest, że zespół powinien utrzymywać zrównoważone tempo prac, nie podlegające nagłym zrywom. W szczególności, metodyka XP podkreśla niedopuszczalność powstawania nadgodzin, które negatywnie wpływają na wydajność pracy przemęczonych programistów.

Rola **Klienta** odpowiada w przybliżeniu roli właściciela produktu w metodyce Scrum. Podstawowe zadania oraz produkty produkowane przez klienta przedstawione są na rysunku 3.6. Klient jest przede wszystkim odpowiedzialny za pisanie **historii użytkownika** (ang. User Story). Historie użytkownika reprezentują niewielkie fragmenty funkcjonalności budowanego systemu, dostarczające użytkownikowi konkretnej wartości biznesowej. Warto podkreślić, że historie użytkownika opisują system z punktu widzenia użytkownika. Nie dotyczą one pojedynczych komponentów architektury systemu, a ich realizacja przebiega przez wszystkie warstwy (od interfejsu użytkownika do bazy danych). Szczegółowo historie użytkownika omawiamy w Module III (Podstawy inżynierii wymagań i projektowania oprogramowania).



Rysunek 3.6: Role podstawowe w metodyce Extreme Programming

Zestaw historii użytkownika wybrany do realizacji w danej iteracji (lub kilku iteracji) stanowi **plan wydania** (ang. Release Plan). Istotną zasadą podczas planowania wydań jest potrzeba dostarczenia klientowi istotnej wartości biznesowej odpowiednio wcześnie w trakcie projektu. Jednocześnie, ważne jest przydzielenie do wczesnych wydań systemu tych historii użytkownika, których wykonanie stanowi istotny problem techniczny. Planowanie wydań jest zatem ważnym elementem zarządzania ryzykiem, zgodnie z zasadą pokonywania najtrudniejszych i najważniejszych problemów odpowiednio wcześnie.

Dla historii użytkownika przydzielonych do danego wydania klient przygotowuje zestaw **testów akceptacyjnych** (inaczej – testów klienta, ang. Customer Tests). Mogą one np. stanowić szczegółowe scenariusze dla historii użytkownika wraz z określeniem zakresu danych wejściowych oraz spodziewanych rezultatów. Warto podkreślić, że testy akceptacyjne stanowią de-facto element wymagań klienta. Umożliwiają one porozumienie między klientem a programistami i testerami odnośnie kryteriów pozytywnego odebrania danego wydania systemu.

Za realizację historii użytkownika odpowiada rola **Programisty**. Pierwszą czynnością wykonywaną przez programistów w ramach iteracji jest **oszacowanie historii użytkownika**. Do oszacowania można wykorzystać różne techniki, w tym np. sesje pokera scumowego. Czasami jednak oszacowanie złożoności technologicznej wykonania historii użytkownika może być trudne. W takiej sytuacji przydaje się technika tzw. **nakłucia** (ang. Spike). Polega ona szybkim rozpoznaniu dostępnych możliwości technicznych poprzez wykonanie fragmentu systemu w bardzo wąskim zakresie. Programiści bardzo szybko budują komponenty o ograniczonej funkcjonalności, stosując np. różne biblioteki, języki czy wzorce. Dzięki temu są w stanie wybrać odpowiednie rozwiązania, jednocześnie wyrzucając te, które się nie sprawdziły.

Zasadniczym zadaniem programistów jest oczywiście **pisanie kodu**. Efektem jest **kod produkcyjny** (ang. Production Code), który jest najważniejszym produktem pracy. Metodyka XP proponuje w tym obszarze kilka ciekawych technik i zasad. Podstawową zasadą jest **kolektywna własność kodu**. Dany fragment kodu (klasa, pakiet) nie ma zatem jednego autora (osoby odpowiedzialnej). Kod w dowolnym miejscu może modyfikować każdy. Z tą zasadą wiąże się konieczność opracowania wspólnych dla całego zespołu **standardów kodowania**. Zasada współprawy kodu wspierana jest przez technikę **programowania parami**. Zgodnie z tą techniką cały kod tworzony jest w parach. Każda para składa się z

osoby piszącej kod oraz osoby sprawdzającej kod. Ma to na celu zwiększenie jakości kodu i w efekcie – większą produktywność (szybsze powstawanie działającego poprawnie kodu). Pary tworzone są na zaledwie kilka godzin. W odpowiednich momentach programiści dobierają się w inne pary i praca jest kontynuowana. Oczywiście, taka technika jest możliwa jedynie dzięki pracy całego zespołu programistów w jednej przestrzeni roboczej.

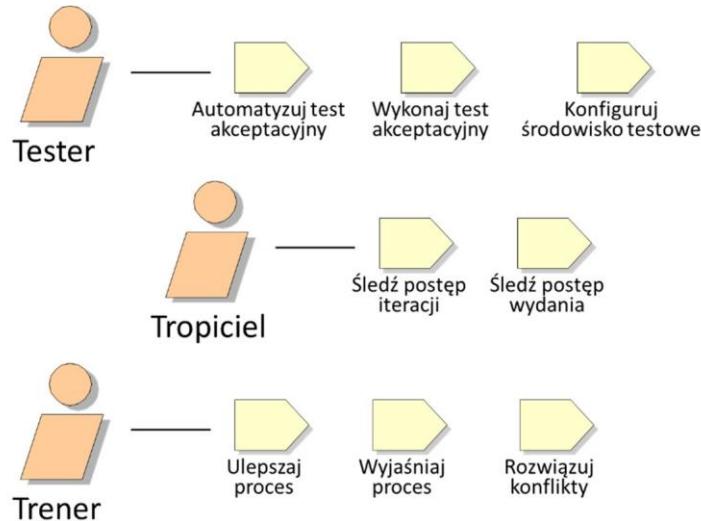
Wbrew swojej nazwie, metodyka XP nie ogranicza zadań programisty jedynie do pisania kodu. Podkreśla ona bowiem wagę technik projektowania kodu. Podstawową zasadą w tym obszarze jest **prosty projekt**. Zgodnie z nią, kod powinien spełniać kilka kryteriów jakościowych – łatwość testowania, zrozumiałość, łatwość przeglądania oraz łatwość wyjaśnienia. Extreme Programming nie definiuje notacji dla zapisu planów projektowych systemu. Zaleca natomiast tworzenie tzw. **metafor** (ang. Metaphor). Metafory stanowią wspólny „słownik” opisujący najważniejsze składniki (komponenty) systemu oraz zależności między nimi. Taki słownik stanowi analogię architektury systemu i pozwala zespołowi na dyskusję nad strukturą systemu w sposób zrozumiały dla wszystkich jego członków. Dyskusje może również wspomóc technika kart CRC (klasy, odpowiedzialność, współpracownicy, ang. Class Responsibilities, Collaborators). Polega ona na stosowaniu kart (np. wielkości kart katalogowych). Na każdej takiej karcie zespół zapisuje nazwę klasy, jej zakres odpowiedzialności, oraz inne klasy, z którymi dana klasa współpracuje. Karty takie stanowią bardzo dobry rekwizyt w sesjach pracy grupowej stanowiąc ułatwienie w zaprojektowaniu funkcjonalności historii użytkownika w technologii obiektowej. Członkowie zespołu mogą np. odgrywać scenariusze działania systemu wskazując na kolejne obiekty klas wykonujące odpowiednie działania zgodnie ze swoją odpowiedzialnością. Warto zaznaczyć, że ponieważ XP nie definiuje notacji dla zapisania metafory (architektury systemu), to zespół może wykorzystać różne znane sobie notacje. Współcześnie, najbardziej rozpowszechnioną notacją dla systemów obiektowych jest język UML. O modelowaniu obiektowym oraz języku UML mówimy w Module II (Modelowanie oprogramowania).

Zespół programistów w swojej pracy dąży do zrealizowania systemu spełniającego wszystkie zadane testy akceptacyjne. Droga do tego jest m.in. tworzenie automatycznych **testów jednostkowych**. Test jednostkowy umożliwia programistom sprawdzenie działania określonego fragmentu kodu (klasy, komponentu). Po przetestowaniu poszczególnych składników systemu, zespół dokonuje **integracji i budowy** (ang. Integrate and Build) systemu wykonywalnego. Proces ten powinien być maksymalnie zautomatyzowany przez zastosowanie odpowiednich narzędzi. Jest to o tyle ważne, że metodyka XP zaleca stosowanie **ciągłej integracji** (ang. Continuous Integration) przez pary programistów. Integracja kodu do wspólnego repozytorium powinna odbywać się w cyklach jednodniowych albo nawet kilkugodzinnych. Sprzyja to wczesnemu wykrywaniu niezgodności, co jest szczególnie ważne w warunkach kolektywnej własności kodu.

Efektem pracy programistów w ramach jednej iteracji jest przyrost wydania. Podlega on testom akceptacyjnym, które wykonywane są przez rolę **Testera**. Testy akceptacyjne są zdefiniowane przez klienta, ale tester musi te testy odpowiednio zaimplementować. Zasadą metodyki XP jest **automatyzacja testów akceptacyjnych**. Polega to na zastosowaniu odpowiednio skonfigurowanego środowiska testowego opartego na wybranym narzędziu do testowania metodą czarnej skrzynki. W ramach takiego należy stworzyć odpowiednie skrypty testowe, które mogą być uruchamiane w sposób automatyczny (bez udziału testerów). Automatyzacja ułatwia przeprowadzenie tzw. testów regresyjnych, tzn. testów powtarzanych w kolejnych iteracjach. O metodach testowania mówimy w Module IV (Podstawy rozwoju i eksploatacji oprogramowania).

Praca całego zespołu wspierana jest przez dwie role pomocnicze. Zadaniem **Tropiciela** (ang. Tracker) jest śledzenie postępów prac zespołu. Tropiciel może np. zarządzać planem projektu i kontrolować realizację zadań technicznych, których efektem są działające historie użytkownika. Docelowo, tropiciel

komunikuje zespołowi jego postępy w budowie kolejnego wydania systemu. Zadaniem **Trenera** jest dbanie o jakość pracy zespołu. Jest to rola podobna do roli mistrza młyna w metodyce Scrum, zatem nie będziemy jej tutaj szerzej omawiać. Zadania opisanych wyżej ról pomocniczych zostały zebrane i przedstawione na rysunku 3.7.



Rysunek 3.7: Role kontrolne i zarządcze w metodyce Extreme Programming

3.3. Metodyki sformalizowane

Metodyki zwinne sprawdzają się w projektach realizowanych przez małe zespoły oraz realizowanych dla klientów akceptujących zasady zwinności. W szczególności, metodyki zwinne trudno jest stosować w projektach zamawianych przez instytucje stosujące formalne zasady zamawiania oprogramowania. W takiej sytuacji zalecane lub wręcz konieczne jest zastosowanie metodyki sformalizowanej. Nie oznacza to jednak, że musimy stosować sztywny cykl wytwarzczy, oparty na zasadzie wodospadu. Współczesne metodyki sformalizowane również stosują cykl iteracyjny i mogą być zastosowane do projektów o różnej skali.

Wybór metodyki stosowanej w danym projekcie może zależeć od wymagań zamawiającego. Przykładowo, w latach 1980 metodyką obowiązkową w zamówieniach publicznych w Wielkiej Brytanii była metodyka SSADM (Structured Systems Analysis and Design Method). Była to metodyka oparta na cyklu wodospadowym oraz stosowała zasady projektowania systemów budowanych w paradygmacie strukturalnym (w odróżnieniu od paradygmatu obiektowego). Podobna sytuacja dotyczy systemów zamawianych przez Departament Obrony Stanów Zjednoczonych. Wymagał on stosowania metodyk zapisanych jako odpowiednie standardy – DOD-STD-2167A oraz MIL-STD-498 (wersja cywilna: EJA J-STD-016). Metodyki te pokazują ewolucję od cyklu wodospadowego do stopniowego przechodzenia na cykl iteracyjny. Oparte one są na definicji wzorów dokumentów i sposobach ich stosowania w projektach konstrukcji oprogramowania. W Niemczech popularną metodyką stosowaną w zamówieniach dla rządu RFN był metodyka V-Modell (wersje 1997 i 2004). Oparta ona była na omawianym w poprzednich rozdziałach – cyklu V, podobnym do cyklu wodospadowego. Podstawową cechą metodyki V-Modell było podkreślenie zależności między czynnościami wytwarzyczymi i walidacyjnymi.

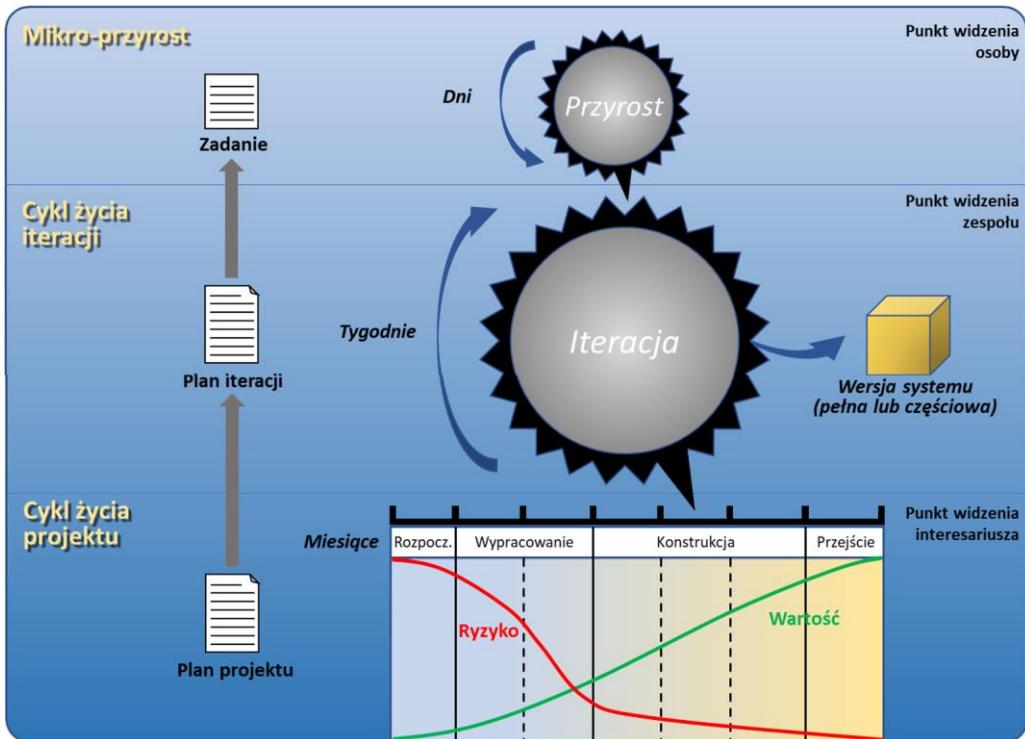
Oprócz metodyk stosowanych przez jednostki publiczne powstało wiele metodyk proponowanych i stosowanych przez poszczególne firmy lub konsorcja. Warto tutaj wymienić:

- Application Lifecycle Management (stosowana przez nieistniejącą już firmę Borland),
- Microsoft Solution Framework (MSF, stosowana przez firmę Microsoft),
- Select Perspective (stosowana przez firmę Select Business Solutions),
- Eiffel Development Framework (oparta na dosyć ciekawym języku programowania Eiffel),
- OPEN (Object-Oriented Process Environment and Notation, definiowana przez konsorcjum OPEN),
- Essence (metodyka inicjatywy SEMAT – Software Engineering Method and Theory).

W niniejszej sekcji skoncentrujemy się na przedstawieniu dwóch, bodaj najpopularniejszych metodyk sformalizowanych. Obydwie pochodzą z tej samej rodziny tzw. **procesów zunifikowanych** (ang. Unified Process). Oryginalną metodyką jest metodyka Rational Unified Process (RUP), opracowana przez firmę Rational, a następnie przejęta przez firmę IBM. Twórcami metodyki są twórcy wspomnianego wyżej języka UML – Ivar Jacobson, James Rumbaugh i Grady Booch. Dlatego też metodyka RUP, w znacznej swojej części opiera produkty pracy na notacji tego języka.

RUP jest metodyką bardzo obszerną i możliwą do zastosowania w bardzo szerokim zakresie dziedzin oraz rozmiarów projektów. Ta uniwersalność metodyki czyni ją jednocześnie trudną do wdrożenia w konkretnym projekcie. Wymagane są stosunkowo pracochnonne czynności związane z adaptacją metodyki do konkretnego środowiska projektowego. Dlatego też opracowano uproszczoną jej wersję – **OpenUP**, zbliżoną zasadami do metodyk zwinnych. Nazwa metodyki nawiązuje do otwartości opisu, gdyż jest ona dostępna w domenie publicznej w odróżnieniu od metodyki RUP, która była dostarczana jako produkt.

Cykł wytwarzczy metodyki OpenUP przedstawia rysunek 3.8. Jak widać, jest on bardzo podobny do cykli wytwarzczych metodyk Scrum i XP. Podstawową jednostką planowania w projekcie jest iteracja. Projekt dzielimy na kilka-kilkanaście iteracji trwających po kilka tygodni. Podział na iteracji dokumentowany jest w postaci **planu projektu**. Dodatkowo, wyróżniane są fazy projektu, które odpowiednio profilują (rozniają) przebieg iteracji w zależności od czasu. OpenUP (oraz RUP) wyróżnia cztery fazy: Rozpoczęcie (ang. Inception), Wypracowanie (ang. Elaboration), Konstrukcja (ang. Construction) oraz Przejście (ang. transition). Należy jednak podkreślić, że iteracje w metodyce OpenUP spełniają podstawową definicję iteracji. Każda iteracja kończy się wykonywalną **wersją systemu** (ang. Build). Wersja stanowi częściowy lub ostateczny produkt, który oddajemy klientowi do oceny. W ramach iteracji poszczególne role wykonują zadania, których rezultatem jest powstanie produktów pracy. Co kilka dni oddawany jest kolejny **przyrost** (ang. Increment) systemu, czyli zintegrowany i zainstalowany w środowisku testowym system. Harmonogram działań w ramach iteracji określa **plan iteracji**.



Rysunek 3.8: Cykl wytwarzczy metodyki Open Unified Process

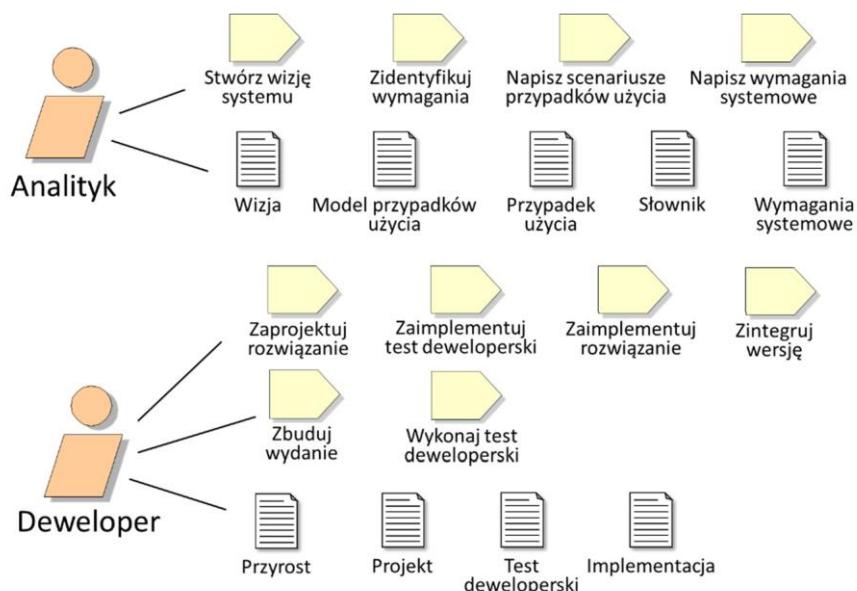
Podział projektu na cztery fazy jest cechą charakterystyczną metodyk zunifikowanych (RUP, OpenUP). Często niestety taki podział powoduje mylne przekonanie, że metodyki te stosują cykl wodospadowy. Jest to jednak błęd. Każda faza składa się z jednej lub kilku iteracji. Rezultatem **fazy rozpoczęcia** powinien być zdefiniowany zakres funkcjonalny systemu, czyli nacisk położony jest na działania w dyscyplinie wymagań. Przy tym, w fazie tej wykonywane są również czynności projektowe, implementacyjne i testowe. Efektem jest wstępnie zaimplementowana pierwsza – bardzo jeszcze niepełna funkcjonalnie – wersja systemu. Faza rozpoczęcia najczęściej składa się z jednej iteracji. W **fazie wypracowania** następuje stabilizacja („wypracowanie”) architektury systemu. Odbywa się to poprzez położenie nacisku na czynności projektowe (architektoniczne) i weryfikację rozwiązań projektowych poprzez implementację kolejnych wersji systemu. Faza wypracowania trwa najczęściej od jednej do trzech iteracji.

Faza **konstrukcji** jest zazwyczaj najdłuższa i składa się najczęściej z 3-7 iteracji. Przed rozpoczęciem fazy konstrukcji powinny być już rozwiązane najważniejsze problemy techniczne. W ramach tej fazy w kolejnych iteracjach powstają wersje systemu uzupełniane o kolejne przyrosty funkcjonalności. Iteracje fazy konstrukcji kładą nacisk na implementację systemu, jednak cały czas wykonywane są czynności w obszarach wymagań, projektowania, testowania oraz wdrożenia. Projekt kończy faza **wdrożenia**. W ramach tej fazy dokonywane jest uzupełnienie funkcjonalności systemu będące wynikiem ewentualnych zmian zgłaszanych przez klienta. W tej fazie szczególny nacisk położony jest na czynności w obszarze wdrożenia (instalacja produkcyjna, dokumentowanie, szkolenia). Mimo to, również ta faza zawiera czynności pozostacych obszarów, łącznie z implementacją i testowaniem, i kończy się kolejną – już ostateczną – wersją systemu.

Rysunek 3.8 pokazuje również dwa wykresy opisujące profil ryzyka i profil wartości systemu. Profile te są charakterystyczne dla większości metodyk iteracyjnych. Profil wartości odpowiada w przybliżeniu liczbie zrealizowanych jednostek funkcjonalnych, co jest jednoznaczne z poziomem przydatności (stosowalności) aktualnej wersji systemu dla klienta. W dobrze prowadzonym projekcie iteracyjnym wartość produktu systematycznie rośnie z każdą zakończoną iteracją. Profil ryzyka pokazuje liczbę istotnych zagrożeń w projekcie. W projekcie iteracyjnym powinniśmy dążyć do pokonania najważniejszych

zagrożeń technicznych i organizacyjnych w pierwszych iteracjach. Dzięki temu, profil ryzyka dosyć szybko spada, aby w fazach konstrukcji i przejścia dążyć do zera.

Projekty prowadzone metodami OpenUP i RUP są sterowane jednostkami funkcjonalności nazywanymi **przypadkami użycia** (ang. Use Case) systemu. Model przypadków użycia jest głównym produktem roli Analityka (patrz rysunek 3.9). Rola przypadku użycia jest podobna do roli historii użytkownika w metodyce XP. Planowanie strategiczne projektu odbywa się poprzez zarządzanie modelem przypadków użycia i poprzez przydział przypadków użycia do poszczególnych iteracji. Podobnie jak historia użytkownika, przypadek użycia stanowi nieduży, lecz kompletny fragment funkcjonalności systemu, stanowiący określoną wartość (biznesową) dla jego użytkownika. O przypadkach użycia będziemy mówić szczegółowo w ramach modułów II (Modelowanie oprogramowania) i III (Podstawy inżynierii wymagań i projektowania oprogramowania).



Rysunek 3.9: Role podstawowe w metodyce Open Unified Process (1)

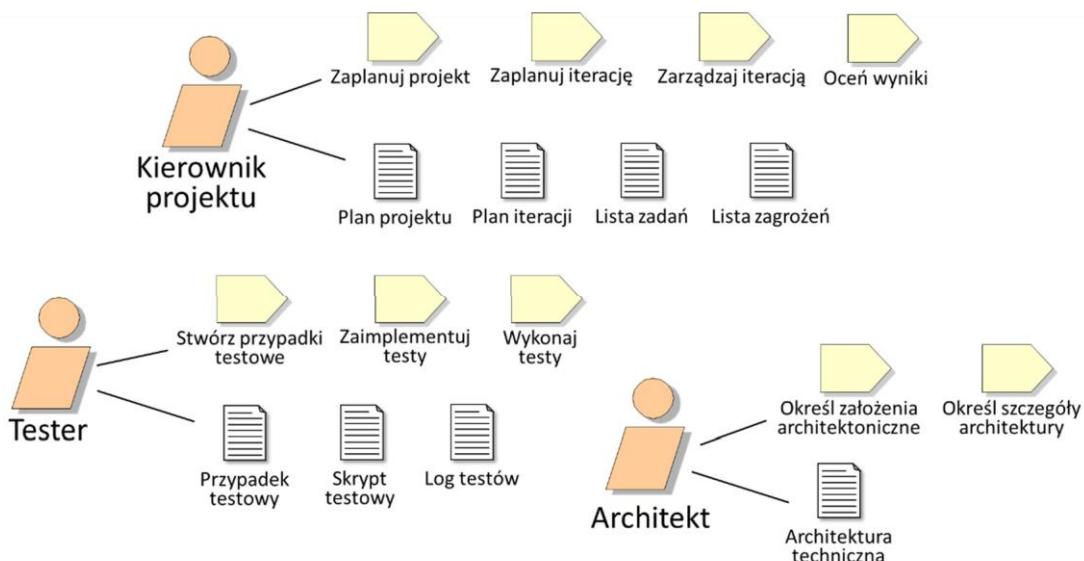
Metodyka OpenUP definiuje trzy grupy ról: role podstawowe, role wdrożeniowe oraz role pomocnicze (środowiska). W opisie metodyki pominiemy szczegółowy opis ról wdrożeniowych. Ogólnie, dotyczą one działań związanych z instalacją systemu w środowisku produkcyjnym, przygotowaniem dokumentacji dla klienta oraz szkoleń. Istotną rolą w tej grupie jest rola **właściciela produktu**, czyli reprezentanta klienta. Jest ona analogiczna do podobnej roli w metodyce Scrum.

Role podstawowe w metodyce OpenUP zostały przedstawione na rysunkach 3.9 i 3.10. Dodatkowo, wśród ról podstawowych wyróżnia się rolę Udziałowca (ang. Stakeholder) oraz tzw. Dowolną Rolę (ang. Any Role). Reprezentują oni osoby zainteresowane wynikami projektu i mogące uczestniczyć w większości działań z obszaru wymagań i projektowania oraz wszystkie osoby zgłaszające zmiany (np. zmiany wymagań).

Analityk (patrz rysunek 3.9) odpowiada za pięć produktów. Składają się one na pełną specyfikację wymagań budowanego systemu. Najbardziej ogólnym produktem jest **wizja** (ang. Vision), która opisuje cechy systemu bezpośrednio wynikające z potrzeb biznesowych klienta. **Model przypadków użycia** wykorzystuje graficzną notację języka UML. Zawiera on specyfikację przypadków użycia, aktorów (użytkowników i systemów zewnętrznych) oraz relacje między nimi. Każdy **przypadek użycia** opisywany jest w sposób szczegółowy poprzez definicję jego scenariuszy oraz scenariuszy (opisów interfejsu użytkownika). Dodatkowo, tworzone są **wymagania systemowe**, które definiują cechy jakościowe, które

najczęściej przebiegają przez cały system (np. użyteczność, wydajność, niezawodność). Dla wszystkich wymagań tworzony jest **słownik** (ang. Glossary), który ma na celu uchwycenie terminologii dziedziny problemu oraz lepsze porozumienie w zespole deweloperskim oraz z klientem. Produkty pracy analityka omawiamy szczegółowo w module III (Podstawy inżynierii wymagań i projektowania oprogramowania).

Deweloper (patrz rysunek 3.9) w metodyce OpenUP łączy funkcje projektanta i programisty. Współpracuje on ściśle z **Architektem**, którego zakres obowiązków jest przedstawiony na rysunku 3.10. Podstawowym produktem pracy, opisującym całość systemu jest **architektura techniczna**. Definiuje ona m.in. podział systemu na komponenty, interfejsy między komponentami, przydział komponentów do konkretnych jednostek wykonawczych (architektura fizyczna) oraz przedstawia decyzje dotyczące stosowanych technologii (języków, bibliotek, platform programistycznych). Architekt współpracuje z projektantami, przydzielając im konkretne komponenty (podsystemy). Bazując na ogólnych decyzjach architektonicznych, projektanci opracowują szczegółowe plany dla poszczególnych komponentów systemu. Efektem jest powstanie **projektu** (ang. Design) systemu. Projekt zawiera definicję struktury komponentów (np. definicje klas, interfejsów) oraz dynamiki ich działania (np. opis dynamiki współpracy między obiektami). Należy podkreślić, że zarówno architektura jak i projekt systemu są produktami pracy „życzącymi” przez cały czas projektu. W kolejnych iteracjach produkty te są korygowane oraz uzupełniane o nowe elementy potrzebne podczas realizacji przydzielonych przypadków użycia. Podstawową notacją zalecaną do tworzenia większości produktów projektowych jest język UML, który jest dokładnie omawiany w module II (Modelowanie oprogramowania). Ponadto, produkty pracy dewelopera i projektanta omawiamy szczegółowo w module III (Podstawy inżynierii wymagań i projektowania oprogramowania).



Rysunek 3.10: Role podstawowe w metodyce Open Unified Process (2)

Deweloper odpowiada również za trzy produkty pracy z obszaru implementacji: **Implementacja**, **Przyrost** oraz **Test deweloperski**. Pierwszy z nich zawiera kod źródłowy oraz różne pliki pomocnicze (np. skrypty budujące przyrost). Drugi produkt to wykonywalna wersja systemu. Jest to główny produkt każdego projektu. Przyrost powinien być gotowy do zainstalowania w środowisku wykonawczym w celu przeprowadzenia testów akceptacyjnych przez testera. Test developerski odpowiada testowi jednostkowemu, który został omówiony w ramach opisu metodyki XP. Podobnie jak produkty projektowe, produkty implementacyjne podlegają ciągłej rozbudowie w miarę dodawania funkcjonalności w kolejnych iteracjach.

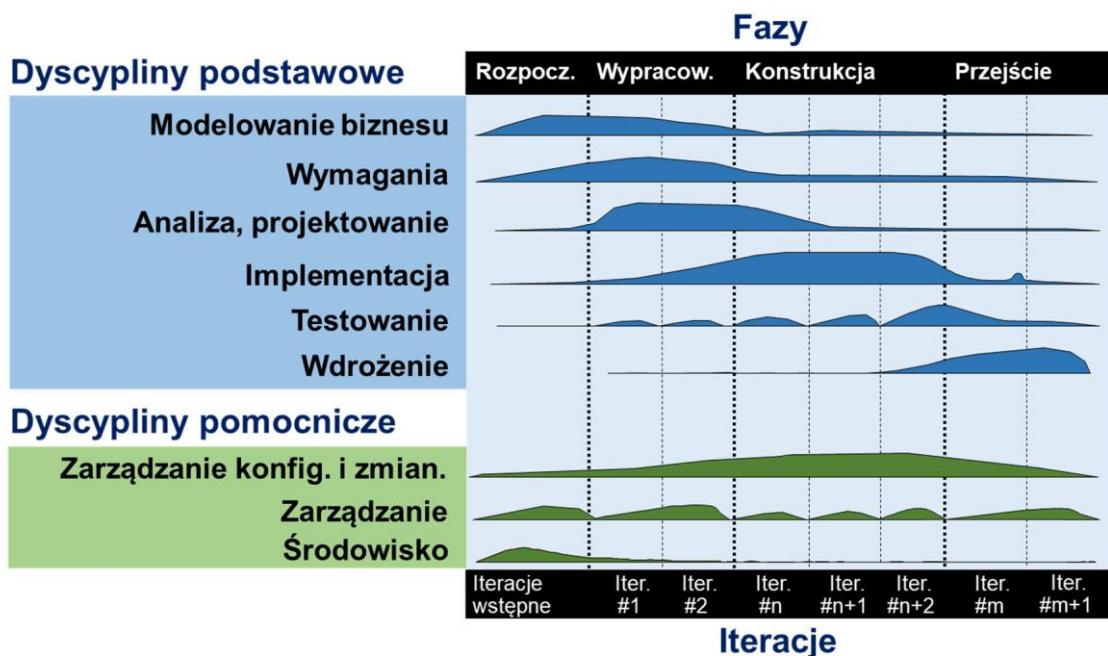
Na rysunku 3.10 przedstawiono również role Testera i Kierownika Projektu. **Tester** ma bardzo podobny zakres obowiązków, co tester w metodyce XP, zatem pominiemy tutaj jego szczegółowy opis. **Kierownik projektu** jest rolą wyraźnie wskazującą na bardziej tradycyjne podejście metodyki OpenUP do procesu wytwarzania oprogramowania. Jest to typowa rola zarządcza, która nie występuje w metodykach zwinnych w sposób jawny. Kierownik tworzy **plan projektu** oraz **plany iteracji**. Jeśli projekt jest prowadzony w sposób zwinny, produkty te mogą być tworzone podobnie jak rejestr produktu i rejestr sprintu w metodyce Scrum. Można też użyć tradycyjnych narzędzi i notacji, jak np. diagramy Gantta. Rolą kierownika projektu jest również ocena wyników iteracji, w tym ocena zagrożeń.

Role podstawowe uzupełniają role pomocnicze, dotyczące zarządzania środowiskiem pracy zespołu (patrz rysunek 3.11). **Inżynier procesu** spełnia rolę podobną do mistrza młyna w metodyce Scrum lub trenera w metodyce XP. Jego zakres obowiązków obejmuje wspieranie zespołu w dostosowywaniu metodyki OpenUP do rzeczywistych warunków w projekcie oraz w organizacji wytwarzającej oprogramowanie, a także wdrażanie metodyki do praktyki. **Specjalista narzędziowiec** wspiera zespół w instalacji oraz konfiguracji narzędzi (środowisko deweloperskie, narzędzia CASE, narzędzia do testowania itd.). Jego udział w projekcie jest kluczowy w pierwszych iteracjach, a potem polega na ciągłym administrowaniu narzędziami i zapewnianiu ich sprawnego użytkowania przez zespół.



Rysunek 3.11: Role pomocnicze (środowiska) w metodyce Open Unified Process

Metodyka OpenUP kieruje się duchem metodyku RUP i dostosowuje ją do warunków małych i średnich projektów. W niektórych projektach, wymagane jest jednak zdecydowanie bardziej szczegółowe określenie ról. Szczególnie dotyczy to dużych zespołów rozproszonych geograficznie. Metoda Rational Unified Process definiuje 33 role – 8 ról analityków (np. analityk biznesowy, analityk systemowy), 10 ról deweloperów (np. architekt, projektant, programista), 7 ról kierowniczych (np. kierownik, inżynier procesu), 7 ról pomocniczych (np. interesariusz, administrator) oraz rolę testera. Role te są podzielone między dyscypliny, które są przedstawione na rysunku 3.12. Oprócz typowych dyscyplin technicznych omówionych w poprzednich rozdziałach, RUP definiuje dyscyplinę Modelowania biznesu oraz kilka dyscyplin pomocniczych. Rysunek 3.12 dodatkowo ilustruje typowe natężenie pracy w poszczególnych dyscyplinach w przekroju całego projektu, tzn. w kolejnych fazach i iteracjach.



Rysunek 3.12: Cykl wytwarzczy metodyki Rational Unified Process

Bardziej dokładne omówienie ról, czynności i produktów metodyki RUP wykracza poza zakres niniejszego podręcznika. Warto jednak podkreślić, że mimo swoich rozmiarów, metodyka ta może być stosowana nawet jako metodyka zwinna. Twórcy metodyki zalecają w sposób bardzo zdecydowany do stosowywanie metodyki do warunków projektu. Dlatego też, w typowym projekcie prowadzonym metodą RUP wykorzystywanych jest jedynie część ról. Takie podejście może stanowić również podsumowanie niniejszego rozdziału. Każdy projekt ma swoją specyfikę i dobrą metodą do warunków projektu jest kluczową decyzją podejmowaną przez zespół projektowy lub całą organizację wytwarzającą oprogramowanie.

Zadania

Zadanie 1

Proszę zbudować rejestr produktu w metodyce SCRUM dla zadanego systemu: określić historie użytkownika (około 10), oraz przydzielić historie użytkownika do dwóch pierwszych sprintów.

Zadanie 2

Proszę określić role w metodykach SCRUM oraz OpenUP, które są odpowiedzialne za wykonanie czynności w obrębie dyscyplin IW oraz APO.

Zadanie 3

Mając zadany opis dziedziny problemu dla budowanego systemu i jego zakres proszę zaproponować zastosowanie określonej metodyki wytwarzania oprogramowania i uzasadnić ten wybór.

Zadanie 4

Mając zadany opis zakresu systemu proszę określić listę podstawowych cech funkcjonalnych systemu. Proszę wybrane cechy systemu przydzielić do pierwszych 2 iteracji. Proszę uzasadnić wybór. Proszę zapewnić, aby cechy funkcjonalne były w całości realizowalne w ramach jednej iteracji (2-3 tygodnie).

Słownik pojęć

Metodyka

Usystematyzowany i zorganizowany zbiór technik objętych cyklem wytwarzyczym, którego produkty wyrażone są za pomocą odpowiedniej notacji.

Produkt pracy

Ostateczny lub pośredni wyniki działań w projekcie, wykorzystywany do podejmowania innych działań. Produktem pracy może być dokument, model, element modelu, kod lub element montażowy.

Rola

Abstrakcyjna definicja pewnego zakresu obowiązków (zbioru czynności) wymagających odpowiedniego zakresu kwalifikacji.

Co trzeba zapamiętać

Co to są metodyki wytwarzania oprogramowania

Zespoły projektowe potrzebują konkretnych wytycznych, które odpowiadają na podstawowe pytanie „jak należy to zrobić?”. Zespół powinien widzieć co należy wykonać, w jaki sposób oraz kto powinien za co odpowiadać. Opis metodyki zazwyczaj zawiera zatem pięć elementów: opis ról, opis czynności, opis produktów pracy, podział na dyscypliny oraz określenie cyklu życia (zazwyczaj ograniczonego do cyklu wytwarzyczego). Ważnym elementem definicji produktów pracy jest notacja używana do ich tworzenia. Przyjęcie jednolitej notacji jest bardzo istotnym elementem zapewnienia jakości wykonywanych produktów. Wśród wielu istniejących i stosowanych metod, można wyróżnić dwie główne grupy: metodyki formalne i agilne (*zwinne, ang. agile*).

Metodyki zwinne

Podstawowe cechy metodyk zwinnych (*agile*) to iteracyjność oraz koncentracja na technikach organizacji zespołu i współpracy z klientem. Bardzo istotna zasadą metodyk zwinnych jest częste dostarczanie działającego oprogramowania. Odbywa się to w stosunkowo krótkich cyklach (iteracjach). Na pierwszy plan wysuwa się zasada zapewnienia maksymalnej satysfakcji klienta. Z najważniejszych metod zwinnych warto wymienić Scrum, Extreme Programming, Crystal, Lean Development i Kanban. Często stosowana metoda Scrum zakłada codzienną kontrolę postępu prac w ramach iteracji (tzw. sprintu) trwającego kilka tygodni (najczęściej 2-3). Każdy sprint funkcjonuje w oparciu o tzw. rejestr sprintu, a jego efektem jest przyrost produktu, który może być w określonym zakresie używany przez klienta. Kolejne sprinty planowane są w ramach tzw. rejestru produktu.

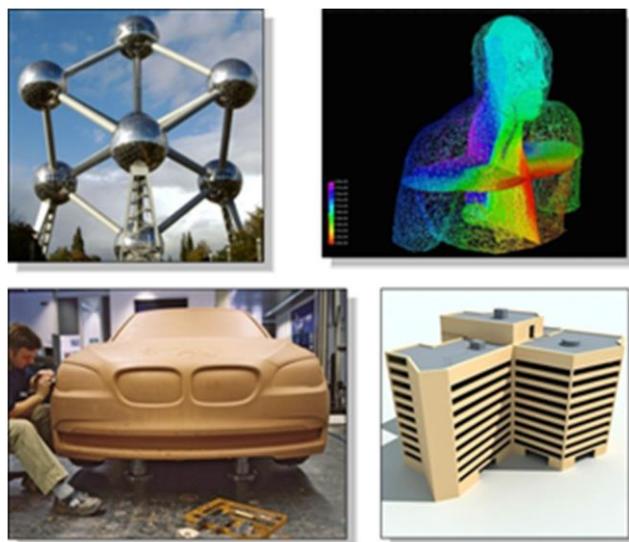
Metodyki sformalizowane

W projektach zamawianych przez instytucje stosujące formalne zasady zamawiania oprogramowania często konieczne jest zastosowanie metodyki sformalizowanej. Metodyka sformalizowana precyjnie definiuje role, czynności i produkty pracy. Współczesne metodyki sformalizowane stosują cykl iteracyjny i mogą być zastosowane do projektów o różnej skali. Popularnymi metodykami sformalizowanymi są metodyki z rodziny tzw. procesów zunifikowanych. Rational Unified Process jest metodyką bardzo obszerną i możliwą do zastosowania w bardzo szerokim zakresie dziedzin oraz rozmiarów projektów. Ta uniwersalność metodyki czyni ją jednocześnie trudną do wdrożenia w konkretnym projekcie. Wymagane są stosunkowo pracochłonne czynności związane z adaptacją metodyki do konkretnego środowiska projektowego. Dlatego też opracowano uproszczoną jej wersję – OpenUP, zbliżoną zasadami do metodyk zwinnych.

4. Wprowadzenie do modelowania obiektowego

4.1. Podstawowe zasady modelowania

Gdy przeglądamy słowniki w poszukiwaniu słowa „model”, napotykamy kilka definicji. Oto niektóre z nich: „wzór, według którego coś jest lub ma być wykonane”, „konstrukcja, schemat lub opis ukazujący działanie, budowę, cechy, zależności jakiegoś zjawiska lub obiektu”, „przedmiot będący kopią czegoś, wykonany zwykle w odpowiedniej skali”. Ogólnie można powiedzieć, że **model** jest reprezentacją określonej rzeczywistości, która pomaga w zrozumieniu tej rzeczywistości. Konkretne przyczyny tworzenia modeli są tak różnorodne jak dziedziny w których modele są stosowane. Modelowanie obecne jest w większości dziedzin działalności człowieka, np. nauce, technice, edukacji czy sztuce. Rysunek 4.1 przedstawia przykłady różnych modeli: model kryształu żelaza zbudowany w powiększeniu, model numeryczny ludzkiego ciała dla potrzeb analizy pola elektromagnetycznego, model nadwozia samochodu oraz model budynku



Rysunek 4.1: Przykładowe modele w różnych dziedzinach

Budowa modeli może mieć różne cele, z których najważniejsze to:

- Testowanie właściwości fizycznych obiektów przed ich wykonaniem. Modele samochodów i samolotów są testowane pod kątem właściwości aerodynamicznych. Modele konstrukcji budowlanych testowane są pod kątem rozkładu sił. Testowanie modeli obiektów daje duże oszczędności, gdyż możliwe jest udoskonalenie konstrukcji przed zbudowaniem rzeczywistego obiektu, zwłaszcza, że przy obecnym rozwoju technik komputerowych, testowane obiekty często są jedynie modelami wirtualnymi.
- Wizualizacja i komunikacja z klientami. Architekci oraz projektanci różnych przedmiotów użytkowych budują makietę bądź tworzą wizualizacje w programach graficznych, aby zaprezentować efekt swojej pracy przed rzeczywistym wykonaniem projektu. Wizualizacja ma znaczenie również wtedy, gdy zachodzi potrzeba przedstawienia budowy lub działania czegoś, co jest zbyt małe by zaobserwować to gołym okiem.

- Redukcja złożoności. Jest to jedna z najważniejszych przyczyn, dla których wykonywane są modele. Rzeczywistość składa się z bardzo wielu elementów, między którymi występuje szereg powiązań i wzajemnych oddziaływań. Z kolei, ludzki mózg ma ograniczoną zdolność jednoczesnego przetwarzania wielu elementów. Według badań psychologicznych, optymalna liczba elementów, które człowiek jest w stanie skutecznie przetwarzać w danej chwili jest 7 (+/- 2). Aby pokonać tę barierę, budowane są modele, które upraszczają rzeczywistość poprzez odseparowanie jedynie tych elementów rzeczywistości, które są niezbędne w danym momencie i pominięcie wszystkich innych.

Z punktu widzenia inżynierii oprogramowania najbardziej interesuje nas ten ostatni z wymienionych celów, czyli radzenie sobie ze złożonością otaczającej nas rzeczywistości. Wspomniana wyżej zasada koncentracji na elementach najważniejszych w danym kontekście jest jedną z podstawowych technik stosowanych przez ludzki umysł. Technika ta nazywa się **abstrakcją**. Żeby ją zastosować, w pierwszej kolejności należy sobie uświadomić jak bardzo istotne są poszczególne charakterystyki modelowanego przez nas w danym momencie tematu z punktu widzenia naszego aktualnego celu. Przykładowo modelując tor lotu piłki znikome znaczenie będzie miał dla nas jej kolor. Zamiast tego należy odzwierciedlić odpowiednie zjawiska zgodnie z prawami fizyki. Wyróżniamy zatem cechy danego systemu stosownie do ich wagi, wyróżniając istotne, a odpowiednio mniej eksponując lub wręcz pomijając te o mniejszym znaczeniu. Daje nam to podstawę do zastosowania dwóch technik realizujących zasadę abstrakcji, czyli generalizacji i klasyfikacji.

Klasyfikacja polega na grupowaniu elementów świata rzeczywistego w kategorie, zgodnie z przyjętymi kryteriami, najczęściej nawiązującymi do jakichś ich wspólnych charakterystyk. Na przykład, w przestrzeni kosmicznej możemy wyróżnić takie grupy obiektów jak gwiazdy, planety czy planetoidy. Na Ziemi możemy wyróżnić takie jednostki podziału terytorium jak: kontynenty, państwa, miasta, dzielnice. Budynki znajdujące się w miastach możemy sklasyfikować jako uczelnie, budynki mieszkalne, fabryki, itd. Przykład klasyfikacji został pokazany na rysunku 4.2.



Rysunek 4.2: Przykład zastosowania zasady abstrakcji

Generalizacja polega na tworzeniu pojęć ogólnych poprzez wyodrębnienie pewnego zbioru cech wspólnych dla szeregu pojęć bardziej szczegółowych. Im mniejszy zbiór cech wspólnych, tym bardziej ogólne pojęcie i tym większy zakres konkretnych obiektów, do których pojęcie może być zastosowane. W miarę powiększania zbioru cech wspólnych, pojęcie staje się coraz bardziej szczegółowe, obejmując coraz mniejszy zakres konkretnych obiektów. Generalizacja pozwala tworzyć hierarchię pojęć. Dla przykładu, pojęciem „uczelnia” możemy nazwać wszystkie istniejące szkoły wyższe. Możemy jednak

dokonać podziału na uczelnie techniczne oraz uczelnie nietechniczne. Innym sposobem podziału byłby podział na uczelnie państwowie i prywatne.

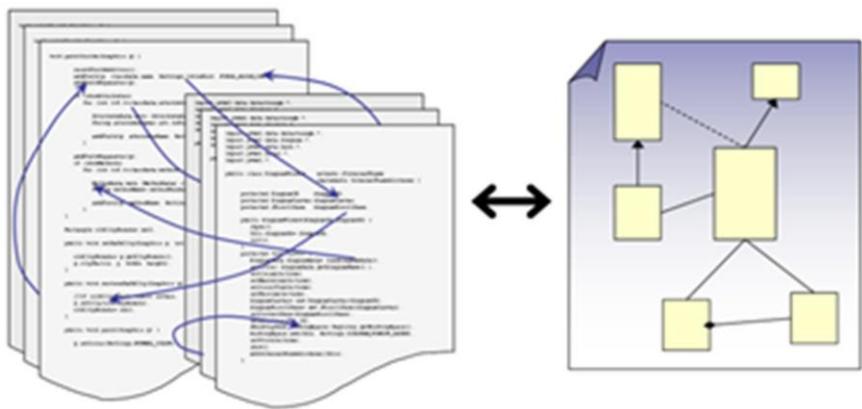
Budując abstrakcyjne modele nie szukamy prawdy absolutnej o otaczającej nas rzeczywistości. Zamiast tego z reguły skupiamy się na wybranym jej fragmencie, który jest najbardziej adekwatny dla celu, jaki przyświeca nam podczas budowy danego modelu. Może zatem istnieć wiele poprawnych, choć mocno różniących się między sobą modeli, które ukazują różne aspekty modelowanego obiektu czy zjawiska.

Oprogramowanie również można potraktować jako model określonego fragmentu świata rzeczywistego. Co ważniejsze jest ono również samo w sobie wysoce skomplikowanym jego elementem. Oprogramowanie w dzisiejszym świecie jest obecne niemal w każdej dziedzinie życia. Może, na przykład, symulować zjawiska fizyczne, sterować działaniem urządzeń technicznych czy odwzorowywać procesy biznesowe zachodzące w różnych przedsiębiorstwach jak banki, uczelnie, placówki handlowe, itp. Duże systemy oprogramowania muszą więc odzwierciedlać ogromne ilości obiektów i mechanizmów zachodzących w danym środowisku, które system ma realizować. Gotowy system składa się zazwyczaj z setek tysięcy linii kodu.

Często również bywa tak, że dana dziedzina nie jest dobrze znana twórcom przystępującym do budowy systemu oprogramowania a wymagania zamawiającego system zmieniają się w trakcie jego realizacji. Trudno byłoby sobie wyobrazić przystąpienie do pisania kodu dużego systemu jedynie na podstawie przekazanego przez zamawiającego, nieformalnego opisu problemu. Taki sposób działania można by porównać do próby zbudowania wieżowca bez szczegółowych planów i modeli architektonicznych lub próby nakręcenia pełnometrażowego filmu fabularnego bez szczegółowego scenariusza i scenopisów.

W całym procesie tworzenia systemu oprogramowania, sam etap pisania kodu zajmuje zazwyczaj stosunkowo niewielką część czasu. Zanim przystąpimy do tego etapu, musimy stworzyć szereg modeli pośrednich, które opisują budowany system oraz jego środowisko z różnych punktów widzenia oraz na różnym poziomie szczegółowości. Wyróżniamy modele struktury, które uwypuklają statyczną budowę systemu, oraz modele zachowania, które opisują aspekty dynamiczne. Modele te mają istotne znaczenie dla właściwego zrozumienia funkcjonalności, jaką system ma zrealizować, jak również pozwalają stworzyć architekturę systemu, którą można elastycznie modyfikować w przypadku zmieniających się wymagań.

Szczegółowe modele projektowe pozwalają stworzenie dobrego, przejrzystego kodu oprogramowania. Można powiedzieć, że model może stanowić wizualną mapę dla kodu, co ilustruje rysunek 4.3. Model abstrahuje od szczegółów kodu (np. treści metod) i pokazuje np. tylko strukturę powiązań między klasami. Programista może dzięki temu łatwiej zorientować się w strukturze kodu i odpowiedzialności poszczególnych klas.



Rysunek 4.3: Model, jako wizualna mapa kodu systemu

Dzięki modelowaniu możemy osiągnąć następujące korzyści:

- Zrozumienie celu budowy systemu oraz sposobu jego realizacji,
- Ułatwienie komunikacji pomiędzy twórcami systemu oraz zamawiającym,
- Ułatwienie zarządzania realizacją systemu oraz zarządzanie ryzykiem,
- Ułatwienie dokumentowania systemu.

Modele powinny być podstawowym produktem każdego projektu wytwarzania oprogramowania. Im większy i bardziej złożony system, tym większe znaczenie modelowania.

4.2. Uniwersalny język modelowania

Jak wiemy, programy możemy pisać w wielu różnych językach programowania. Obecnie, najpowszechniej wykorzystywane są języki obiektowe, takie jak Java, C#, Python czy TypeScript. Pierwszym w pełni obiektowym językiem programowania był Smalltalk, powstały w latach 70-tych XX wieku. W związku z rosnącą popularnością obiektowych języków programowania zaczęły również pojawiać się obiektowe języki modelowania. W latach 90tych nastąpił proces unifikacji, co doprowadziło do powstania ujednoliconego języka modelowania – **UML** (ang. Unified Modelling Language). Twórcami języka byli Grady Booch, James Rumbaugh i Ivar Jacobson, którzy połączyli stosowane wcześniej różne notacje w jeden uniwersalny język. UML jest obecnie najpowszechniej stosowanym językiem modelowania, który jednocześnie został zdefiniowany jako standard (m.in. jako standard ISO). UML jest językiem graficznym, za pomocą którego można tworzyć modele niezbędne w całym procesie budowy systemu informatycznego. Można go również wykorzystywać do modelowania rzeczywistości niekoniecznie powiązanej z systemem informatycznym. W dalszej części książki będziemy sukcesywnie przedstawiać składnię i semantykę języka UML w wersji 2. Pełną specyfikację języka znaleźć można na stronach internetowych organizacji Object Management Group (www.omg.org), która zajmuje się rozwojem i standaryzacją języka.

Booch, Rumbaugh i Jacobson tworząc język UML kierowali się dobrymi praktykami modelowania. Poniżej przedstawiamy kilka z nich, które dobrze definiują cele i zasady rządzące procesem modelowania.

„Podjęcie decyzji, jakie modele tworzyć, ma wielki wpływ na to, w jaki sposób zaatakujemy problem i jaki kształt przyjmie rozwiążanie.”

Stając przed jakimś problemem, należy skupić się na tworzeniu takich modeli, które prowadzą najkrótszą drogą do właściwego rozwiązania problemu. Tworzenie źle dobranych modeli jest stratą czasu i może oddalać nas od celu, do jakiego dążymy.

„Każdy model może być opracowany na różnych poziomach szczegółowości.”

Poziom szczegółowości modelu zależy od tego, dla kogo i w jakim celu się ten model tworzy. W przypadku systemu informatycznego, dla zamawiającego oraz analityka przydatne są modele na małym poziomie szczegółowości, które pokazują, co ma być zrobione. Projektant oraz programista będą natomiast potrzebowali modeli bardziej szczegółowych i precyzyjnych, które koncentrują się na tym, jak coś ma być wykonane. Poziom szczegółowości wrasta w miarę postępu prac nad systemem.

„Najlepsze modele odpowiadają rzeczywistości.”

Model jest tym lepszy, im lepiej oddaje modelowaną rzeczywistość. Oczywiście, jak wspominaliśmy wcześniej, modele upraszczają skomplikowaną rzeczywistość, aby łatwiej można było ją zrozumieć. Istotne jest zatem, aby to uproszczenie nie dotyczyło elementów istotnych. Model powinien pomijać tylko te szczegóły, które nie mają znaczenia dla realizacji naszego celu.

„Żaden jeden model nie jest wystarczający. Niewielka liczba niemal niezależnych modeli to najlepsze rozwiązanie w wypadku każdego niebanalnego systemu.”

Modele powinny być przygotowywane oraz analizowane niezależnie, mimo że są ze sobą do pewnego stopnia powiązane. Dla systemu oprogramowania, możemy wyróżnić kilka różnych perspektyw: pojeciowa, specyfikacyjna oraz implementacyjna. Każda z nich może opisywać aspekty statyczne, jak i dynamiczne systemu. Mimo, że perspektywy te dopełniają się nawzajem i mają części wspólne, to model dla każdej z nich powinien być możliwy do przeanalizowania w oderwaniu od pozostałych.

4.3. Obiekty jako podstawa modelowania

Jak już wspomnieliśmy, najbardziej rozpowszechnionym współcześnie paradigmatem modelowania jest modelowanie obiektowe. Zgodnie z nazwą opiera się ono na obiektach, czyli wyodrębnionych elementach rzeczywistości istotnych z perspektywy tworzonego modelu. Każdy obiekt stanowi osobny byt wyraźnie wyodrębniony z całości modelowanej dziedziny. Obiekty mogą zatem reprezentować różnego rodzaju przedmioty, osoby, zdarzenia, procesy lub inne twory niematerialne występujące w danym środowisku.

Obiekty stanowią dobrą płaszczyznę porozumienia w projekcie konstrukcji oprogramowania. Częstym problemem w takich projektach jest bowiem zasadnicza różnica wiedzy na temat dziedziny problemu oraz możliwości technicznych oprogramowania. Z jednej strony mamy zamawiających, którzy posiadają wiedzę na temat dziedziny, którą ma wspierać system oprogramowania. Brakuje im jednak wiedzy technicznej na temat sposobów i możliwości jego realizacji. Z drugiej strony mamy deweloperów, którzy wiedzą jak zrealizować system od strony technicznej, jednak ich wiedza na temat dziedziny problemu jest zwykle ograniczona. Oznacza to, że jedni i drudzy posługują się innymi „językami” w swojej pracy. Dlatego też bardzo istotne jest zapewnienie platformy porozumienia, jakiej dostarcza paradigmat modelowania obiektowego, co ilustruje rysunek 4.4.



Rysunek 4.4: Obiekty – wspólny język dla tworzenia modeli

Z punktu widzenia zamawiającego system, obiekty odpowiadają rzeczywistym elementom modelowanego środowiska (przedmiotom, osobom itd.). Z punktu widzenia programistów, obiekty są podstawowymi jednostkami implementacji oprogramowania w obiektowych językach programowania. Obiekty będące elementami oprogramowania nie zawsze odzwierciedlają w pełni obiekty rzeczywiste z określonego środowiska – często są ich uproszczeniem lub modyfikacją. Ponadto, w systemach oprogramowania występuje szereg dodatkowych obiektów związanych nie ze środowiskiem, lecz z techniczną stroną oprogramowania, jak np. okna, formularze wprowadzania danych, interfejsy, bazy danych, obiekty sterujące działaniem systemu, itp. Na podstawie obiektów z danej dziedziny świata rzeczywistego, analitycy i projektanci tworzą modele obiektowe, które z kolei są podstawą do stworzenia kodu oprogramowania. Modelowanie obiektowe polega więc na:

- znajdowaniu interesujących nas obiektów rzeczywistych w danej dziedzinie,
- opisywaniu struktury i sposobu działania tych obiektów,
- klasyfikacji i generalizacji obiektów,
- znajdowaniu powiązań między nimi,
- opisywaniu dynamicznych aspektów współpracy pomiędzy obiektami.

Zgodnie z paradygmatem obiektowości, **obiekt** (ang. *object*) posiada trzy główne cechy: **tożsamość, stan i zachowanie**. Cechy te omówimy na podstawie prostego przykładu.

Rysunek 4.5 przedstawia dwa samochody, które wykorzystamy jako przykład ilustrujący opis obiektów. Oba auta są bardzo podobne do siebie – ich marka, model, parametry techniczne i właściwości jazdne są identyczne. Mogą się jednak różnić wartościami takich właściwości jak kolor, przebieg oraz stan baku. Różni są też właściciele obu samochodów. Można sobie jednak wyobrazić, że przebieg, stan baku czy nawet kolor oraz właściciel pojazdu mogą ulec zmianie. Może się zdarzyć tak, że oba samochody będą miały w pewnym momencie dokładnie taki sam stan. Nadal pozostaną to jednak dwa różne obiekty o różnej tożsamości. W przypadku samochodów, o tożsamości może decydować np. numer nadwozia, który jest unikalny dla każdego wyprodukowanego samochodu i pozostaje niezmienny przez cały czas jego użytkowania. Oba samochody mogą się też w określony sposób zachowywać – można je zatankować, włączyć czy wyłączyć silnik, jechać, hamować, itd. Zachowanie obu aut jest dokładnie takie samo i zostało określone przez konstruktorów tego modelu auta.



Rysunek 4.5: Tożsamość, stan i zachowanie obiektów

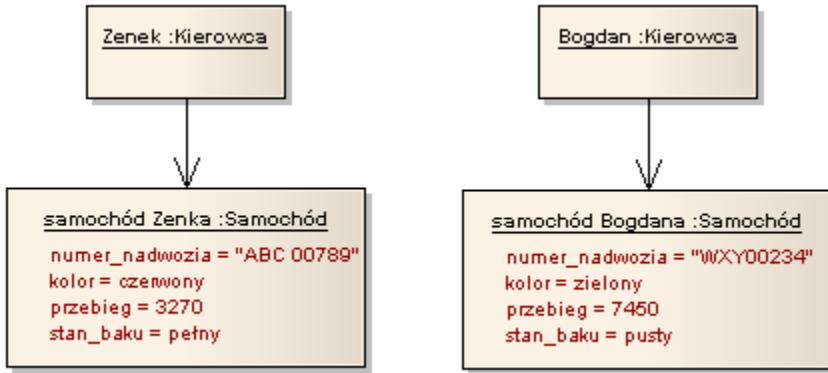
Zdefiniujmy zatem czym są wspomniane wcześniej trzy cechy obiektu.

Tożsamość (ang. identity) obiektu jest cechą umożliwiającą jego identyfikację i odróżnienie od innych obiektów. Tożsamość jest cechą unikalną wśród wszystkich obiektów i pozostaje niezmienna przez cały czas życia obiektu. W przypadku obiektów w systemie oprogramowania, cechą określającą tożsamość obiektu może być adres obszaru pamięci komputera, w którym dany obiekt się znajduje lub specjalna, ukryta właściwość obiektu, której unikalna wartość jest nadawana automatycznie.

Stan (ang. state) obiektu jest określany przez aktualne wartości wszystkich jego właściwości. Każdy obiekt posiada zbiór właściwości, które go charakteryzują. Zbiór ten nie ulega zmianie przez całe życie obiektu. Zmianie mogą ulegać jedynie wartości tych właściwości. Wartości mogą być np. liczbami, napisami czy innymi obiekty.

Zachowanie (ang. behavior) obiektu to zbiór usług, które obiekt potrafi wykonać na rzecz innych obiektów. Zachowanie obiektów określa dynamikę systemu – sposób komunikacji pomiędzy obiektami. Efektem wykonania usługi może być jakaś wartość zwracana obiekowi, który poprosił o wykonanie usługi. Wartość ta może zależeć od aktualnego stanu obiektu wykonującego usługę. Podczas wykonywania usługi, obiekt może operować na swoim zbiorze wartości, w wyniku czego może ulec zmianie jego stanu.

Język UML dostarcza nam odpowiedniej notacji do reprezentowania obiektów – ich stanu i tożsamości. Przykład tej notacji przedstawia rysunek 4.6. Podstawową ikoną obiektu jest prostokąt zawierający podkreślzoną jego nazwę. Po nazwie obiektu można zapisać oddzieloną dwukropkiem nazwę typu (klasy) obiektu (o klasach obiektów będzie mowa w dalszej części rozdziału). W przykładzie na rysunku widzimy np. dwa obiekty o nazwach „Zenek” i „Bogdan”, dla których określony został typ – „Kierowca”. W razie potrzeby możemy również przedstawić aktualny stan obiektu (w tym również tych właściwości, które określają stan obiektu). Listę właściwości obiektu umieszczamy poniżej jego nazwy. Aktualne wartości poszczególnych właściwości mogą być umieszczone po ich nazwie, oddzielone znakiem „=”. Przykładami obiektów, dla których określono stan są „samochód Zenka” i „samochód Bogdana”.



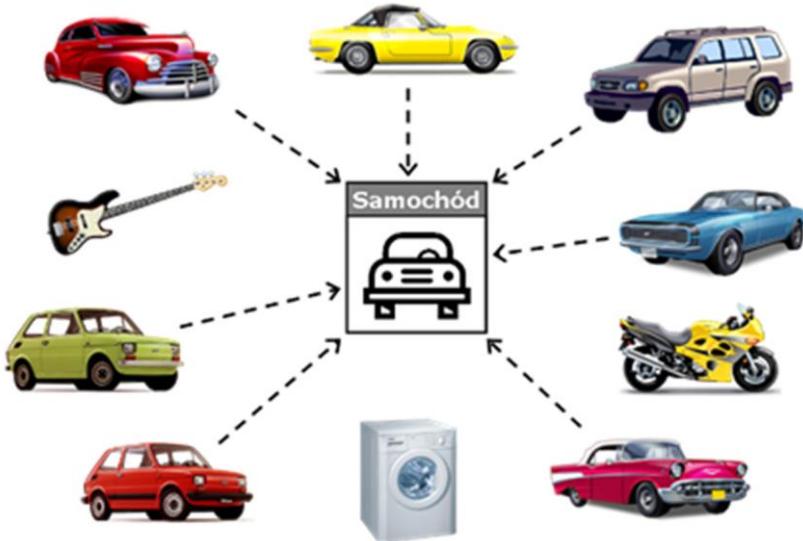
Rysunek 4.6: Notacja dla obiektów w języku UML

Obiekty na diagramie mogą być połączone łącznikami (ang. *Link*), które odzwierciedlają relacje między obiektemi. Powiązane obiekty mogą komunikować się między sobą, na przykład poprzez wywoływanie swoich usług. Powiązania mogą być ukierunkowane, wskazując kierunek komunikacji pomiędzy obiektami. Przykładowo, na rysunku 4.6 obiekt „Bogdan” jest powiązany z obiektem „samochód Bogdana”. Kierunek powiązania wskazuje na to, że obiekt „Bogdan” może poprosić obiekt „samochód Bogdana” o wykonanie jakiejś usługi lub poznać jego stan. Odwrotna komunikacja nie jest natomiast dostępna. To, jakie usługi „samochód Bogdana” może wykonać, określa jego zachowanie. Zachowanie obiektów nie przedstawia się na diagramach obiektów. Zachowanie jest takie samo dla wszystkich obiektów danej klasy, co omawiamy poniżej.

4.4. Klasy obiektów

Typowe dziedziny problemów (np. bankowość, handel, produkcja, zjawiska fizyczne, ...) składają się z tysięcy czy nawet milionów współistniejących obiektów. Wystarczy na przykład rozważyć tematykę obsługi ubezpieczeń komunikacyjnych. Łatwo zaobserwować, że wystąpią w niej jako obiekty wszystkie ubezpieczone samochody oraz ich właściciele. System oprogramowania obsługujący taką dziedzinę problemu będzie musiał obsługiwać dane wszystkich tych obiektów oraz dodatkowo wielu innych. Z oczywistych względów nieracjonalne byłoby tworzenie modelu przedstawiającego wszystkie te obiekty. Konieczne jest opanowanie tej złożoności. Jak już wiemy, jednym ze sposobów radzenia sobie ze złożonością jest stosowanie zasady abstrakcji poprzez klasyfikację. W modelowaniu obiektowym podstawowym elementem modelowania nie będzie zatem obiekt, lecz jego uogólnienie, czyli klasa obiektów. Klasa powinna stanowić opis grupy na tyle podobnych obiektów, że z punktu widzenia perspektywy, dla której tworzymy dany model można je potraktować wspólnie.

Przykład klasyfikacji obiektów przedstawia rysunek 4.7. Spośród wszystkich widocznych na rysunku obiektów możemy wyróżnić grupę takich obiektów, które mają podobne właściwości oraz zachowanie. Obiekty z tej grupy należą do wspólnej klasy obiektów o nazwie „Samochód”. Pozostałe obiekty charakteryzują się mniej lub bardziej odmiennymi właściwościami i zachowaniem niż obiekty klasy „Samochód”. Należą zatem do innych klas. W skrócie, możemy formalnie zdefiniować klasę w następujący sposób: „**Klasa** (ang. *class*) to opis grupy obiektów, które mają taki sam zestaw właściwości oraz sposób zachowania”.



Rysunek 4.7: Klasa Samochód grupuje podobne do siebie obiekty

Każda klasa ma przypisaną nazwę, która wyróżnia ją spośród innych klas w danym kontekście. Właściwości obiektów reprezentowanych przez klasę zwane są **atrybutami** (ang. attribute). Klasa może mieć dowolną liczbę atrybutów (w szczególności – nie mieć żadnych atrybutów). Sposób zachowania obiektów danej klasy nazywamy **operacjami** (ang. operation). Każda operacja określa konkretną usługę, którą obiekty danej klasy mogą wykonywać, np. na rzecz obiektów innych klas. Podobnie jak w przypadku atrybutów, klasa może mieć dowolną liczbę operacji.

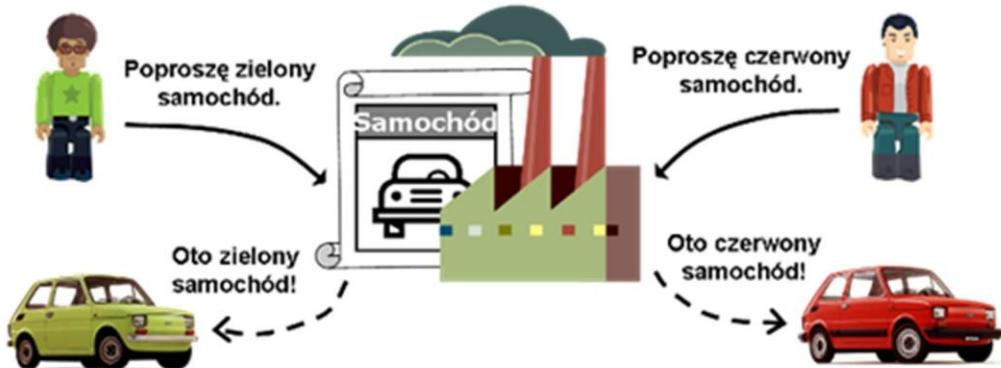
Rysunek 4.8 przedstawia przykładowy diagram reprezentujący klasy w języku UML, odpowiadający wcześniejszym przykładom. Bardziej szczegółowe objaśnienia języka UML w ogólności, a diagramów klas w szczególności będą zawarte w dalszych częściach materiałów. Na razie zwróćmy tylko uwagę, że klasy są w nim reprezentowane poprzez prostokąty opatrzone ich nazwami. Poniżej w odpowiednich przegródkach wymienione są ich atrybuty i operacje. Dodatkowo wszelkie zależności pomiędzy klasami, nazywane ogólnie relacjami, odzwierciedlane są w postaci łączących klas linii. W naszym przykładzie mamy więc klasy reprezentujące samochód i kierowcę połączone odpowiednią relacją. Posiadają one zdefiniowane odpowiednie atrybuty i operacje, zgodne z wcześniejszymi przykładami dotyczącymi obiektów.



Rysunek 4.8: Przykładowy diagram klas

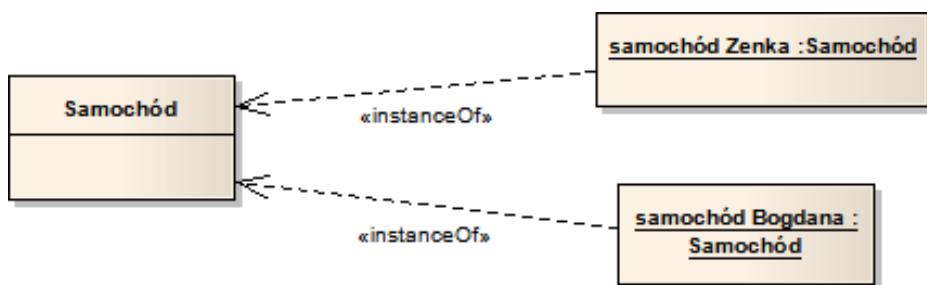
Klasę można także obrazowo porównać do fabryki obiektów, co ilustruje rysunek 4.9. Definicja klasy stanowi swoiste plany konstrukcyjne stosowane w takiej hipotetycznej fabryce. Atrybuty i operacje klasy określają cechy produkowanych obiektów, czyli w naszym przykładzie – konkretne egzemplarze samochodów. Wszystkie wyprodukowane samochody będą zachowywały się tak, jak przewidzieli to inżynierowie i zachowanie to jest niezmienne. O pewnych właściwościach samochodu, określających jego stan zaraz po wyprodukowaniu, może decydować klient zamawiający konkretny egzemplarz. Może on np. określić kolor samochodu. Inne właściwości określone są przez fabrykę, np. przebieg czy

stan baku. Stan początkowy samochodu będzie się oczywiście zmieniał w miarę jego użytkowania przez właściciela.



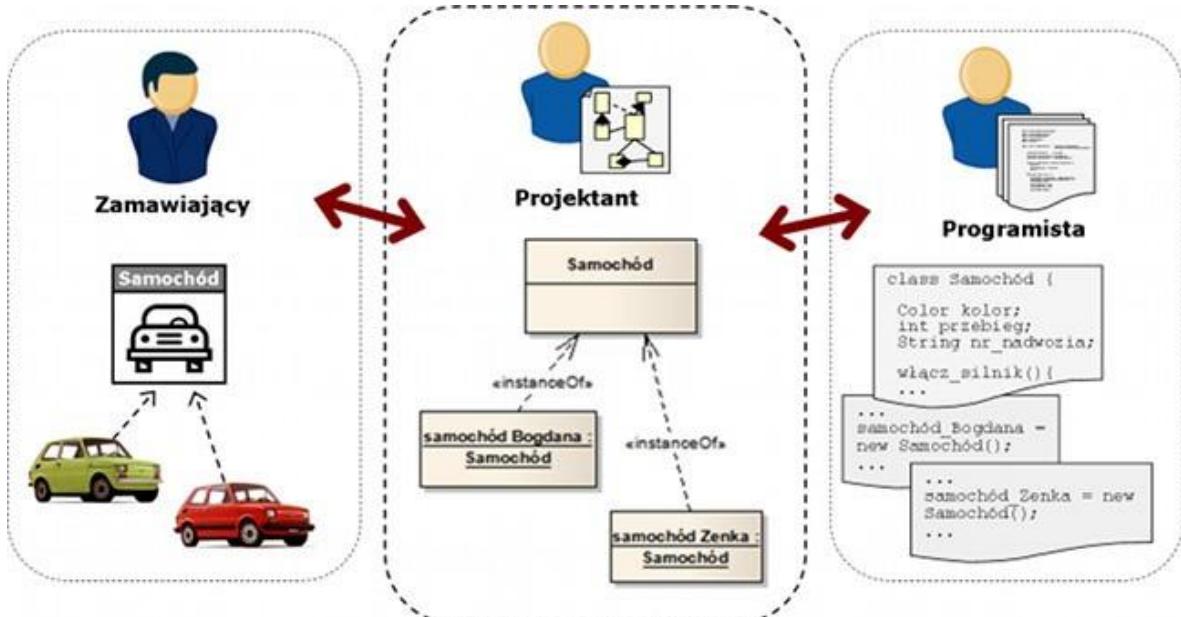
Rysunek 4.9: Klasa jako „fabryka obiektów”

W języku UML obiekty przypisane do danej klasy, nazywane są instancjami (ang. *Instance*) klasy. Rysunek 4.10 przedstawia dwa obiekty będąceinstancjami klasy „Samochód”, na co wskazuje zarówno typ obiektów pokazany w ich nazwie, jak również relacje «instanceOf» łączące obiekty z klasą.



Rysunek 4.10: Obiekty jako instancje klas

Powiedzieliśmy wcześniej, że modele powinny być tworzone z perspektywy konkretnego celu, któremu będą służyć. Różni czytelnicy modeli patrzą na obiekty i ich klasy z różnych perspektyw, w zależności od tego jaka rolę pełnią w procesie tworzenia systemu oprogramowania. Rysunek 4.11 ilustruje możliwość prezentowania klas obiektów na różnych poziomach szczegółowości, odpowiednich dla bieżącego celu i odbiorców tworzonych modeli. Z jednej strony mamy zamawiającego oraz przyszłych użytkowników systemu, dla których klasy odnoszą się do rzeczywistych pojęć z ich dziedziny problemu. Z drugiej strony mamy programistów oraz inne osoby będące ekspertami w kwestiach technicznych związanych z tworzeniem systemów oprogramowania, dla których klasy są podstawowymi jednostkami kodu systemu, które grupują odpowiednie dane i sposoby ich przetwarzania. Odwzorowanie pojęć świata rzeczywistego na formalne elementy technologii informatycznych zapewniają modele oparte o klasy obiektów, tworzone przez analityków i projektantów.



Rysunek 4.11: Modele obiektowe jako wspólny język dla zamawiających, projektantów i programistów

Podsumowując, możemy wyróżnić trzy perspektywy, których można użyć przy tworzeniu modeli, zwłaszcza modeli klas:

Perspektywa pojęciowa. Przyjmując tę perspektywę, modelujemy klasy reprezentujące pojęcia analizowanego środowiska. Model pojęciowy tworzony jest niezależnie od oprogramowania. Zazwyczaj nie ma pełnej odpowiedniości pomiędzy elementami tego modelu a klasami implementacyjnymi.

Perspektywa specyfikacyjna. Perspektywa ta dotyczy bezpośrednio oprogramowania, ale raczej w sensie definiowania typów obiektów niż szczegółów implementacji. Typ może być implementowany przez wiele klas, a klasa może implementować wiele typów. W perspektywie tej unika się zazwyczaj szczegółów związanych z konkretnym językiem programowania czy technologią.

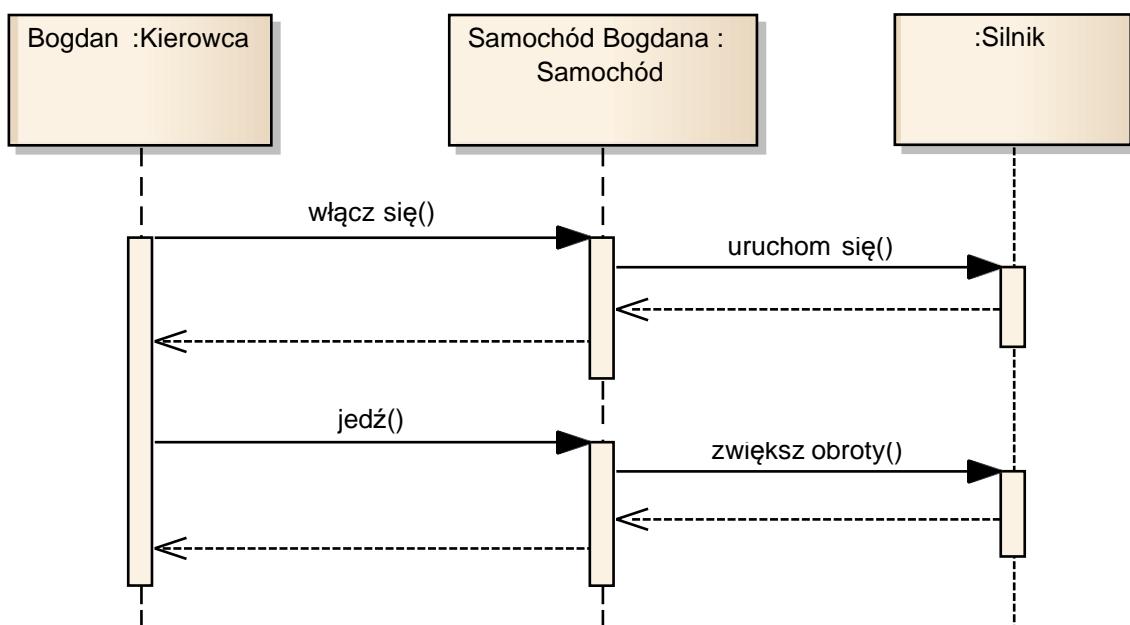
Perspektywa implementacyjna. Perspektywa ta ukazuje wszystkie szczegóły implementacji. Na podstawie modeli implementacyjnych możliwe jest stworzenie kodu oprogramowania.

Wymienione perspektywy nie są formalnie zdefiniowane w języku UML. Są jednak bardzo przydatne podczas tworzenia i analizy modeli, chociaż granice między nimi nie zawsze są jednoznaczne.

4.5. System jako zbiór współpracujących obiektów

Klasy i należące do nich obiekty służą odzwierciedleniu statycznej struktury wybranej dziedziny problemu lub systemu oprogramowania. Obiekty najczęściej nie są jednak bezczynne, lecz wzajemnie na siebie oddziałują. Zbiór współpracujących ze sobą obiektów nazywamy systemem. Oczywistym przykładem systemu jest dowolny system oprogramowania, ale jest to pojęcie znacznie szersze. Działanie każdego systemu – w szczególności systemu oprogramowania – można przedstawić jako wzajemne przesyłanie komunikatów w ramach pewnego zbioru obiektów w ścisłe określonym celu. Jak wiemy, obiekty posiadają zachowanie określone przez swoje klasy, w postaci zbioru usług. Usługi te mogą być wykonywane na prośbę innych obiektów. Prośby o wykonanie usługi oraz ewentualne odpowiedzi nazywamy **komunikatami** (ang. Message). W ramach wykonania usługi, obiekt może operować na wartościach swoich atrybutów, w wyniku czego jego stan może ulec zmianie. Może też przesyłać komunikaty do innych obiektów.

Język UML zawiera szereg różnych diagramów pozwalających odzwierciedlać dynamiczne aspekty modelowanych dziedzin. Przebiegu wymiany komunikatów pomiędzy obiektami możemy przedstawić za pomocą modeli interakcji (ang. Interaction model). Przykład takiego modelu widzimy na diagramie zamieszczonym na rysunku 4.12. Diagram ten zawiera trzy tzw. linie życia, odpowiadające obiektom przedstawionym na górze diagramu („Bogdan”, „Samochód Bogdana” i nienazwany obiekt klasy „Silnik”). Pomiędzy liniami przebiegają komunikaty wyrażone w formie strzałek. Komunikaty te czytane w kolejności od góry do dołu tworzą sekwencję interakcji między obiektami, w tym wypadku opisującą proces uruchamiania samochodu i rozpoczęcia jazdy. W podanym przykładzie kierowca najpierw przesyła do samochodu polecenie, aby się włączył. Powoduje to z kolei przesłanie przez samochód polecenia do silnika, aby się uruchomił. Następnie kierowca rozpoczęta jazdę, czyli wysyła do samochodu polecenie „jedź” (np. w rzeczywistości naciska pedał gazu). Wymaga to zwiększenia obrotów silnika, co samochód wysyła odpowiednie polecenie do silnika. Na diagramie widzimy zarówno prośby o wykonanie usługi (pełne strzałki), jak i odpowiedzi oznaczające zakończenie wykonywania usługi przez obiekt (przerywane strzałki). Bardziej szczegółowe omówienie diagramów interakcji oraz innych diagramów języka UML używanych do odzwierciedlania dynamiki systemu znajduje się w jednym z następnych rozdziałów podręcznika.



Rysunek 4.12: Przykładowy diagram interakcji

Warto zauważyć, że wpływ na to jak zostanie wykonana określona usługa, mają trzy czynniki.

- Aktualny stan obiektu w momencie odebrania komunikatu od innego obiektu. W przypadku samochodu, usługa „włącz_silnik” będzie miała inny efekt przy pełnym baku a inny, gdy baki będzie pusty.
- Parametry komunikatu. Parametry (ang. *parameter*) komunikatu, to lista wartości lub obiektów przekazywanych adresatowi komunikatu w celu sterowania sposobem wykonania usługi. Dla przykładu, parametrem usługi „zatankuj” może być ilość paliwa.
- Stan innych obiektów, z usług których korzysta obiekt w celu wykonania swojej usługi. W celu wykonania usługi „włącz się”, obiekt klasy „Samochód” wysyła do obiektu klasy „Silnik” komunikat „uruchom się”. Gdy stan silnika nie pozwala na wykonanie tej usługi, również usługa „włącz się” nie przyniesie spodziewanego efektu.

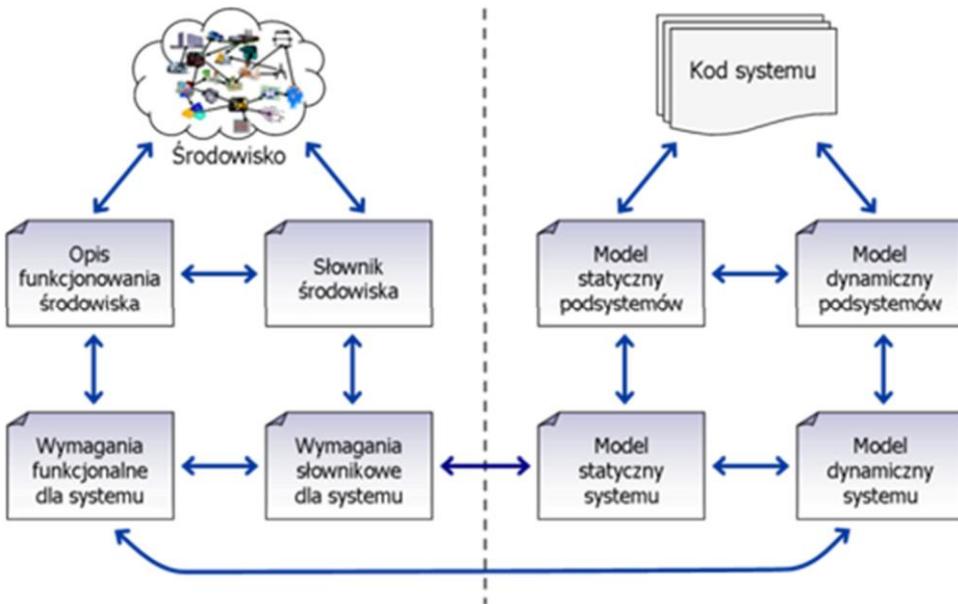
Zauważmy też, że w podanym przykładzie kierowca chcąc uruchomić silnik, wysyła jedynie odpowiedni komunikat do samochodu i oczekuje na rezultat. Nie wie on, w jaki sposób samochód realizuje usługę „włącz się” i jakie inne obiekty są w to zaangażowane. Samochód jest dla niego czarną skrzynką z przyciskami do wywoływania określonych usług, co ilustruje rysunek 4.13. Można powiedzieć że z punktu widzenia kierowcy uruchomienie samochodu stanowi tzw. proces, jako że znane jest mu tylko jego wejście (co zrobił, żeby go rozpoczęć) i wyjście (co uzyskał w jego efekcie). Podejście procesowe jest często przydatne przy wytwarzaniu oprogramowania, jako że dobrze zaprojektowany obiekt udostępnia innym obiektom tylko te usługi lub właściwości, które są im potrzebne. Ważne jest jednak, aby udostępniane usługi (nazwy operacji w klasie) były dobrze i jednoznacznie opisane, tak, aby wywołujący te usługi wiedział co otrzyma w wyniku ich realizacji.



Rysunek 4.13: Obiekt jako czarna skrzynka z przyciskami

4.6. Modele w procesie inżynierii oprogramowania

Podsumowując niniejszy rozdział warto zauważyć, że w procesie budowy systemu oprogramowania przydatnych jest szereg modeli na różnym poziomie szczegółowości. Modele te pozwalają połączyć świat zamawiających system ze światem jego budowniczych (deweloperów, programistów). Można powiedzieć, że modele tworzą ścieżkę prowadzącą od opisu środowiska (dziedziny problemu) do kodu programu, który wspiera funkcjonowanie tego środowiska. Ścieżka ta została zilustrowana na rysunku 4.14. Warto zauważyć, że odpowiada ona poszczególnym dyscyplinom inżynierii oprogramowania omówionym w module I.



Rysunek 4.14: Modele na ścieżce od opisu środowiska do kodu

Po lewej stronie rysunku 4.14 znajdują się modele będące produktami czynności analitycznych. Po prawej znajdują się modele czynności projektowych (syntezy), prowadzące do powstania kodu systemu. Wszystkie te modele powiązane są zależnościami symbolizowanymi przez strzałki. Niektóre modele są tworzone oddzielnie, jednak uzupełniają się wzajemnie; niektóre są tworzone poprzez uszczegółowienie tych stworzonych wcześniej. Powiązania wszystkich modeli definiują ścieżkę umożliwiającą przejście od analizowanego środowiska do kodu systemu oprogramowania, który będzie odzwierciedlał to środowisko zgodnie z oczekiwaniemi zamawiającego systemu.

Dwa modele wynikające bezpośrednio z analizy środowiska opisujące zarówno jego dynamikę jak i strukturę, możemy zaliczyć do perspektywy pojęciowej. Typowo do tego celu można użyć odpowiednio diagramów czynności i diagramów klas. Mogą one być przekształcone w modele wymagań, które są podstawowymi składnikami specyfikacji wymagań. W tym celu znowu używa się odpowiednio diagramów przypadków użycia i diagramów klas. Na tej podstawie, budowane są modele systemu, stając się jego projekt architektoniczny. Projekt taki określa, z jakich elementów należy zbudować system, aby najlepiej realizował postawione wymagania. Do modelowania statyki systemu używa się diagramów komponentów, a do dynamiki – diagramów interakcji. Modele wymagań oraz modele systemu możemy określić jako perspektywę specyfikacyjną. Modele podsystemów są uszczegółowieniem projektu architektonicznego – ukazują szczegóły budowy i działania elementów składowych systemu. Modele podsystemów służą bezpośrednio do stworzenia kodu, zaliczamy je więc do perspektywy implementacyjnej. Do modelowania statyki podsystemów używa się diagramów klas, a do ich dynamiki – diagramów interakcji. Wszystkie wspomniane typy diagramów zostaną szczegółowo omówione w dalszych rozdziałach niniejszego modułu.

Zadania

Zadanie 1

Mamy zadane obiekty (konto osobiste Jana Kowalskiego, konto firmowe firmy XYZ, rachunek do konta, ...). Zaproponuj klasy obiektów oraz zbuduj hierarchię gen-spec.

Zadanie 2

Mamy zadaną dziedzinę problemu i perspektywę. Podziel podany zbiór cech na cechy istotne i nieistotne. Pogrupuj cechy istotne w klasy i stwórz model klas bazujący na tych cechach.

Słownik pojęć

Atrybut

Właściwość obiektów reprezentowanych przez klasę.

Klasa

Opis grupy obiektów, które mają taki sam zestaw właściwości oraz sposób zachowania.

Komunikat

Prośba obiektu o wykonanie usługi przez jakiś obiekt lub odpowiedź na tą prośbę.

Model

Reprezentacja określonej rzeczywistości, która pomaga w zrozumieniu tej rzeczywistości.

Obiekt

Element rzeczywistości, który posiada trzy główne cechy: **tożsamość, stan i zachowanie**.

Operacja

Podstawowy element sposobu zachowania obiektów danej klasy.

Co trzeba zapamiętać

Na czym polega modelowanie

Podstawą modelowania jest tworzenie modeli, czyli uproszczonych reprezentacji rzeczywistości. Celem modelowania może być np. testowanie właściwości fizycznych obiektów przed ich wykonaniem, wizualizacja i komunikacja z klientami, czy też redukcja złożoności modelowanego zagadnienia. Modelowanie stosuje różne zasady radzenia sobie ze złożonością rzeczywistości, a w szczególności – zasadę abstrakcji. Zasada abstrakcji jest realizowana m.in. przez techniki klasyfikacji i generalizacji.

Język UML

Unified Modelling Language (ujednolicony język modelowania) powstał na początku lat 90-tych XX wieku i jest cały czas rozwijany. UML jest językiem graficznym, za pomocą którego można tworzyć modele niezbędne w całym procesie budowy systemu informatycznego. Można go również wykorzystywać do modelowania rzeczywistości niekoniecznie powiązanej z systemem informatycznym.

Modelowanie obiektowe

Najbardziej rozpowszechnionym współcześnie paradigmatem modelowania jest modelowanie obiektowe. Zgodnie z nazwą opiera się ono na obiektach, czyli wyodrębnionych elementach rzeczywistości istotnych z perspektywy tworzonego modelu. Każdy obiekt stanowi osobny byt wyraźnie

wyodrębniony z całości modelowanej dziedziny. Obiekty mogą zatem reprezentować różnego rodzaju przedmioty, osoby, zdarzenia, procesy lub inne twory niematerialne występujące w danym środowisku. Modelowanie obiektowe polega na znajdowaniu interesujących nas obiektów rzeczywistych w danej dziedzinie, opisywaniu struktury i sposobu działania tych obiektów, klasyfikacji i generalizacji obiektów, znajdowaniu powiązań między nimi oraz opisywaniu dynamicznych aspektów współpracy pomiędzy obiektami.

5. Modelowanie struktury systemu

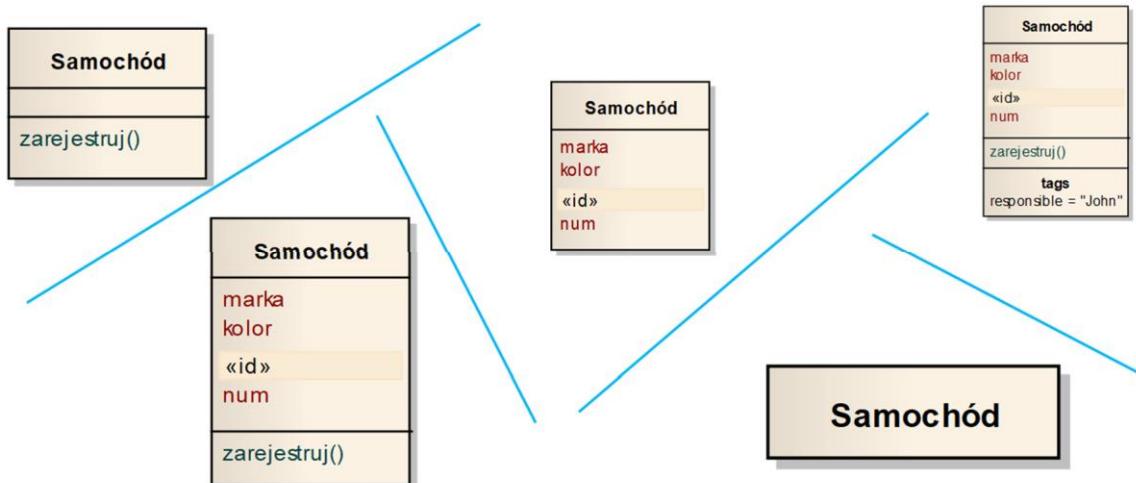
Każdy system – czy to system biznesowy, czy system informatyczny – posiada specyficzną dla siebie strukturę. Struktura określa elementy składowe danego systemu oraz zespół relacji (powiązań) między tymi elementami. Model struktury definiuje statyczne, czyli zasadniczo niezmienne w czasie aspekty modelowanego systemu. Typowo obejmuje to kategoryzację występujących w systemie elementów, a następnie określenie konkretnego układu łączących je zależności. W inżynierii oprogramowania modele struktury mają zastosowanie w opisie szeregu różnych aspektów budowanego systemu oprogramowania. Najbardziej elementarnym zastosowaniem jest zdefiniowanie słownika pojęć dla zadanej dziedziny problemu. Ogólnie, oznacza to odzwierciedlenie w modelu elementów rzeczywistości nie związanych bezpośrednio z programowaniem, czyli opisu strukturalnych aspektów wiedzy z danej dziedziny. Słownik może zostać rozwinięty w model danych, który określa format przetwarzanych danych oraz strukturę relacji między elementami danych. Podczas projektowania systemu, bardzo ważne jest stworzenie szczegółowego modelu struktury samego kodu. Strukturę tą można modelować na różnych poziomach abstrakcji. Na poziomie ogólnym możemy definiować komponenty lub podsystemy, a na poziomie szczegółowym – konkretne klasy odpowiadające elementarnym jednostkom kodu w konkretnym języku programowania.

Język UML oferuje szereg typów modeli służących do modelowania struktury. Różne typy dostosowane są do różnych celów, którym mają służyć. Najbardziej podstawowym i najczęściej używanym typem modelu jest **model klas** (ang. Class Model). Jest to uniwersalny model, używany do odzwierciedlania struktury wynikowej z klasyfikacji elementów i uporządkowania łączących je zależności. Diagramy klas mogą dotyczyć opisu rzeczywistości, definicji struktur danych lub projektu szczegółowej struktury kodu. Drugim często używanym typem modeli jest **model komponentów** (ang. Component Model). Model komponentów znajduje zastosowanie przede wszystkim w definiowaniu architektur logicznych systemów oprogramowania. Ma on na celu przedstawianie struktur systemów, które są na tyle złożone, że konieczne jest opisanie ich na wyższym poziomie abstrakcji niż poziom klas. Dzięki temu możliwe jest istotne ograniczenie liczby prezentowanych szczegółów, a tym samym – lepsze panowanie nad złożonością całości. Do modelowania architektur fizycznych służy natomiast **model wdrożenia** (ang. Deployment Diagram), również nazywany diagramem montażu. Diagramy wdrożenia pozwalają na zdefiniowanie fizycznych elementów budowanego systemu informatycznego (np. serwerów, urządzeń mobilnych, maszyn wirtualnych) oraz połączeń między nimi (np. sieci lokalnych, sieci radiowych). Umożliwiają one również pokazanie zależności pomiędzy strukturą logiczną a fizyczną podmiotu modelowania.

Język UML zawiera również inne, rzadziej używane modele służące do modelowania struktury. Są to **model obiektów** (ang. Object Model), **model pakietów** (ang. Package Model) oraz **model składowych** (ang. Composite Structure Model). Ich omówienie wykracza poza zakres tego podręcznika. W następnych sekcjach przedstawimy składnię języka UML dla trzech podstawowych modeli struktury oraz wyjaśnimy ich semantykę (znaczenie) w sposób ogólny. O zastosowaniu i znaczeniu modeli struktury w modelowaniu wymagań oraz projektowaniu oprogramowania mówimy w module III (Podstawy inżynierii wymagań i projektowania oprogramowania).

5.1. Model klas

Diagramy klas stanowią najbardziej uniwersalną i prawdopodobnie najbardziej rozpowszechnioną formę modelowania struktury. Podstawowym elementem diagramu klas jest oczywiście klasa, czyli uogólnienie zbioru obiektów o tym samym sposobie zachowania i atrybutach (patrz poprzedni rozdział). Klasa definiuje zatem grupę podobnych obiektów, które zostały uznane za warte wspólnego reprezentowania na potrzeby danego modelu. Na rysunku 5.1 przedstawione zostały różne dopuszczalne reprezentacje graficzne klas w języku UML. Elementem wspólnym dla wszystkich tych reprezentacji jest przedstawienie klasy jako prostokąta zawierającego jej nazwę.



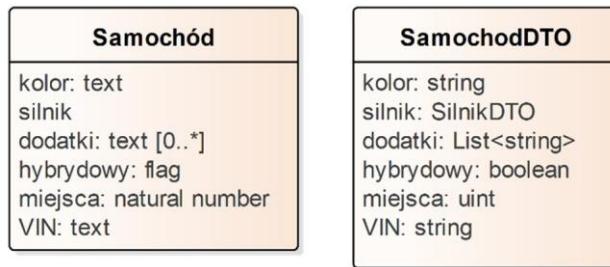
Rysunek 5.1: Reprezentacje klas o różnym poziomie szczegółowości

Najprostszą reprezentacją klasy jest zatem prostokąt zawierający wyśrodkowaną nazwę klasy. Ikona klasy może zawierać kilka przegródek, oddzielonych liniami poziomymi. Najczęściej spotykane są reprezentacje składające się z jednej, dwóch lub trzech przegródek. Pierwsza przegróda od góry zawiera nazwę klasy, druga zwiera jej atrybuty, a trzecia – operacje. Język UML dopuszcza tworzenie większej liczby przegródek w zależności od potrzeb. Możliwe jest umieszczenie atrybutów lub operacji w kilku różnych przegródkach, zmiana kolejności przegródek (oprócz przegródki z nazwą) oraz dodawanie przegródek przeznaczonych na inne elementy modelowania. W przykładzie na rysunku 5.1 reprezentacja w prawym górnym rogu zawiera dodatkową przegródkę na tzw. metki (ang. Tag), które określają meta-atributy klasy (np. informacje o wersji klasy czy osobie odpowiedzialnej). Na danym diagramie klas, poszczególne przegródki mogą być ujawnione (zwinięte) lub nie. Umożliwia to tworzenie diagramów o różnym poziomie szczegółowości. Dla przykłady, na jednym diagramie możemy pokazać wszystkie szczegóły kilku klas, a na innym – wiele klas bez ujawnionych informacji o ich atrybutach czy operacjach.

Jak już wiemy, klasa stanowi wzorzec dla reprezentujących ją obiektów, opisując sposób zachowania oraz własności wspólne dla nich wszystkich. Język UML dostarcza dwóch podstawowych składników opisu klasy: atrybutów i operacji. Elementy te odpowiadają typowej strukturze opisu klas w obiektowym podejściu do modelowania. Oczywiście, taka struktura modelu klas odpowiada strukturze kodu w obiektowych językach programowania, co znacząco ułatwia projektowanie kodu. Atrybuty i operacje odpowiadają polom i metodom w kodzie, jednak mogą mieć też interpretacje zupełnie nie związane z programowaniem. Dzięki temu, możliwe jest tworzenie modeli klas dla różnych odbiorców i pokazujących różne aspekty rzeczywistości (dziedziny problemu) oraz budowanego systemu oprogramowania.

Przykłady notacji języka UML dla atrybutów przedstawione są na rysunku 5.2. **Atrybuty** definiują własności opisujące obiekty danej klasy, którym da się przypisać konkretną wartość. Posiadają one

konkretnie nazwy oraz mogą mieć określony typ. Przykłady notacji języka UML dla operacji widzimy na rysunku 5.3. **Operacje** reprezentują możliwe sposoby zachowania modelowanych obiektów poprzez określenie możliwych do wykonania funkcji lub procedur. Podobnie jak w przypadku atrybutów, operacje posiadają zdefiniowaną nazwę oraz mogą posiadać typ zwracanego rezultatu. Operacje odróżniane są od atrybutów poprzez podanie parametrów umieszczonych w nawiasach. W najprostszym przypadku lista parametrów może być pusta, jednak zawsze musimy po nazwie operacji umieścić nawiasy. Notacja parametrów operacji jest identyczna jak notacja dla atrybutów (patrz dalej), przy czym parametry rozdzielone są przecinkami. Dla przykładu, operacja „ObliczCene” na rysunku 5.3 zawiera dwa parametry o nazwach „znizka” oraz „podatek”.



Rysunek 5.2: Przykładowe klasy z pokazanymi atrybutami

Zarówno atrybuty jak i operacje reprezentowane są tekstowo i umieszczone w odpowiednich przegródkach ikony klasy (zazwyczaj – osobno operacje i osobno atrybuty), co ilustrują rysunki 5.2 i 5.3. Po **nazwie** elementu (atrybutu lub operacji) umieszczany jest po dwukropku jego **typ** (typ danych dla atrybutu lub typ zwracanego rezultatu dla operacji). Zwróćmy uwagę na to, że język UML nie definiuje dopuszczalnych typów dla atrybutów i operacji. Typy nadawane elementom klas zależą od zastosowania, zatem nie byłoby wskazane ograniczanie możliwości w tym zakresie. Dla przykładu, rysunek 5.2 pokazuje dwie klasy na różnym poziomie opisu. Klasa po lewej stronie stanowi fragment opisu dziedziny i używa terminologii zrozumiałej dla klienta (eksperta w danej dziedzinie problemu). Klasa po prawej stronie (tu: klasa przedstawiająca obiekt transferu danych) jest klasą reprezentującą fragment kodu. Dlatego też używane są typy charakterystyczne dla konkretnego języka programowania. Typy mogą być określane jako typy elementarne (np. „tekst”, „liczba”, „uint”) lub mogą odwoływać się do innych klas (np. „SilnikDTO”). Wynika to z tego, że klasy też są typami danych i mogą być odpowiednio używane jako takie. Zauważmy ponadto, że w miarę możliwości staramy się zachować zgodność między modelem dziedziny a modelem kodu.



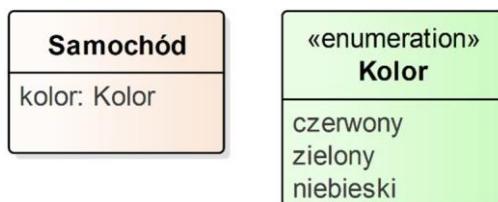
Rysunek 5.3: Przykładowe klasy z pokazanymi operacjami

W przykładzie na rysunku 5.2 możemy zauważać dodatkowe oznaczenie dla jednego z atrybutów – „*dodatki*”. Oznaczenie to określa tzw. **krotność** (ang. Multiplicity) lub inaczej – liczność. Krotność dla atrybutów ujmujemy w nawiasy kwadratowe i podajemy jako zakres, np. [1..5] oznacza zakres od 1 do 5. Znak '*' oznacza dowolną wartość, czyli zakres [0..*] oznacza zakres od zera do nieskończoności. W naszym przykładzie, krotność dla atrybutu „*dodatki*” oznacza, że samochody (obiekty klasy

„Samochód”) mogą zawierać dowolną liczbę dodatków. W przypadku, jeśli klasa określa model kodu, krotność może odpowiadać liście lub tablicy elementów. Zauważmy, że w naszym przykładzie, klasa „SamochodDTO” posiada jawnie określony typ listowy. Alternatywnie, moglibyśmy podać krotność jak dla klasy „Samochód”, która podczas zamiany modelu w kod (generacji kodu) mogłaby być zamieniona na odpowiedni typ (listę lub tablicę).

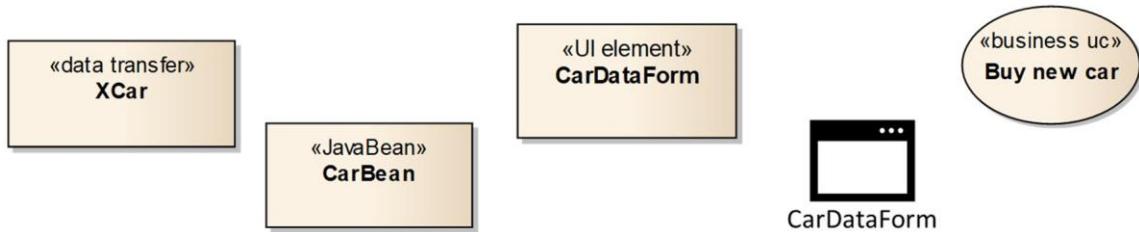
Dodatkowym oznaczeniem składników klasy, używanym najczęściej podczas modelowania kodu, jest **widoczność**. Widoczność możemy określić zarówno dla atrybutów, jak i operacji. Rodzaje widoczności w języku UML odpowiadają najczęstszym rodzajom widoczności w obiektowych językach programowania. Oznaczenie widoczności poprzedza nazwę atrybutu lub operacji. Widoczności publiczna oznacza możliwość dostępu (np. odczytu) do danego elementu przez obiekty innych klas oraz inne obiekty danej klasy. Oznaczana jest ona znakiem „+”. Widoczność prywatna oznacza możliwość dostępu do danego elementu tylko w ramach kodu danej klasy (brak dostępu przez obiekty zewnętrzne). Oznaczana jest ona znakiem „-“. Widoczność chroniona (ang. protected) oznacza możliwość dostępu do danego elementu w ramach kodu danej klasy oraz klas specjalizujących od niej. Oznaczana jest ona znakiem „#”. Odpowiednie przykłady widoczności można znaleźć na rysunkach 5.2 i 5.3 (w klasach SamochodDTO i MSamochod).

Język UML umożliwia również definiowanie własnych typów prostych, za pomocą tak zwanych **typów wyliczeniowych** (ang. enumeration). Umożliwiają one zdefiniowanie typu poprzez określenie listy wartości dla tego typu. Typy wyliczeniowe reprezentowane są w sposób bardzo zbliżony do klas. Nad nazwą typu wyliczeniowego umieszczamy dodatkowe oznaczenie «enumeration», a w drugiej przegródce podajemy listę dopuszczalnych wartości. Przykładowy typ wyliczeniowy wraz z przykładem jego zastosowania został przedstawiony na rysunku 5.4. W przykładzie tym, atrybut „kolor” posiada typ wyliczeniowy „Kolor” i może przyjmować jedną z trzech wartości zdefiniowanych tym typem.



Rysunek 5.4: Przykład typu wyliczeniowego wraz z zastosowaniem

Oznaczenie wykorzystywane w notacji typów wyliczeniowych nosi nazwę **stereotypu**. Stereotypy możemy nadawać dowolnym elementom języka UML omawianym w tym i kolejnych rozdziałach, czyli w szczególności również klasom. Umożliwiają on nadanie dodatkowego (np. bardziej precyzyjnego) znaczenia danemu elementowi modelu. W ten sposób możemy tworzyć nowe, własne rodzaje elementów, które pochodzą od elementów standardowych. Szereg istniejących elementów języka UML również używa oznaczenia za pomocą stereotypów. Przykładem są omówione powyżej typy wyliczeniowe oraz interfejsy, które będą omówione w dalszej części rozdziału. Stereotyp jest określany poprzez podanie jego nazwy w nawiasach kątowych (tzw. nawiasy francuskie). Przykłady stereotypów przedstawione są na rysunku 5.5. Stereotyp może być powiązany z określeniem nowego kształtu dla danego elementu modelu. Dla przykładu, na rysunku, stereotypowi «UI Element» został przypisany kształt elementu ekranowego (okienka). Po nadaniu elementowi tego stereotypu, kształt ikony dla tego elementu może się zmienić ze standardowego na przypisany do stereotypu. Zwróćmy zatem uwagę, że mechanizm stereotypów pozwala rozszerzać notację języka UML w sposób praktycznie nieograniczony.

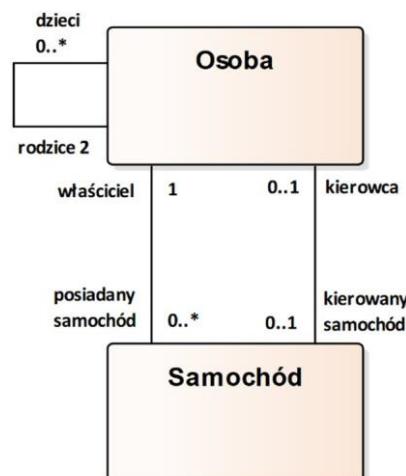


Rysunek 5.5: Przykładowe elementy języka UML z nadanymi stereotypami

Zgodnie z definicją modelowania struktury, diagramy klas oprócz klas powinny zawierać również różnych rodzajów relacji między klasami. Najczęściej używanymi relacjami na diagramach klas są relacje asocjacji oraz relacje generalizacji. Relacje asocjacji posiadają dwa bardziej wyspecjalizowane warianty – relacje agregacji i relacje kompozycji. Często są również stosowane relacje zależności oraz relacje realizacji.

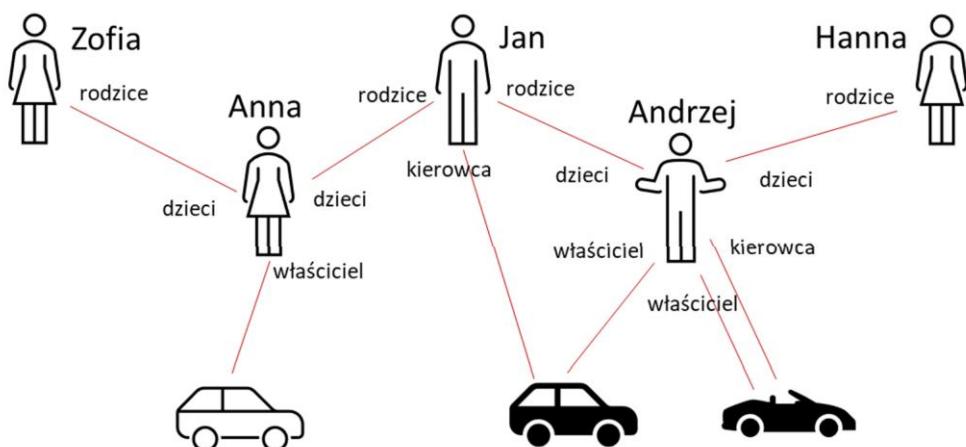
Relacja asocjacji definiuje możliwe relacje między obiektami klas (w szczególności – relacje między obiektami tej samej klasy). Obiekty na końcach asocjacji mogą mieć zdefiniowane określone role. Asocjacje oznaczamy linią łączącą odpowiednie klasy. Role klas możemy zapisać jako nazwy końców asocjacji. Końce asocjacji mogą również definiować krotneści dla uczestniczących w relacji obiektów. Krotność określa, ile obiektów danej klasy może potencjalnie być w relacji z określonym obiektem klasy przeciwniej. Krotneści oznaczane są w tej samej sposób jak dla atrybutów (bez nawiasów kwadratowych) – podajemy zakres możliwych wartości. Warto zauważyć, że asocjacja może odzwierciedlać również powiązania między różnymi obiektami tej samej klasy. W takiej sytuacji, asocjacja łączy klasę z samą sobą za pomocą „zawiniętej” linii.

Przykłady asocjacji przedstawia rysunek 5.6. Jak widzimy, między dwoma klasami możemy zdefiniować kilka asocjacji, które określają różne role dla relacji między obiektami. Dана osoba może występować np. w roli kierowcy lub w roli właściciela dla danego pojazdu. Może również być jednocześnie kierowcą i właścicielem. Zgodnie z diagramem, konkretna osoba może posiadać dowolnie dużo (od 0 do wielu) samochodów oraz może kierować (w danej chwili) maksymalnie jednym samochodem (krotność od zera do jednego). Z kolei samochód może mieć dokładnie jednego właściciela oraz maksymalnie jednego kierowcę. Zwrócić uwagę na to, że krotność „dokładnie jeden” możemy oznaczyć zarówno jako „1..1” jak i po prostu jako „1”. Ponadto – z założenia – brak określenia krotności dla danego końca asocjacji (lub np. dla atrybutu) oznacza krotność „1”.



Rysunek 5.6: Przykładowe zastosowania asocjacji

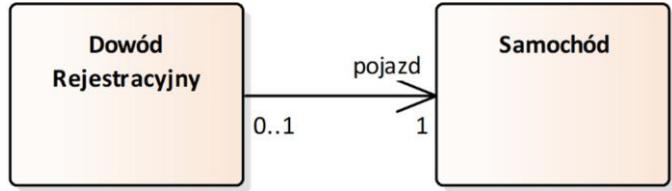
Na rysunku 5.6 widzimy również asocjację dotyczącą tylko klasy „Osoba” (asocjacja „zawinięta”). Zgodnie w tym fragmentem modelu, dana osoba ma dokładnie dwóch rodziców oraz może mieć dowolnie dużo dzieci. Znaczenie takich asocjacji wyjaśnia dodatkowo rysunek 5.7. Widzimy tu pięć obiektów klasy „Osoba” („Zofia”, „Jan”, „Hanna”, „Anna” oraz „Andrzej”). Anna oraz Andrzej posiadają po dwóch rodziców (wspólnym dla nich rodzicem jest Jan). Zauważmy, że Jan, Zofia i Hanna nie mają określonych rodziców. Nie jest to zgodne z modelem z rysunku 5.6, gdyż zgodnie z nim każda osoba powinna mieć dokładnie dwóch rodziców. W tym wypadku oznacza to konieczność modyfikacji diagramu klas poprzez zmianę krotności na 0..2 (możliwy brak określenia jednego lub obydwu rodziców danej osoby). Na rysunku 5.7 widzimy również ilustrację znaczenia dwóch relacji między klasami Osoba i Samochód. Andrzej jest właścicielem dwóch czarnych samochodów. Właścicielem białego samochodu jest Anna. W konkretnej sytuacji na rysunku, biały samochód nie ma żadnego kierowcy. Z kolei, jeden z czarnych samochodów jest kierowany przez Jana, który nie jest jego właścicielem. Drugi czarny samochód jest natomiast kierowany przez swojego właściciela, czyli Andrzeja. Te relacje między obiekty są zgodne z krotnościami ustalonymi przez diagram klas na rysunku 5.6.



Rysunek 5.7: Ilustracja relacji między obiekty definiowanymi przez asocjacje

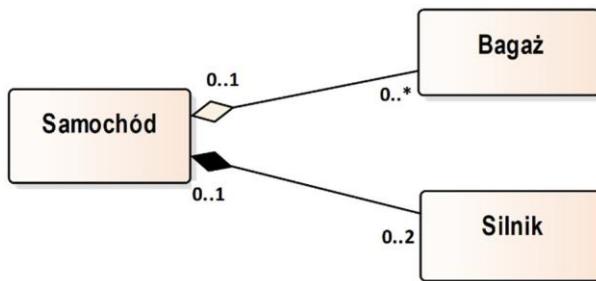
Specjalną właściwością asocjacji jest tzw. **nawigowalność** (skierowanie asocjacji). Nawigowalność oznacza możliwość efektywnego osiągnięcia obiektów jednej klasy przez obiekty drugiej klasy. Inaczej mówiąc, obiekty jednej klasy „wiedzą” o istnieniu odpowiednich obiektów innej klasy i mogą korzystać z ich usług, np. odczytywać ich atrybuty lub uruchamiać ich operacje (dotyczy tylko elementów o widoczności publicznej). Nawigowalność asocjacji oznaczamy poprzez dodanie grotu strzałki na końcu asocjacji. Zwróćmy uwagę, że strzałki możemy umieścić na wszystkich końcach asocjacji, czyli na obydwu końcach w przypadku asocjacji między dwoma klasami. Przy okazji warto wspomnieć, że asocjacje mogą łączyć jednocześnie więcej klas.

Przykład asocjacji nawigowej jest przedstawiony na rysunku 5.8. Z rysunku wynika, że obiekty klasy „Dowód rejestracyjny” mają dostęp do danych odpowiadających im samochodów, natomiast obiekty klasy „Samochód” nie mają zdefiniowanego dostępu do danych przypisanych do nich dowodów rejestracyjnych.



Rysunek 5.8: Przykład asocjacji nawigowej

Specjalnym rodzajem asocjacji są relacje **agregacji**. Stosujemy je, kiedy potrzebne jest odzwierciedlenie związku między grupą, a jej elementem lub elementami. Relacja agregacji jest zatem relacją grupowania, na przykład grupowania składników (części) pewnej całości. Relacja agregacji stosuje wszystkie elementy notacji asocjacji i dodatkowo jest wyróżniana poprzez umieszczenie małej ikony rombu po jednej stronie relacji. Romb umieszczany jest przy tej klasie, która stanowi całość. Przykład relacji agregacji widzimy na rysunku 5.9. Klasa „Samochód” stanowi element grupujący (całość), a klasa „Bagaż” definiuje elementy grupowane (składniki).



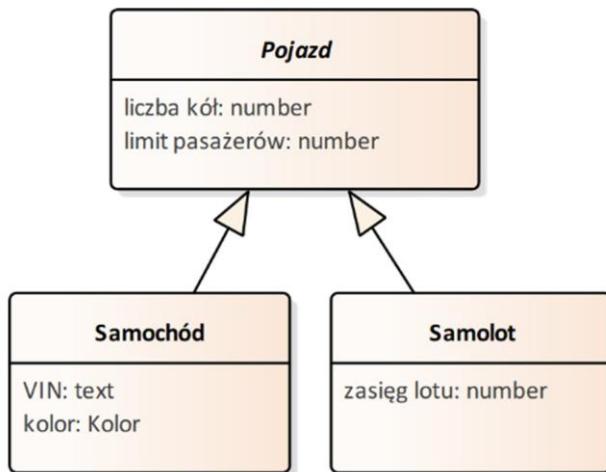
Rysunek 5.9: Przykład zastosowania agregacji i kompozycji

Rysunek 5.9 zawiera również relację między klasami „Samochód” i „Silnik”. Jest ona oznaczona podobnie jak relacja agregacji, lecz ikona rombu jest wypełniona. Jest to relacja **kompozycji**, która stanowi „silniejszą” wersję relacji agregacji. Reprezentuje ona związek pomiędzy całością a jej integralnymi częściami. Istotną cechą relacji kompozycji jest zasada, że obiekty odpowiadające składnikom nie mogą być zawarte w więcej niż jednym obiekcie odpowiadającym całości. Inaczej mówiąc – w relacji kompozycji składniki nie mogą być dzielone między różne całości. Dlatego też, krotność po stronie całości nie może być większa niż 1. W naszym przykładzie na rysunku 5.9, każdy silnik może być zawarty w co najwyżej 1 samochodzie (ew. może nie być zawarty w żadnym samochodzie). Przy okazji, warto też zwrócić uwagę na krotność po drugiej stronie tej relacji. Wynika z niej, że samochód może posiadać maksymalnie 2 silniki (np. silnik spalinowy i elektryczny). Dodatkowym wyróżnikiem relacji kompozycji jest to, że składniki zazwyczaj powstają i kończą swoje istnienie razem z całością, do której należą – czas życia składników jest ograniczony czasem życia całości (np. silniki samochodu są złomowane razem z samochodem). Ten aspekt relacji kompozycji jest szczególnie istotny podczas modelowania struktury kodu, gdyż definiuje proces tworzenia i destrukcji obiektów składowych. Warto jednak podkreślić, że różnica między relacjami asocjacji, agregacji i kompozycji jest dosyć płynna. Ich stosowanie często zależy od konkretnej dziedziny problemu, kontekstu zastosowania oraz celu danego modelu.

Bardzo ważnym aspektem modelowania obiektowego jest tworzenie taksonomii klas. W obiektowych językach programowania umożliwia to mechanizm dziedziczenia. W języku UML taksonomie tworzymy za pomocą relacji generalizacji-specjalizacji (w skrócie – **generalizacji**). Relacja generalizacji określa zależność pomiędzy klasami obiektów bardziej ogólnych i klasami obiektów bardziej specjalizowanych. Klasa specjalizowana „dziedziczy” po klasie ogólnej wszystkie jej atrybuty, operacje oraz relacje

(asocjacje, agregacje, kompozycje, generalizacje). Oznacza to, że obiekty klasy specjalizowanej posiadają wszystkie elementy zdefiniowane w danej klasie, oraz dodatkowo – wszystkie elementy pochodzące z klasą ogólnej. Język UML dopuszcza również możliwość generalizacji wielobazowej. Klasy mogą zatem specjalizować jednocześnie kilka klas ogólnych. Warto jednak zwrócić uwagę na to, że niektóre języki programowania nie dopuszczają dziedziczenia wielobazowego. W takiej sytuacji należy oczywiście unikać stosowania generalizacji wielobazowej w modelu kodu.

Relacje generalizacji oznaczamy strzałką z dużym grotem w kształcie trójkąta. Strzałka zwrócona jest od klasy specjalizowanej do klasy ogólnej (wskazuje na klasę ogólną). Przykłady zastosowania relacji generalizacji widzimy na rysunku 5.10. Od klasy ogólnej „Pojazd” specjalizują dwie klasy – „Samochód” oraz „Samolot”. Z relacji tych wynika, że obiekty klasy Samochód będą posiadać 4 atrybuty (liczba kół, limit pasażerów, VIN i kolor), a obiekty klasy Samolot – trzy (liczba kół, limit pasażerów, zasięg lotu).



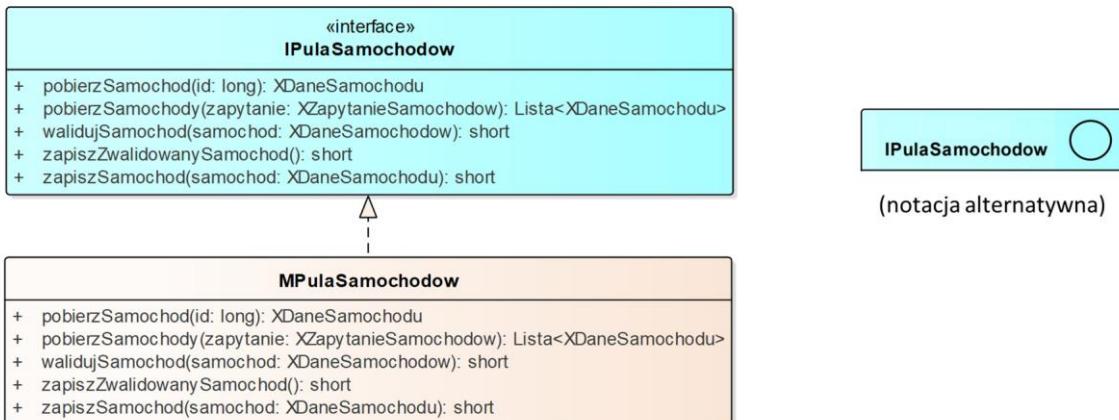
Rysunek 5.10: Przykład zastosowania generalizacji i klas abstrakcyjnych

W relacjach generalizacji często spotykamy klasy na tyle ogólne, że nie posiadają z zasady żadnych obiektów będących ich instancjami. Takie klasy nazywamy **klasami abstrakcyjnymi**. Używa się ich przede wszystkim w celu przedstawiania cech wspólnych innych klas. Reprezentacja klasy abstrakcyjnej różni się od zwykłej klasy w ten sposób, że jej nazwa zapisana jest kursywą. W przykładzie na rysunku 5.10 klasą abstrakcyjną jest klasa Pojazd. Oznacza to, że w rzeczywistości opisywanej tym diagramem nie istnieją pojazdy jako takie. Konkretny pojazd musi być albo samochodem, albo samolotem, czyli instancją jednej z klas specjalizujących po klasie Pojazd.

Klasy abstrakcyjne stosowane są zarówno w modelowaniu rzeczywistości (modele dziedziny) jak i w modelowaniu kodu. Warto tutaj zwrócić uwagę na to, że w wielu językach programowania istnieją konstrukcje umożliwiające definiowanie klas abstrakcyjnych. Inną konstrukcją używaną podczas programowania, analogiczną do klas abstrakcyjnych jest **interfejsy**. Interfejsy, podobnie jak klasy abstrakcyjne nie posiadają instancji. Zasadniczą rolą interfejsów jest definiowanie zestawów operacji opisujących usługi dostarczane przez określone podsystemy. Stosowanie interfejsów pozwala na oddzielenie specyfikacji funkcjonalności (interfejs) od jej implementacji (konkretna klasa). W odróżnieniu od klas abstrakcyjnych, interfejsy zazwyczaj nie posiadają atrybutów, a jedynie operacje. W języku UML interfejsy modelujemy podobnie jak klasy, przy czym dodatkowo oznaczamy stereotypem «interface». Inną metodą oznaczenia interfejsu jest umieszczenie ikonki ‘o’ (okrągu) w prawym górnym rogu ikony interfejsu.

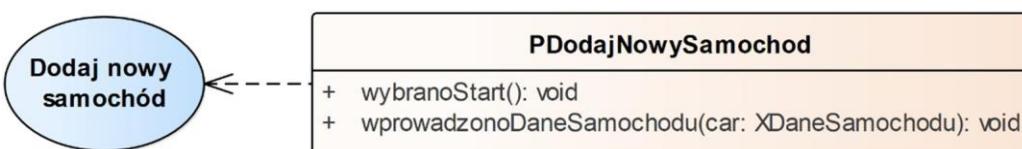
Notację interfejsów ilustruje rysunek 5.11. Na rysunku, interfejs „IPulaSamochodow” jest implementowany przez klasę „MPulaSamochodow”. Implementacja interfejsu przez klasę w języku UML

modelowana jest za pomocą relacji **realizacji**. Relacja ta jest notacyjnie podobna do relacji generalizacji, przy czym strzałka jest rysowana linią przerywaną. Warto zauważyć, że klasa „MPulaSamochodow” powtarza wszystkie operacje realizowanego interfejsu. W ten sposób wskazujemy, że klasa ta posiada metody (kod wykonawczy) dla odpowiednich operacji interfejsu.



Rysunek 5.11: Przykład interfejsu wraz realizującą go klasą

Wszystkie wyżej wymienione elementy są charakterystyczne dla modelu klas. Istnieją jednak również elementy generyczne, które mogą się pojawić na różnego typu diagramach. Takimi elementami, często używanymi również na diagramach klas są relacje zależności oraz notatki. **Relacje zależności (ang. Dependency)** ilustruje rysunek 5.12. Są one oznaczane strzałką, skierowaną zawsze w jedną stronę i narysowaną za pomocą przerywanej linii. Relacje zależności możemy umieszczać między dwoma klasami, ale również między dwoma dowolnymi elementami języka UML. Przykładowo, na rysunku 5.12 widzimy, że klasa „PDodajNowySamochod” zależy od przypadku użycia „Dodaj nowy samochód” (o przypadkach użycia mówimy w następnym rozdziale). Klasa ta stanowi „klienta” relacji, a przypadek użycia – „dostawcę”. Klient jest w pewien sposób zależny od dostawcy, to znaczy – jego znaczenie nie jest kompletne bez istnienia (wyjaśnienia znaczenia) dostawcy.

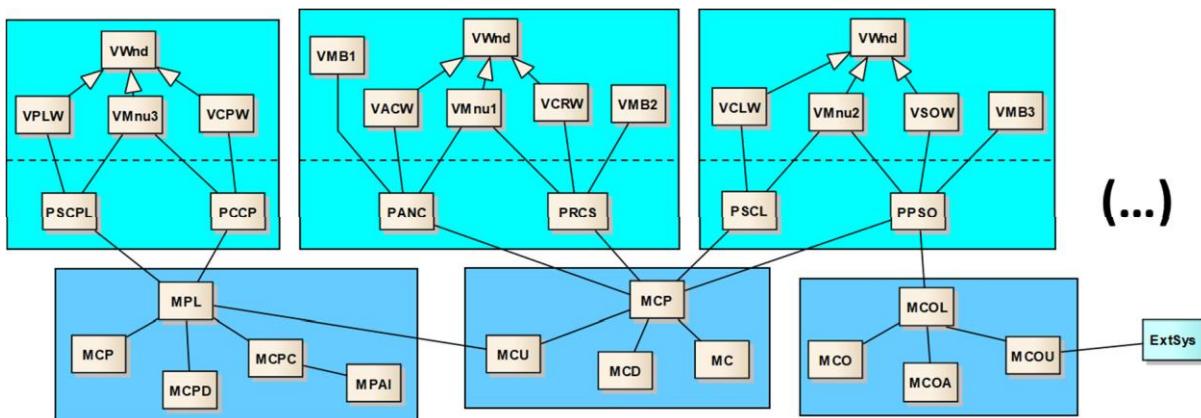


Rysunek 5.12: Przykład relacji zależności

Warto podkreślić, że znaczenie relacji zależności nie jest ustalonych. Przykładowa semantyka może dotyczyć na przykład tego, że każda zmiana w opisie dostawcy (np. scenariuszach przypadku użycia) rodzi konieczność zmiany opisu klienta (np. kodu klasy). Rysunek 5.12 pokazuje także inny przykład ustalenia semantyki relacji zależności. Wynika z niego, że kod klasy dostawcy jest zależny od klasy klienta w ten sposób, że w kodzie klasy klienta istnieje parametr lub zmienna lokalna definiowana przez klasę dostawcy. Wyjaśnienie takiej semantyki umieszczone zostało w **notatce**. Notatki w języku UML mogą być doczepione do dowolnego elementu modelu, w tym również do relacji. Ikona notatki przypomina kartkę z zagiętym rogiem z treścią będącą dowolnym tekstem. Notatka powinna być połączona z komentowanym elementem (lub elementami) za pomocą przerywanej linii.

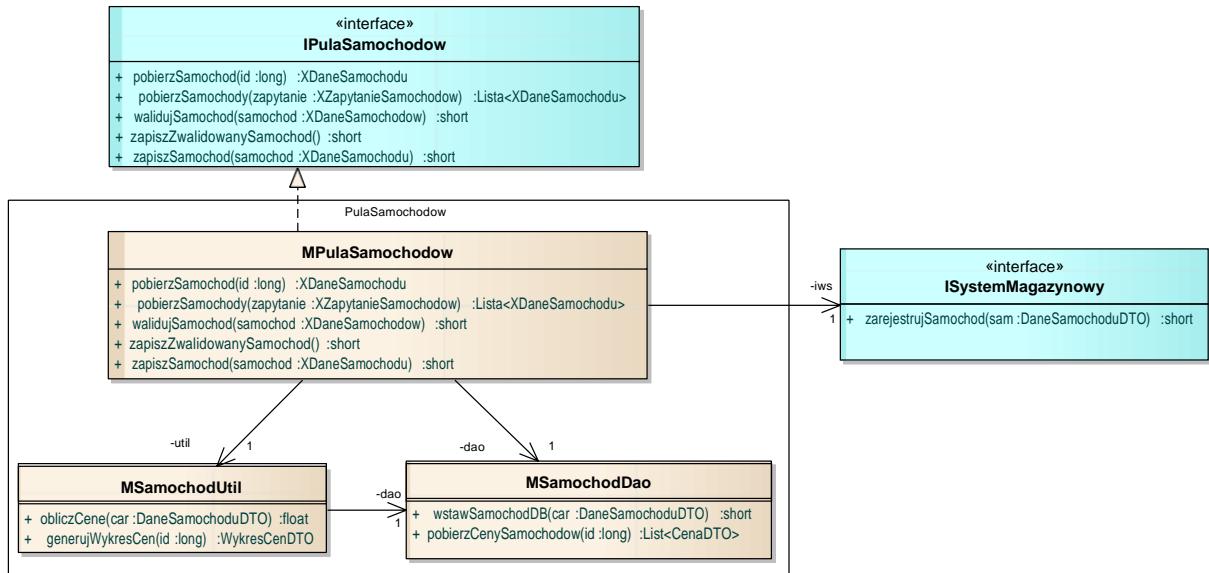
5.2. Model komponentów i model wdrożenia

Kod systemu oprogramowania może składać się z setek lub nawet tysięcy klas. Czyni to model klas bardzo złożonym i powoduje trudności w zarządzaniu nad tą złożonością. Najczęściej stosowaną techniką zarządzania nad złożonymi systemami jest podział całości na mniejsze podsystemy lub komponenty. Technikę podziału struktury złożonego systemu na podsystemy ilustruje rysunek 5.13. Widzimy tu diagram klas definiujący strukturę kodu przykładowego systemu oprogramowania (nazwy klas zostały uproszczone dla zwięzości rysunku). Klas na diagramie jest ponad 30, ale cały system może ich zawierać znacznie więcej. Chcąc podzielić system na podsystemy grupujemy po kilka-kilkanaście klas o dużej intensywności współpracy (spójności). Jednocześnie dokonujemy podziału tam, gdzie współpraca między klasami (np. liczba asocjacji) jest stosunkowo niewielka.



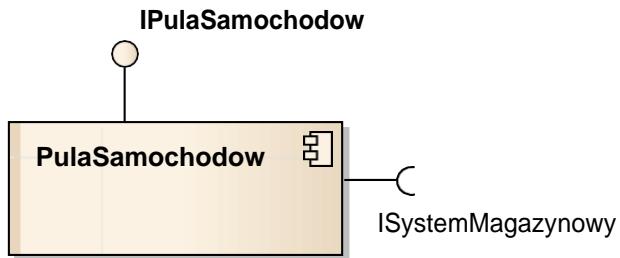
Rysunek 5.13: Podział struktury systemu na podsystemy

Kolejną techniką ułatwiającą zarządzanie nad złożonością systemów jest stosowanie dobrze określonych interfejsów pomiędzy komponentami. Interfejs definiuje „fasadę” dla komponentu, poprzez którą z danym komponentem komunikują się inne komponenty. Ważne jest to, że wewnętrzne komponentu zostaje ukryte za fasadą i nie jest dostępne z zewnątrz. Komunikacja poprzez interfejsy pozwala ograniczyć kanały komunikacji wewnętrznych systemu. Pozwala to zmniejszyć wzajemne zależności między klasami, które utrudniają diagnozę ewentualnych błędów. Zasady definiowania podsystemów z interfejsami ilustruje rysunek 5.14. Diagram przedstawia podsystem „PulaSamochodow” składający się z 3 klas. Jedna z klas realizuje interfejs „IPulaSamochodow” dostarczany przez ten podsystem. Równocześnie, klasa ta korzysta z interfejsu „ISystemMagazynowy”, realizowanego przez inny podsystem, nie uwidoczniony na diagramie.



Rysunek 5.14: Podsystem wykorzystujący interfejsy

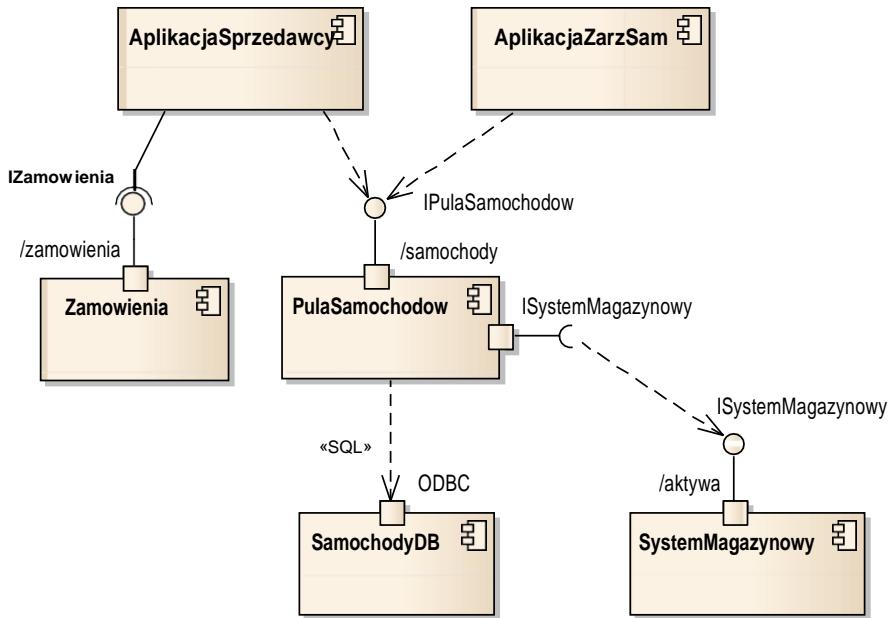
Diagram na rysunku 5.14 pokazuje w szczegółach strukturę podsystemu. Naszym celem jest jednak ukrycie szczegółów i koncentracja na najważniejszych elementach. Do tego celu możemy wykorzystać model komponentów. Podstawową jednostką modelu komponentów jest **komponent**, którego notacja w języku UML jest przedstawiona na rysunku 5.15. Komponenty reprezentowane są podobnie do klas, czyli w formie prostokątów z nazwą. Zasadniczo, nie posiadają one jednak dodatkowych przegródek na atrybuty i operacje. Aby odróżnić komponenty od klas oznaczamy je specjalną ikoną w prawym górnym rogu. Na diagramach komponentów umieszczamy komponenty wraz z interfejsami. **Interfejsy dostarczane** przez komponent oznaczane są specjalną „wypustką” zakończoną okręgiem. **Interfejsy wymagane** przez komponent różnią się tym, że wypustka zakończona jest półokręgiem. Zwróćmy uwagę na to, że oznaczenia interfejsów na diagramach komponentów wykorzystują analogię wtyczki i kontaktu. Interfejs wymagany i dostarczany można połączyć ze sobą tworząc odpowiednią relację między komponentami.



Rysunek 5.15: Podstawowe elementy notacji komponentu

Porównując rysunki 5.14 i 5.15 łatwo zauważyc, że diagram komponentów ukrywa wiele szczegółów. Jeśli chcemy przedstawić ogólną strukturę systemu, tworzymy diagram komponentów. Jeśli chcemy przedstawić szczegóły struktury poszczególnych komponentów, pokazujemy szczegółowe diagramy klas zawierające definicje interfejsów i klas je realizujących. Oczywiście, na diagramie komponentów możemy uwidaczniać wiele komponentów oraz relacje między nimi. Rysunek 5.16 przedstawia najważniejsze elementy notacyjne dla relacji między komponentami. Relacje najczęściej modelujemy za pomocą relacji zależności (strzałka z przerywaną linią). Najbardziej podstawową metodą jest łączenie interfejsu wymaganego z interfejsem dostarczanym (patrz: „ISystemMagazynowy” na rysunku 5.16). Skróconą metodą zapisu jest pominięcie interfejsu wymaganego i połączenie komponentu zależnością

bezpośrednio z interfejsem dostarczonym przez inny komponent (patrz: „IPulaSamochodów”). Jeszcze bardziej zwartą metodą zapisu jest zastosowanie tzw. **połączenia montażowego** (ang. Assembly). Jest to specjalny rodzaj relacji, którego notacja wykorzystuje połączone notacje interfejsu wymaganego i dostarczanego (patrz: „IZamowienia”). Możliwe jest również połączenie zależnością komponentów w sposób bezpośredni – bez interfejsu. Takie połączenie tworzymy najczęściej wtedy, kiedy komponenty komunikują się w sposób inny niż proceduralny (np. poprzez zapytania SQL). Takie zależności warto wtedy oznaczyć dodatkowym stereotypem (patrz stereotyp «SQL» na rysunku 5.16).

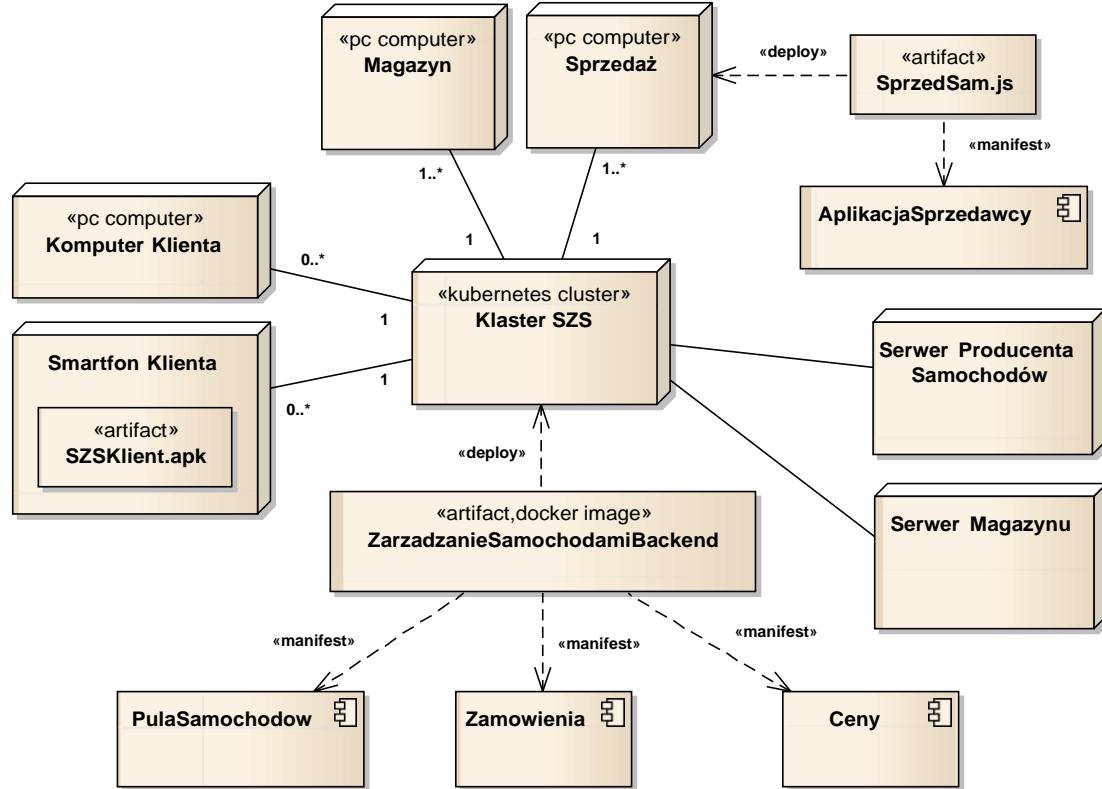


Rysunek 5.16: Przykładowy diagram komponentów

Rysunek 5.16 prezentuje jeszcze jeden istotny element diagramów komponentów – tzw. **porty**. Port oznacza punkt interakcji komponentu, służący do komunikacji z otoczeniem (np. udostępnieniem usług). Port reprezentowany jest za pomocą małego kwadratu umieszczonego na krawędzi komponentu. Opcjonalna nazwa portu może być umieszczona obok ikony portu. Port może stanowić miejsce powiązania interfejsu z komponentem. Mówimy wtedy, że interfejs jest udostępniany (albo wymagany) poprzez odpowiedni port. Port może mieć różne znaczenie zależne od technologii stosowanej do komunikacji między komponentami. Przykładowo w technologiach stosujących usługi typu REST, port może definiować punkt dostępu (ang. endpoint) z odpowiednim adresem względnym określonym w nazwie portu (patrz np. porty „/samochody” i „/zamówienia” na rysunku 5.16). W technologiach dostępu do relacyjnych baz danych, port może odpowiadać uchwytem połączenia bazodanowego (np. ODBC).

Model komponentów definiuje ogólną logiczną strukturę systemu. Często jednak konieczne jest również przedstawienie struktury fizycznej, czyli sprzętowej oraz powiązanie jej ze strukturą logiczną. Do tego celu wykorzystujemy **model wdrożenia**, inaczej nazywany modelem montażu (ang. Deployment Model). Fizyczne składniki systemu, stanowiące środowisko działania oprogramowania, nazywane są **węzłami** (ang. Node). Węzłami mogą być na przykład serwery, komputery osobiste, urządzenia mobilne i maszyny wirtualne. Reprezentacją węzłów są prostopadłościany z umieszczoną wewnątrz nazwą. Miedzy węzłami można poprowadzić asocjacje (jak w modelu klas) oznaczające określone ścieżki komunikacyjne. Asocjacje mogą być wyposażone w krotkość definującą możliwe liczby węzłów danego typu. Węzły z relacjami asocjacji zostały zilustrowane na rysunku 5.17. Zgodnie z tym modelem, system powinien obejmować węzeł (np. duża farma serwerów) typu „Klaster Szs” oraz wiele połączonych z nim węzłów typu „Magazyn”, „Sprzedaż”, „Komputer Klienta” i „Smartfon Klienta” (co najmniej

1 dla pierwszych dwóch). Dodatkowo, Klaster SZS łączy się z węzłami typu „Serwer Producenta Samochodów” oraz „Serwer Magazynu”. Zwróćmy uwagę na to, że węzłom (tak jak dowolnym elementom modeli w języku UML) możemy nadawać stereotypy. W naszym przykładzie, stereotypy («kubernetes cluster», «pc computer») doprecyzowują rodzaj węzła z punktu widzenia zastosowanego rozwiązania technologicznego (klaster serwerów w technologii Kubernetes, komputer PC).



Rysunek 5.17: Przykładowy diagram wdrożenia

Na diagramach wdrożenia możemy również umieszczać tzw. **artefakty** (ang. Artifact), które reprezentują elementy wykonywane w odpowiednich węzłach. Artefakty najczęściej oznaczają pliki wykonawcze zgodne z konkretną technologią (np. pliki EXE w systemie Windows, pliki APK w systemie Android, pliki JS wykonywane w przeglądarkach internetowych). Notacja dla artefaktów przypomina notację dla komponentów, przy czym wyróżnikiem jest umieszczenie stereotypu «artifact». Artefakty mogą być „montowane” w węzłach. Rysunek 5.17 pokazuje dwa możliwe sposoby zaznaczenia montażu artefaktów. Pierwszy z nich polega na umieszczeniu ikony artefaktu wewnętrznie ikony węzła (patrz „SZSKlient.apk”). Drugi polega na zastosowaniu specjalnej relacji zależności (tzw. **relacji montażu**) ozначенą stereotypem «deploy» (patrz „SprzedSam.js” i „ZarzadzanieSamochodamiBackend”). Przy okazji zauważmy, że elementy języka UML mogą mieć wiele stereotypów. W naszym przykładzie artefaktowi „ZarzadzanieSamochodamiBackend” nadaliśmy dodatkowy stereotyp «docker image», który sygnalizuje technologię wykonania tego artefaktu (tzw. obraz w technologii Docker).

Artefakty stanowią połączenie modelu fizycznego i logicznego. Modelują one bowiem fizyczne elementy (np. plik), które wyrażają odpowiednie komponenty, czyli składniki logicznej struktury systemu. Na diagramach wdrożenia możemy również pokazać tego typu zależności. Stosujemy w tym celu tzw. **relację wyrażania**, oznaczaną stereotypem «manifest». Na rysunku 5.17 widzimy kilka takich relacji, które wskazują na komponenty przedstawione w modelu komponentów (por. rysunek 5.16).

W niniejszym rozdziale skoncentrowaliśmy się na omówieniu podstawowej notacji języka UML w zakresie modelowania struktury. Przedstawiona notacja jest wystarczająca do opisu zdecydowanej większości aspektów budowy systemu oprogramowania.Więcej informacji na temat zastosowania modeli struktury do specyfikowania wymagań oraz projektowania oprogramowania znajdziemy w module III.

Zadania

Zadanie 1

Stwórz model klas z tematyki figur geometrycznych. Uwzględnij elementy takie jak trójkąt, czworokąt, trójkąt równoramienny, trójkąt równoboczny, prostokąt, romb, trapez, kwadrat, elipsa i koło. Użyj adekwatnych relacji i zaproponuj parę przykładowych atrybutów. Wyjaśnij z jakiej perspektywy jest stworzony powstały model.

Zadanie 2

Stwórz model klas z tematyki oceniania przedmiotów akademickich. Uwzględnij elementy takie jak przedmiot, ocena, student i nauczyciel akademicki. Użyj adekwatnych relacji i zaproponuj parę przykładowych atrybutów. Wyjaśnij z jakiej perspektywy jest stworzony powstały model.

Zadanie 3

Zaproponuj hipotetyczną strukturę komponentów dla systemu obsługi dziekanatu. Uwzględnij system powinien wspierać zarządzanie studentami, zajęciami i ocenami.

Zadanie 4

Narysuj diagram wdrożenia dla modelu z zadania 3. Diagram powinien zawierać co najmniej dwa węzły.

Słownik pojęć

Agregacja

Rodzaj asocjacji, która reprezentuje relacje zawierania obiektów, np. grupowania składników (części) pewnej całości. Agregacja oznaczana jest poprzez dodanie do asocjacji symbolu rombu po stronie całości.

Artefakt

Reprezentacja elementu wykonywanego w odpowiednich węzłach. Najczęściej oznacza plik wykonywalne zgodne z konkretną technologią. Artefakt oznaczany jest prostokątem z nazwą oraz stereotypem „artifact”.

Asocjacja

Definicja możliwych relacji między obiektemi klas. Asocjacja może definiować role oraz krotkości obiektów wchodzących ze sobą w relacje. Oznaczamy ją za pomocą linii ciągłej łączącej określone elementy modelu (najczęściej – klasy).

Atrybut

Własność opisująca obiekty danej klasy, której da się przypisać konkretną wartość. Atrybut posiada nazwę i typ.

Klasa

Grupa podobnych obiektów, które zostały uznane za warte wspólnego reprezentowania na potrzeby danego modelu. Klasę reprezentujemy za pomocą prostokąta z nazwą.

Komponent

Reprezentacja podsystemu realizującego pewien fragment funkcjonalności całego systemu. Komponent oznaczany jest za pomocą prostokąta z nazwą (jak klasa) i dodatkową ikoną komponentu.

Generalizacja

Określenie zależności pomiędzy klasą obiektów bardziej ogólnych i klasą obiektów bardziej specjalizowanych. Klasa specjalizowana dziedziczy cechy klasy ogólnej. Generalizacja oznaczana jest strzałką z trójkątnym grotem skierowaną w kierunku klasy ogólnej.

Interfejs

Definicja zestawu operacji opisujących usługi dostarczane przez określone podsystemy. Interfejs, podobnie jak klasa abstrakcyjna nie posiada instancji i jest oznaczany ikoną okręgu.

Interfejs dostarczany

Oznaczenie realizacji przez dany komponent funkcjonalności zdefiniowanej przez określony interfejs. Interfejs dostarczany oznaczamy poprzez okrąg z linią połączoną z komponentem.

Interfejs wymagany

Oznaczenie konieczności skorzystania przez dany komponent z funkcjonalności zdefiniowanej przez określony interfejs. Interfejs wymagany oznaczamy poprzez półokrąg z linią połączoną z komponentem.

Klasa abstrakcyjna

Klasa na tyle ogólna, że nie posiada z zasady żadnych obiektów będących jej instancjami. Nazwa klasy abstrakcyjnej zapisana jest kursywą.

Kompozycja

Rodzaj agregacji, który reprezentuje związek pomiędzy całością a jej integralnymi częściami. Kompozycja oznaczana jest podobnie jak agregacja, przy czym romb jest wypełniony.

Krotność (liczność)

Określenie liczby elementów danego typu występujących w relacji z innym elementem modelu (np. obiektem). Krotność podajemy jako zakres możliwych wartości.

Nawigowalność

Oznaczenie możliwości efektywnego osiągnięcia obiektów jednej klasy przez obiekty drugiej klasy. Nawigowalność oznaczana jest poprzez dodanie do asocjacji grotu strzałki.

Operacja

Reprezentacja możliwego sposobu zachowania modelowanych obiektów poprzez określenie możliwej do wykonania funkcji lub procedury. Operacja posiada nazwę oraz listę parametrów i typ rezultatu.

Połączenie montażowe

Specjalny rodzaj relacji, który reprezentuje korzystanie przez jeden komponent z interfejsu dostarczanego przez drugi komponent. Notacja tej relacji wykorzystuje połączone notacje interfejsu wymaganego i dostarczanego.

Port

Oznaczenie punktu interakcji komponentu, służącego do komunikacji z otoczeniem (np. udostępnienie usług). Port reprezentowany jest za pomocą małego kwadratu umieszczonego na krawędzi komponentu.

Realizacja

Relacja reprezentująca implementację interfejsu przez klasę. Relacja ta jest notacyjnie podobna do relacji generalizacji, przy czym strzałka jest rysowana linią przerywaną.

Relacja montażu

Relacja reprezentująca możliwość zamontowania i wykonania artefaktu na określonym węźle. Relacja ta jest oznaczana jak relacja zależności z dodatkowym stereotypem „deploy”.

Relacja wyrażania

Relacja reprezentująca odzwierciedlenie komponentu (elementu logicznego) za pomocą odpowiedniego artefaktu (elementu fizycznego, np. pliku). Relacja ta jest oznaczana jak relacja zależności z dodatkowym stereotypem „manifest”.

Stereotyp

Oznaczenie elementu modelu, umożliwiające nadanie dodatkowego (np. bardziej precyzyjnego) znaczenia temu elementowi modelu. Stereotyp oznaczamy przez jego nazwę umieszczoną w nawiasach francuskich.

Typ wyliczeniowy

Definicja typu danych poprzez określenie listy możliwych wartości obiektów tego typu.

Węzeł

Fizyczny składnik systemu, stanowiący środowisko działania oprogramowania. Reprezentacją węzła jest prostopadłościan z umieszczoną wewnątrz nazwą.

Widoczność

Określenie możliwego dostępu do danego elementu modelu przez inne elementy modelu. Widoczność może być m.in. publiczna, prywatna oraz chroniona.

Zależność

Rodzaj generycznej relacji między dwoma elementami, z których jeden jest klientem (elementem zależnym), a drugi jest dostawcą (elementem, od którego coś zależy). Relację tą oznaczamy przerywaną strzałką skierowaną w stronę dostawcy.

Co trzeba zapamiętać

Model klas

Model klas jest uniwersalnym modelem, używanym do odzwierciedlania struktury wynikowej z klasyfikacji elementów i uporządkowania łączących je zależności. Diagramy klas mogą dotyczyć opisu rzeczywistości, definicji struktur danych lub projektu szczegółowej struktury kodu. Podstawowymi elementami modelu klas są klasy i interfejsy oraz łączące je relacje: asocjacje, agregacje, kompozycje, generalizacje, realizacja. Klasy mogą posiadać atrybuty oraz operacje.

Model komponentów

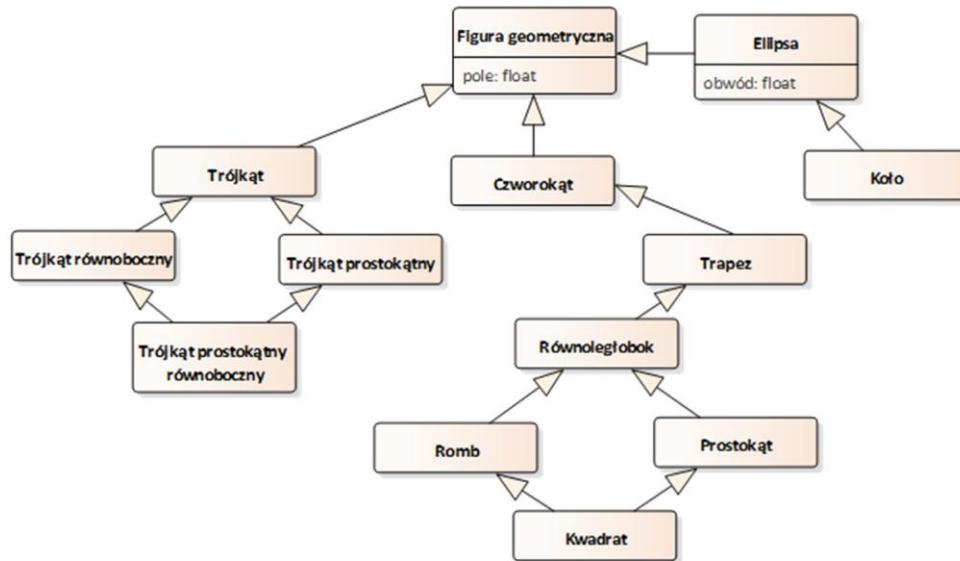
Model komponentów znajduje zastosowanie przede wszystkim w definiowaniu architektur logicznych systemów oprogramowania. Ma on na celu przedstawianie struktur systemów, które są na tyle złożone, że konieczne jest opisanie ich na wyższym poziomie abstrakcji niż poziom klas. Dzięki temu możliwe jest istotne ograniczenie liczby prezentowanych szczegółów, a tym samym – lepsze panowanie nad złożonością całości. Diagramy komponentów zawierają komponenty z portami oraz interfejsami dostarczanymi i wymaganymi, a także odpowiednie relacje zależności między nimi.

Model wdrożenia

Model wdrożenia (nazywany też modelem montażu) znajduje zastosowanie w modelowaniu architektur fizycznych systemów oprogramowania. Diagramy wdrożenia pozwalają na zdefiniowanie fizycznych elementów budowanego systemu informatycznego oraz połączeń między nimi. Umożliwiają one również pokazanie zależności pomiędzy strukturą logiczną a fizyczną podmiotu modelowania. Na diagramach montażu umieszczamy węzły z relacjami oraz artefakty z komponentami, połączone odpowiednimi relacjami montażu i wyrażania.

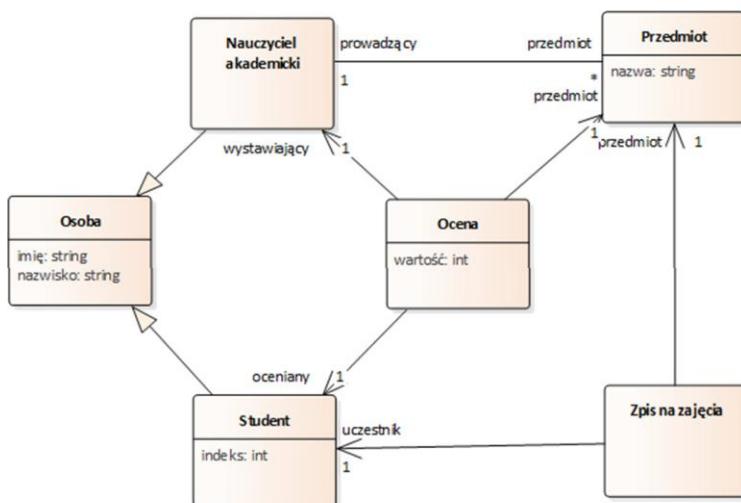
Rozwiązania zadań

Rozwiązanie zadania 1



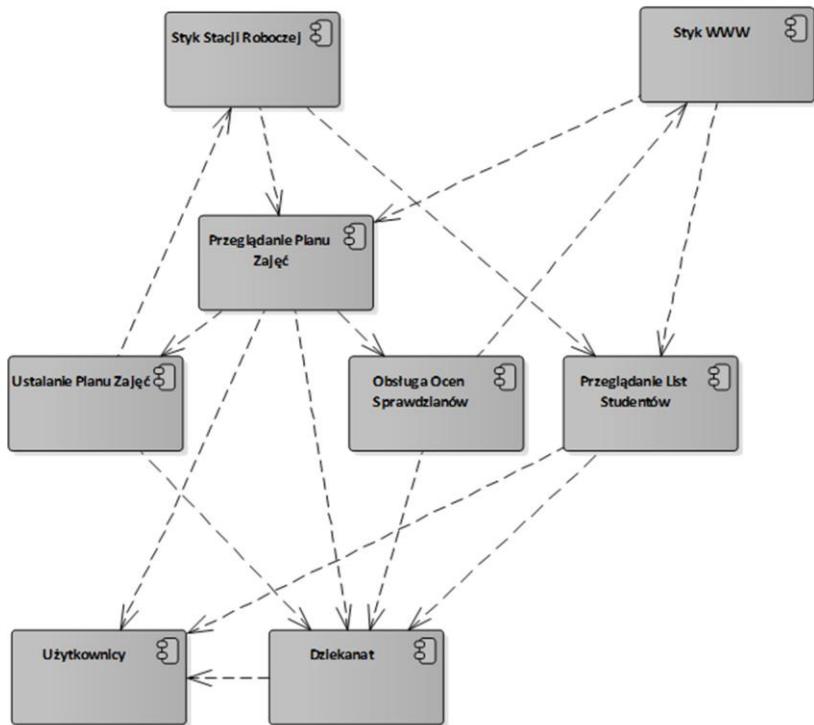
Model jest stworzony z perspektywy opisu rzeczywistości.

Rozwiązanie zadania 2

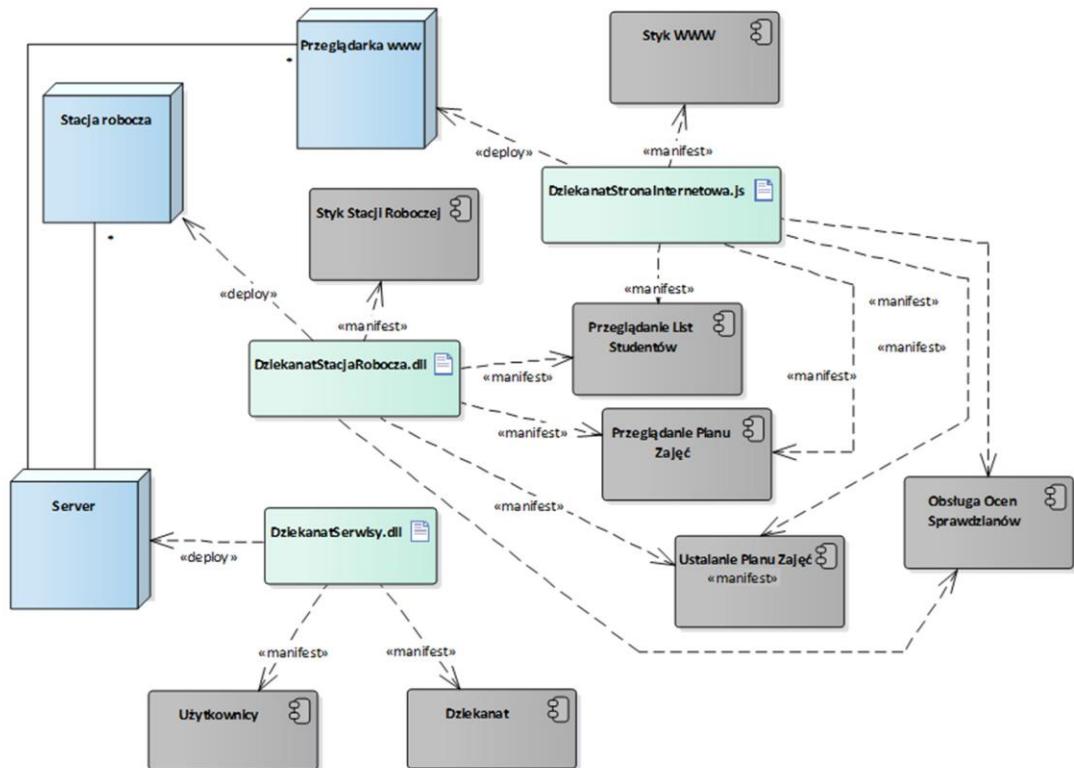


Model może być widziany jako stworzony zarówno z perspektywy modelowania rzeczywistości, jak i modelowania kodu.

Rozwiązań zadania 3



Rozwiązań zadania 4



6. Modelowanie dynamiki systemu

Każdy modelowany system podlega zmianom. Obiekty składające się na system wchodzą we wzajemne interakcje, wykonują określone akcje oraz zmieniają swój stan. Dynamika danej dziedziny problemu może dotyczyć na przykład realizacji procesów biznesowymi, interakcji między współpracownikami czy przebiegu określonych zjawisk fizycznych. Dynamika systemu oprogramowania, ujęta w sposób obiektowy, stanowi skomplikowaną sieć akcji podejmowanych przez obiekty będące elementami kodu. Oprócz przedstawienia struktury systemu bardzo istotne jest zatem przedstawienie jego dynamiki. Przy tym, przez model dynamiki rozumiemy przedstawianie przedmiotu modelowania od strony istotnych zmian jakim on podlega.

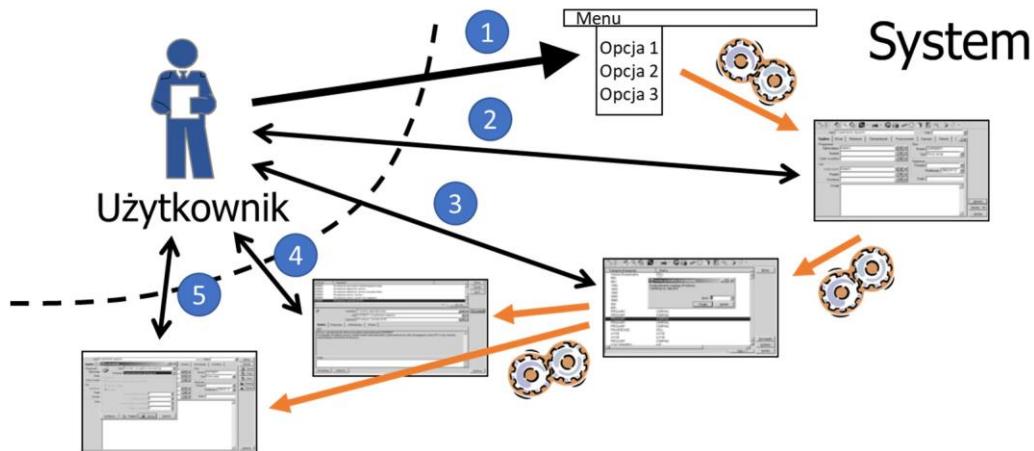
Języku UML oferuje szereg modeli, za pomocą których można przedstawić różne aspekty dynamiki systemu. Najprostszym jeśli chodzi o składnię jest **model przypadków użycia** (ang. Use Case Model). Za pomocą diagramów przypadków użycia możemy opisywać zewnętrzny sposób zachowania się systemu, tzn. jego funkcjonalność z punktu widzenia środowiska (głównie – użytkowników). Przypadki użycia (ang. Use Case) realizowane są na rzecz aktorów (ang. Actor), którzy reprezentują klasy obiektów współpracujących z systemem (użytkowników i systemy zewnętrzne). Innym, często używanym modelem jest **model czynności** (ang. Activity Model). Diagramy czynności stanowią grafy skierowane opisujące sieci akcji wraz z odpowiednimi przepływami sterowania między akcjami. Bardzo podobną notację grafową posiada **model maszyny stanów** (ang. State Machine Model). Diagramy maszyny stanów służą do pokazywania sposobu zachowania się jakiegoś elementu struktury systemu (np. klasy obiektów) w postaci przejść pomiędzy jego stanami. Warto zwrócić uwagę na to, że podstawową cechą odróżniającą model czynności od modelu maszyny stanów jest treść węzłów grafu. W pierwszym przypadku węzłami są akcje, a w drugim – stany. Czwartym często używanym rodzajem modeli dynamiki systemu są modele interakcji, a w szczególności – **model sekwencji** (ang. Sequence Model). Diagramy sekwencji pojawiły się już w pierwszym rozdziale tego modułu. Pokazują one sekwencje komunikatów przesyłanych między obiektami w ramach działającego systemu. Komunikaty oznaczają polecenia wykonania określonych usług, opisują przepływ sterowania między obiektami oraz przepływ danych (parametry komunikatów).

Innymi modelami interakcji są **modele komunikacji** (ang. Communication Model), **modele opisu interakcji** (ang. Interaction Overview Model) oraz **modele następstwa czasowego** (ang. Timing Model). Są one jednak na tyle rzadko używane, że pominiemy ich opis jako wykraczający poza zakres tego podręcznika. W następnych sekcjach przedstawimy składnię języka UML dla czterech podstawowych modeli dynamiki oraz wyjaśnimy ich semantykę (znaczenie) w sposób ogólny. O zastosowaniu i znaczeniu modeli dynamiki w modelowaniu wymagań oraz projektowaniu oprogramowania mówimy w module III (Podstawy inżynierii wymagań i projektowania oprogramowania).

6.1. Model przypadków użycia

Częstą potrzebą podczas modelowania systemu oprogramowania jest potrzeba zidentyfikowania elementarnych jednostek funkcjonalności systemu z punktu widzenia jego użytkowników. Takie jednostki w języku UML nazywamy przypadkami użycia systemu. Przypadki użycia powinny dostarczać użytkownikowi konkretnej korzyści poprzez np. realizację jakiegoś celu biznesowego. W typowym, interaktywnym systemie oprogramowania, cel biznesowy osiągamy w rezultacie przejścia przez ciąg interakcji użytkownika z systemem. Przykładowo, na rysunku 6.1 widzimy użytkownika systemu, który chce

uzyskać określony rezultat (np. zarejestrować nowego kontrahenta albo wydrukować listę kontrahentów). Aby osiągnąć ten rezultat, użytkownik musi najpierw wybrać określoną opcję w menu (pkt. 1 na rysunku). Efektem jest np. wyświetlenie formularza lub okienka na co użytkownik reaguje wprowadzeniem dany oraz wybriem jednej z dostępnych w nowym oknie opcji (pkt. 2). Taki dialog między użytkownikiem a systemem może doprowadzić do osiągnięcia zamierzonego celu (pkt. 3 i 4). Może też zakończyć się niepowodzeniem (pkt. 3 i 5). Niepowodzenie może być spowodowane np. błędem weryfikacji danych lub wybriem przez użytkownika opcji anulowania transakcji.



Rysunek 6.1: Współpraca systemu z użytkownikiem

Aby zidentyfikować funkcjonalność systemu z punktu widzenia jego użytkowników powinniśmy w pierwszej kolejności zdefiniować tychże użytkowników. W języku UML do tego celu wykorzystujemy **aktorów**. Aktor definiuje klasę obiektów spoza modelowanego systemu. Obiekty te mogą być osobami, urządzeniami lub zewnętrznym systemami informatycznymi. Możemy również powiedzieć, że aktor definiuje rolę graną przez obiekty w stosunku do danego systemu. Ważne jest to, że aktor nie może określać jakiejkolwiek części modelowanego systemu oprogramowania. Aktor, jako obiekt zewnętrzny, komunikuje się z modelowanym systemem najczęściej za pośrednictwem jakiegoś interfejsu. Może to być interfejs użytkownika (np. interfejs graficzny – GUI) lub interfejs programistyczny (np. API). Przez interfejs użytkownika z systemem komunikują się aktorzy będący osobami, a przez interfejs programistyczny – zewnętrze urządzenia i systemy informatyczne.

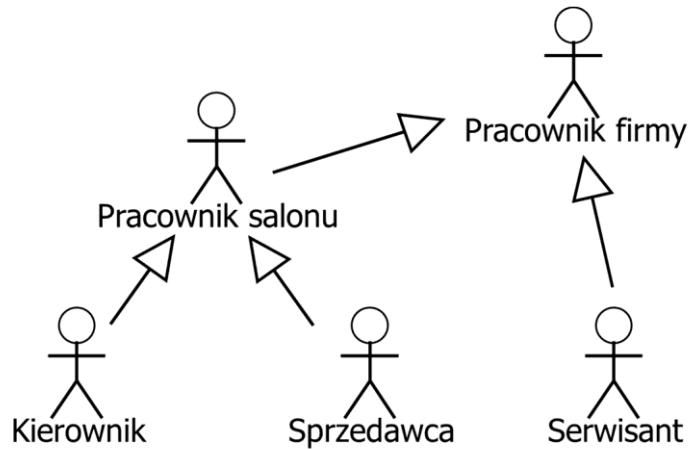
Podstawową notacją aktora w języku UML jest ikona człowieka narysowana prostymi kreskami, co ilustruje rysunek 6.2 (ikona po lewej stronie). Tak jak w przypadku innych elementów modeli, również aktorom można nadawać stereotypy. Częstą praktyką jest nadawanie stereotypu «system» aktorom definiującym zewnętrzne systemy informatyczne (ikona w środku rysunku). Jeśli stosujemy odpowiednie narzędzie do modelowania w języku UML, nadanie stereotypu może być również związane ze zmianą kształtu ikony (ikona po prawej stronie).



Rysunek 6.2: Warianty notacji aktora

Warto podkreślić, że aktor jest rodzajem klasy. Aktorzy mają zatem niektóre cechy wspólne z klasami w modelu klas. W szczególności, możliwe jest stosowanie relacji generalizacji między aktorami. Przykład zastosowania tych relacji widzimy na rysunku 6.3. W przykładzie tym, najbardziej ogólnym

aktorem jest „Pracownik firmy”. Pozostali aktorzy specjalizują (dziedziczą) od niego zgodnie z odpowiednio skierowanymi relacjami. Dziedziczeniu podlegają głównie relacje aktorów z przypadkami użycia. Aktor ogólny określa pewien zakres odpowiedzialności wyrażony jego uczestnictwem w określonych przypadkach użycia. Aktorzy szczególnowi dziedziczą ten zakres odpowiedzialności, ale mogą uczestniczyć w kolejnych przypadkach użycia, które poszerzają ich zakres odpowiedzialności.



Rysunek 6.3: Relacje generalizacji dla aktorów

Aktorzy grają zatem istotną rolę w całym modelu przypadków użycia. Zbiór aktorów określa granicę między światem zewnętrznym a modelowanym systemem. Dla aktorów i przy ich udziale realizowane są wspomniane na początku „elementarne jednostki funkcjonalności”, czyli – przypadki użycia systemu. **Przypadek użycia** definiujemy jako klasę zachowań modelowanego systemu (tzw. podmiotu), prowadzących do osiągnięcia obserwowalnego, istotnego rezultatu dla jakiegoś aktora (lub aktorów). Warto podkreślić, że przypadek użycia – podobnie jak aktor – jest rodzajem klasy. Jegoinstancjami (obiektami) są konkretne wykonania definiowanego przez niego zachowania systemu. Przypadek użycia najczęściej definiuje różne warianty zachowania, czyli jego wykonania mogą przebiegać w różny sposób.

Podstawową notacją dla przypadków użycia w języku UML jest ikona elipsy, co ilustruje rysunek 6.4. Nazwa przypadku użycia może być umieszczona pod lub wewnątrz elipsy. Jako ciekawostkę można przytoczyć wariant notacji, który stosuje ikonę klasy (po prawej stronie rysunku). Jest to rzadko spotykana notacja, która nawiązuje do faktu, że przypadki użycia są rodzajem klas.



Rysunek 6.4: Warianty notacji przypadku użycia

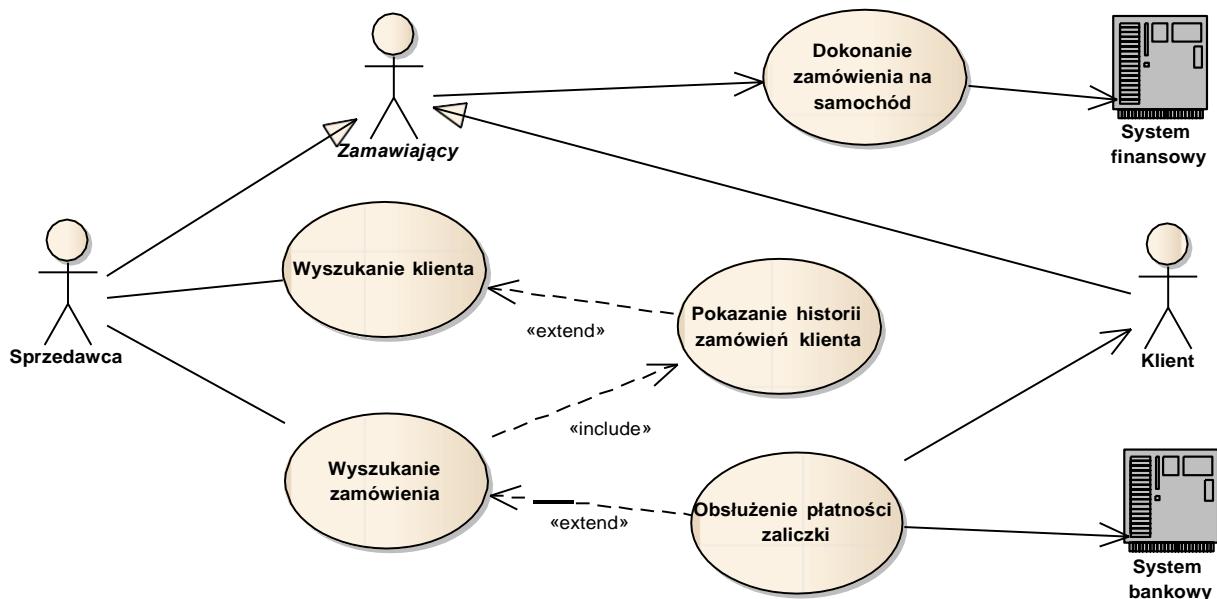
Każdy przypadek użycia powinien mieć jasno określony cel, który powinien wynikać z jego **nazwy**. Warto tutaj przestrzegać określonych konwencji, które ułatwiają określenie celu przypadku użycia. Ważne jest, aby nazwy były pisane w jednolitej formie. Można na przykład przyjąć formę, w której stosujemy rzeczowniki odczasownikowe (np. „Zarejestrowanie pojazdu”, „Dodanie użytkownika”, „Pokaż ...”, „Obsłużenie ...”). Taka forma podkreśla efekt wykonania przypadku użycia. Inną formą jest forma polecenia (np. „Zarejestruj pojazd”, „Dodaj użytkownika”, „Pokaż ...”, „Obsłuż ...”). Taka forma podkreśla to, że za osiągnięcie zadanego celu jest przede wszystkim odpowiedzialny system – to jemu wydajemy odpowiednie polecenia. Warto zwrócić uwagę na to, że istotnym błędem jest stosowanie

prostych rzeczowników lub fraz rzeczownikowych w podstawowych formach jako nazw przypadków użycia (np. „*Lista pojazdów*”, „*Rejestracja*”, „*Zamówienia*”).

Sposobem na realizację celu określonego w nazwie przypadku użycia jest wymiana komunikatów między aktorem (lub aktorami) a systemem. Komunikaty te przeplatane są także akcjami systemu, których skutki są następnie widoczne dla aktorów. Pierwszy komunikat pochodzi zazwyczaj od aktora, który uruchamia daną funkcjonalność. Przypadek użycia powinien definiować wszystkie możliwe alternatywne zachowania systemu prowadzące do danego celu. Te alternatywy mogą prowadzić do zadanego celu inną drogą niż podstawowy sposób zachowania. Mogą również kończyć się niepowodzeniem – cel nie zostanie osiągnięty mimo podjęcia próby jego osiągnięcia.

Warto podkreślić, że system opisany za pomocą przypadków użycia jest traktowany jako „czarna skrzynka”. Pokazane jest jedynie to, co widzi i czego skutki może zobaczyć aktor. W szczególności, szczegółowy opis przypadku użycia powinien zawierać scenariusze opisujące nawigację poprzez ekranы systemu oraz zmianę stanu systemu odczuwalną (teraz lub później) przez aktorów. Nie powinien natomiast zawierać szczegółów technicznych – np. sposobów komunikacji między komponentami systemu, zapisu danych w bazie danych, tworzenia wątków przetwarzania, przesyłania danych poprzez sieć lokalną. Te elementy opisujemy dopiero podczas projektowania realizacji poszczególnych przypadków użycia systemu.

Na diagramach przypadków użycia umieszczamy aktorów i przypadki użycia wraz z łączącymi je relacjami. Na rysunku 6.5 widzimy diagram zawierający przykłady takich relacji. Najczęściej spotykaną jest relacja **asocjacji** między aktorem i przypadkiem użycia. Relacja ta oznacza możliwość uczestnictwa danego aktora w powiązanym z nim przypadku użycia. W ramach takiego uczestnictwa, aktor może być tzw. aktorem głównym. Oznacza to, że aktor uruchamia dany przypadek użycia i jest on realizowany na jego rzecz. Bardzo często, taki przypadek użycia jest rozpoczynany poprzez wybranie odpowiedniej opcji z ekranu głównego danej aplikacji. Udział aktora jako aktora głównego możemy zaznaczyć poprzez zastosowanie asocjacji nawigowej, skierowanej do przypadku użycia. Na rysunku 6.5 przykładem takiej relacji jest relacja między „*Zamawiającym*”, a „*Dokonaniem zamówienia na samochód*”. Inną możliwością jest oznaczenie danej asocjacji stereotypem «use» („wykorzystanie”).

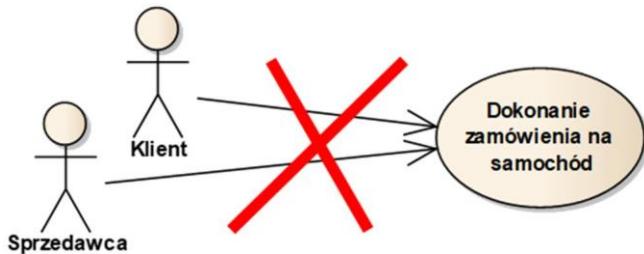


Rysunek 6.5: Przykładowy diagram przypadków użycia

Aktor może również pełnić rolę poboczną (pomocniczą) w relacji z przypadkiem użycia. Oznacza to, że aktor zaczyna brać udział dopiero w dalszych krokach scenariuszy danego przypadku użycia.

Przykładowo, może to oznaczać, że system w pewnym momencie wyświetla na ekranie aktora pobocznego monit (np. komunikat o konieczności potwierdzenia operacji). Aktor musi wtedy zareagować na taki monit, aby wykonanie przypadku użycia mogło być kontynuowane. Przykładem na rysunku 6.5 są relacje przypadku użycia „Obsłużenie płatności zaliczki” z aktorami „Klient” i „System bankowy”. Na podstawie tego fragmentu diagramu możemy założyć, że pełne wykonanie tego przypadku użycia wymaga interwencji klienta (np. klient przeciąga przez czytnik swoją kartą płatniczą) oraz interakcji z systemem bankowym (dokonanie transakcji płatniczej). Uczestnictwo aktora pomocniczego możemy zaznaczyć poprzez zastosowanie nawigowalności asocjacji skierowanej w stronę tegoż aktora.

Warto tutaj zwrócić uwagę na częsty błąd popełniany podczas modelowania przypadków użycia., który ilustruje rysunek 6.6. Twórca modelu zapewne chciał zapisać, że przypadek użycia „Dokonanie zamówienia na samochód” może być uruchomiony przez aktora „Klient” lub przez aktora „Sprzedawca”. W rezultacie powstał model, który oznacza jednoczesne uczestnictwo tych aktorów w tym przypadku użycia jako aktorów głównych (uruchamiających). Prawidłowy model widzimy na rysunku 6.5. Wprowadzono tam abstrakcyjnego aktora „Zamawiający” (nazwa pisana kursywą), który generalizuje dwóch rzeczonych aktorów. Relacja z przypadkiem użycia dotyczy tylko tego nowego aktora, co skutkuje uzyskaniem zamierzzonego efektu – przypadek użycia może być uruchomiony przez jednego ze specjalizujących aktorów. Należy podkreślić, że język UML nie zabrania modelowania więcej niż jednego aktora głównego dla danego przypadku użycia. Takie sytuacje są stosunkowo rzadkie, ale mogą wystąpić – szczególnie w systemach czasu rzeczywistego. Dotyczą przypadków, kiedy kilka osób jednocześnie musi uczestniczyć w uruchamianiu jakiejś funkcjonalności modelowanego systemu.



Rysunek 6.6: Błąd w modelowaniu relacji aktorów z przypadkiem użycia

Oprócz relacji między aktorami i przypadkami użycia często stosowane są relacje między przypadkami użycia. Język UML standardowo definiuje dwie relacje zależności – relację «include» i relację «extend». Relacja «**include**» („wstawienie”) oznacza bezwarunkowe włączenie w treść scenariuszy jednego przypadku użycia, treści scenariuszy innego. Na rysunku 6.5 widzimy jedną taką relację, oznaczoną przerywaną strzałką z odpowiednim stereotypem. Aby zilustrować znaczenie relacji «include» musimy poznać treść scenariuszy obydwu połączonych przypadków użycia. Przyjmijmy, że przypadek użycia „Pokazanie historii zamówień klienta” ma następujący scenariusz główny:

- 1) użytkownik wybiera opcję pokazania historii, 2) system pobiera historię, 3) system wyświetla okno historii, 4) użytkownik wybiera zamówienie, 5) system pobiera wybrane zamówienie, 6) system pokazuje okno szczegółów zamówienia, 7) użytkownik wybiera opcję zamknięcia historii.

Zgodnie z modelem na rysunku, ta treść zostaje włączona w odpowiednim miejscu w treść przypadku użycia „Wyszukanie zamówienia”. Możemy na przykład przyjąć, że ma on następujący scenariusz główny:

- 1) użytkownik wybiera opcję wyszukania zamówienia, 2) system pokazuje ogólny formularz wyszukiwania, 3) użytkownik wprowadza identyfikator klienta, 4) system pobiera dane klienta, 5) system pokazuje okno klienta, 6) **tu wstawiamy treść:** „Pokazanie historii zamówień klienta”, 7) użytkownik wybiera opcję zakończenia.

Zwróćmy uwagę na to, że wstawienie treści nie podlega żadnym warunkom. Jednocześnie, należy podkreślić, że wykonanie przypadku użycia włączającego (tu: „Wyszukanie zamówienia”) nie musi zawsze zawierać wykonania przypadku użycia włączanego (tu: „Pokazanie historii ...”). Możliwe jest wykonanie zgodne z jednym ze scenariuszy alternatywnych, który nie zawiera włączenia. Przykładem są następujące dwa scenariusze przypadku użycia „Wyszukanie zamówienia”:

1) – 2) jak w scenariuszu głównym, 3') użytkownik wybiera opcję zakończenia.

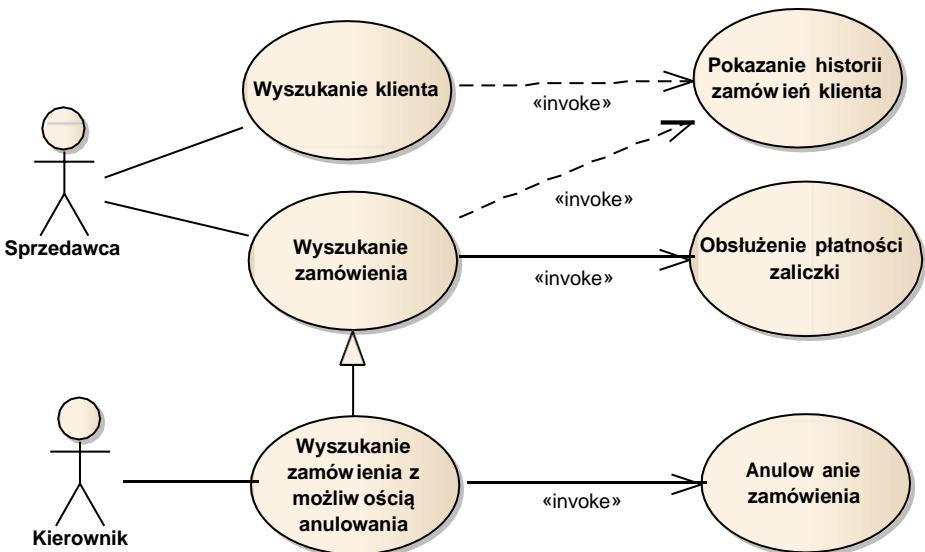
1) – 2) jak w scenariuszu głównym, 3'') użytkownik wprowadza identyfikator zamówienia, 4'') system pobiera wybrane zamówienie, 5'') system pokazuje okno szczegółów zamówienia, 6'') użytkownik wybiera opcję zakończenia.

Możemy zauważać, że przypadek użycia „Pokazanie historii zamówień klienta” uczestniczy w dwóch relacjach. Druga z nich to relacja «extend» („rozszerzenie”). Oznacza ona możliwość wplecenia treści scenariuszy przypadku użycia rozszerzającego w treść scenariuszy przypadku użycia rozszerzanego. Możliwość wplecenia może podlegać określonym warunkom, które powinny być dołączone do specyfikacji danej relacji «extend». W najprostszym przypadku, wplecenie warunkowane jest przez wybraniem opcji uruchamiającej rozszerzający przypadek użycia podczas wykonywania rozszerzanego przypadku użycia. Zilustrujemy to przykładowym scenariuszem przypadku użycia „Wyszukanie klienta”:

1) użytkownik wybiera opcję wyszukania klienta, 2) system pokazuje formularz wyszukiwania klienta, 3) użytkownik prowadza dane filtra klientów, 3) system wyszukuje klientów zgodnych z filtrem klientów, 4) system pokazuje okno listy klientów, 5) użytkownik wybiera klienta, 6) system pokazuje okno szczegółów klienta, 7) [wybranie opcji] tu wpiątamy treść: „Pokazanie historii zamówień klienta, 8) użytkownik wybiera opcję zakończenia.

Punkt 7 powyższego scenariusza stanowi tzw. **punkt rozszerzenia** (ang. Extension Point). Jest on wykonywany warunkowo, gdzie warunek („wybranie opcji”) umieszczony jest w nawiasach kwadratowych. Zwróćmy uwagę na to, że w naszym przykładzie, warunek dotyczył decyzji podejmowanej przez użytkownika systemu (wybranie lub nie wybranie opcji). Możliwe są również warunki dotyczące stanu systemu. Możemy na przykład założyć, że w systemie jest odpowiednio ustawiany „przełącznik”, który zabrania przeglądania historii zamówień bez znajomości ich identyfikatorów (poprzez wybór z listy). Wtedy warunek z punktu 7 należałoby rozszerzyć, np. następująco: [wybranie opcji ORAZ przeglądanie możliwe].

Semantyka relacji «include» i «extend» oparta jest na bezwarunkowym lub warunkowym wstawianiu treści jednych przypadków użycia w treść innych. Zrozumienie tej semantyki często sprawia kłopoty. W szczególności, często popełnianym błędem jest niewłaściwe skierowanie relacji. Zwróćmy uwagę na to, że relacja «include» wskazuje w kierunku przypadku użycia wstawianego. Z kolei, relacja «extend» wskazuje w kierunku przypadku użycia rozszerzanego. Z tego powodu powstała propozycja alternatywy - relacji «invoke». Relacja ta ma semantykę podobną do semantyki wywołania procedury. Zawsze jest skierowana od przypadku wywołującego do przypadku wywoływanego. Wywołanie może być bezwarunkowe lub możemy dla niego określić warunek. Rysunek 6.7 pokazuje fragment zawierający aktora i przypadki użycia z rysunku 6.5, gdzie zamiast standardowych relacji zastosowano relacje «invoke».



Rysunek 6.7: Wykorzystanie relacji «invoke»

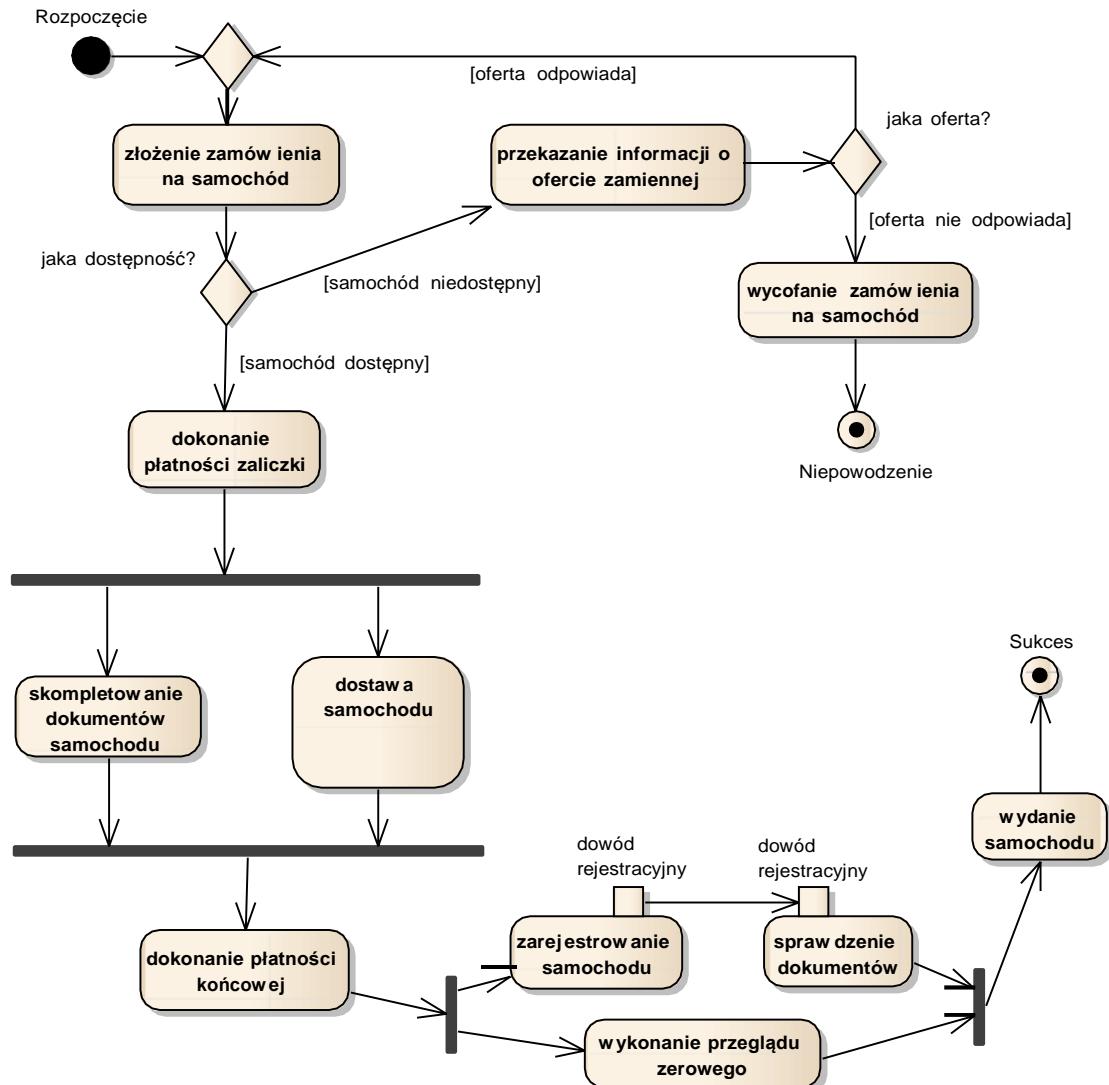
Podobnie jak dla aktorów, również między przypadkami użycia możliwe jest stosowanie relacji **generalizacji**. Oznacza ona, że przypadek użycia specjalizowany dziedziczy cechy przypadku użycia ogólnego. Rysunek 6.7 dostarcza nam odpowiedniego przykładu. Przypadek użycia „Wyszukanie zamówienia z możliwością anulowania” specjalizuje „Wyszukanie zamówienia”. Oznacza to, że sprzedawca ma do dyspozycji podstawową funkcjonalność wyszukiwania zamówienia, która m.in. umożliwia uruchomienie funkcjonalności „Pokazania historii zamówień klienta”. Kierownik ma natomiast do dyspozycji przypadek użycia, który rozszerza możliwość wyszukiwania zamówień o funkcjonalność przypadku użycia „Anulowanie zamówienia”.

6.2. Model czynności

Częstą potrzebą podczas modelowania systemów jest przedstawienie jakiegoś procesu w postaci schematu blokowego. Oznacza to konieczność zbudowania modelu składającego się z akcji oraz przepływów sterujących między akcjami, czyli modelu czynności. Modele czynności mogą opisywać algorytmy, procesy biznesowe, procesy fizyczne lub scenariusze przypadków użycia, które też tworzą pewnego rodzaju proces. W modelowaniu obiektowym, akcje najczęściej odpowiadają operacjom wykonywanym przez obiekty zawartym w definicjach odpowiednich klas. Zadaniem modelu czynności jest połączenie takich akcji w sieć (graf) opisującą kolejność ich wykonywania.

Rysunek 6.8 przedstawia przykładowy diagram, który posłuży nam do wyjaśnienia notacji modelu czynności. Diagramy czynności stanowią grafy, przy czym wyróżniamy w nich kilka rodzajów węzłów. Najbardziej typowym węzłem są **akcje**, oznaczane ikoną prostokąta z zaokrąglonymi rogami. Wewnątrz ikony umieszczona jest treść akcji, która powinna stanowić nazwę elementarnej operacji wykonywanej w ramach opisywanego procesu (np. „dokonanie płatności zaliczki”). Akcje mogą posiadać **wypustki** (ang. Pin) oznaczane małymi kwadratami umieszczonymi na krawędziach akcji. Wypustki służą definiowaniu danych przepływających między akcjami i oznaczane są zazwyczaj nazwą tych danych (np. „dowód rejestracyjny”). Do oznaczania decyzji służą **węzły decyzyjne** (ang. Decision Node) oraz **węzły scalenia** (ang. Merge Node). Obydwa rodzaje węzłów oznaczane są ikoną rombu (diamantu). Obok ikony węzła decyzyjnego można umieścić treść podejmowanej decyzji, często w formie pytania (np. „jaka dostępność?”). Na diagramach czynności możemy również zaznaczać ciągi akcji wykonywanych równolegle. Do tego celu wykorzystujemy belki synchronizacji. Belka **rozwidlenia** (ang. Fork) pozwala

rozdzielić sterowanie między kilka równolegich przepływów akcji. Z kolei belka **złączenia** (ang. Join) pozwala połączyć równolegle przepływy akcji w jeden. Belki synchronizacji rysujemy jako poziome lub pionowe grube kreski. Każdy diagram czynności powinien również zawierać **węzeł początkowy** (duża czarna kropka) oraz **węzły końcowe** (mała czarna kropka z obwódką).



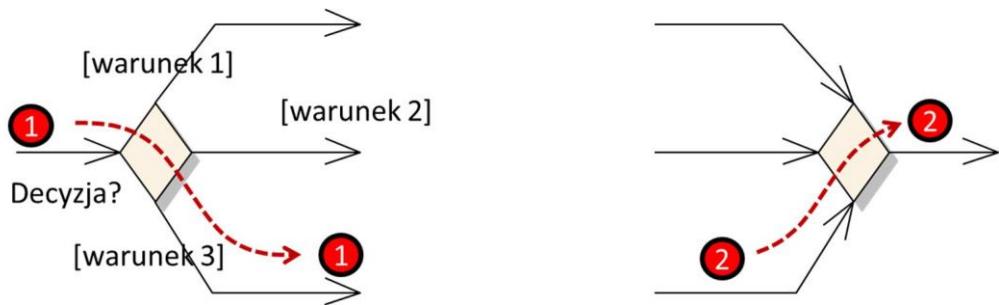
Rysunek 6.8: Przykładowy diagram czynności

Miedzy węzłami umieszczamy **przepływy sterowania** (ang. Control Flow) lub **przepływy obiektów** (ang. Object Flow). Obydwa rodzaje przepływów mają taką samą notację strzałki. Obok strzałki możemy umieścić **warunek** (ang. Guard), który zapisujemy jako tekst umieszczony w nawiasach kwadratowych (np. „[samochód dostępny]”). Przepływ sterowania oznacza proste przejście między węzłami, natomiast przepływ obiektów dodatkowo oznacza przesłanie między akcjami określonego zestawu danych. Przepływy obiektów umieszczają się najczęściej między wypustkami akcji. W ten sposób możliwe jest łatwe określenie danych przesyłanych takimi przepływami. Przykładem przepływu obiektów na rysunku 6.8 jest przepływ między wypustkami akcji „Zarejestrowanie samochodu” i „sprawdzenie dokumentów”. Pierwsza z tych akcji tworzy zestaw danych („dowód rejestracyjny”), który po jej zakończeniu jest przesyłany do drugiej z tych akcji. Pozostałe przepływy na rysunku są przepływami sterowania.

Działanie modelu czynności możemy opisać poprzez wykorzystanie żetonów poruszających się zgodnie z przepływami sterowania i przepływami obiektów przez sieć węzłów. Zaczynamy stawiając żeton w

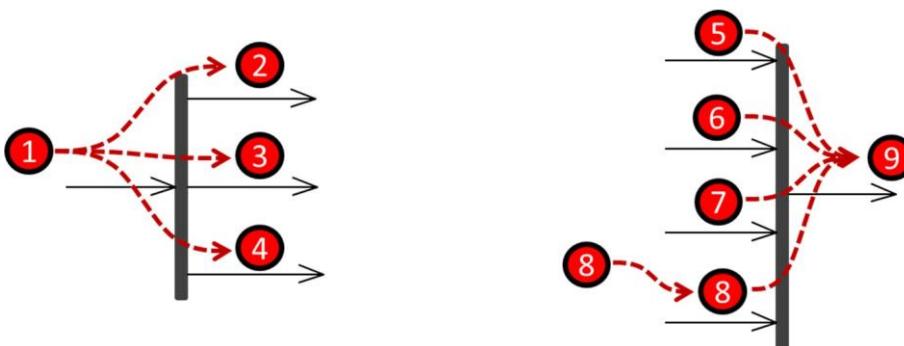
węzle początkowym (na rysunku 6.8 jest to węzeł oznaczony „Rozpoczęcie”). Z takiego węzła może wychodzić tylko jeden przepływ. Po rozpoczęciu „działania” danego diagramu czynności, żeton jest zatem przesuwany od razu do węzła będącego na końcu tego przepływu. Dalsze działanie zależy od rodzajów węzłów, co omówimy na kilku przykładach.

Rysunek 6.9 pokazuje sposób działania dla węzłów decyzyjnych i węzłów scalenia. Węzeł decyzyjny (po lewej stronie rysunku) posiada jeden przepływ wchodzący oraz wiele przepływów wychodzących. W momencie, kiedy dociera do niego żeton (oznaczony „1”), jest on natychmiast przesyłany na jedno z jego wyjść. Wybór wyjścia zależy od spełnienia warunków umieszczonych na przepływie. Należy zwrócić uwagę na to, że warunki powinny się wzajemnie wykluczać, powinny pokrywać pełną przestrzeń możliwości oraz być określone dla wszystkich wyjść. W przeciwnym wypadku, działanie węzła decyzyjnego nie będzie deterministyczne (może być losowe). Analogicznie, węzeł scalenia posiada wiele przepływów wejściowych i jeden przepływ wyjściowy. Żeton (tu oznaczony „2”) docierający na jedno z wejść takiego węzła jest natychmiast transportowany na wyjście.



Rysunek 6.9: Działanie węzłów decyzyjnych i węzłów scalenia

Rysunek 6.10 pokazuje sposób działania dla belek synchronizacji. Węzeł rozwidlenia (po lewej stronie rysunku) jest podobny do węzła decyzyjnego w tym sensie, że posiada jeden przepływ wchodzący i wiele przepływów wychodzących. Zasadnicza różnica polega na tym, że po przyjściu żetonu (tutaj oznaczonego „1”), węzeł rozwidlenia tworzy nowe żetony (tutaj – „2”, „3” i „4”) i przesyła je na wszystkie swoje wyjścia. W ten sposób w sieci czynności pojawia się wiele żetonów, które mogą być przesyłane równolegle między węzłami w trakcie dalszego przetwarzania. Analogicznie, węzeł złączenia posiada wiele przepływów wchodzących oraz jeden przepływ wychodzący. Jego działanie polega na tym, że czeka na pojawienie się żetonów na wszystkich swoich wejściach. Dopiero kiedy ostatni z tych żetonów (w naszym przykładzie – żeton „8”) dotrze do węzła, na wyjście przesyłany jest nowo utworzony żeton (tu – żeton „9”). W ten sposób przetwarzanie pochodzące z różnych ścieżek diagramu czynności może być synchronizowane i łączane w jedną ścieżkę.

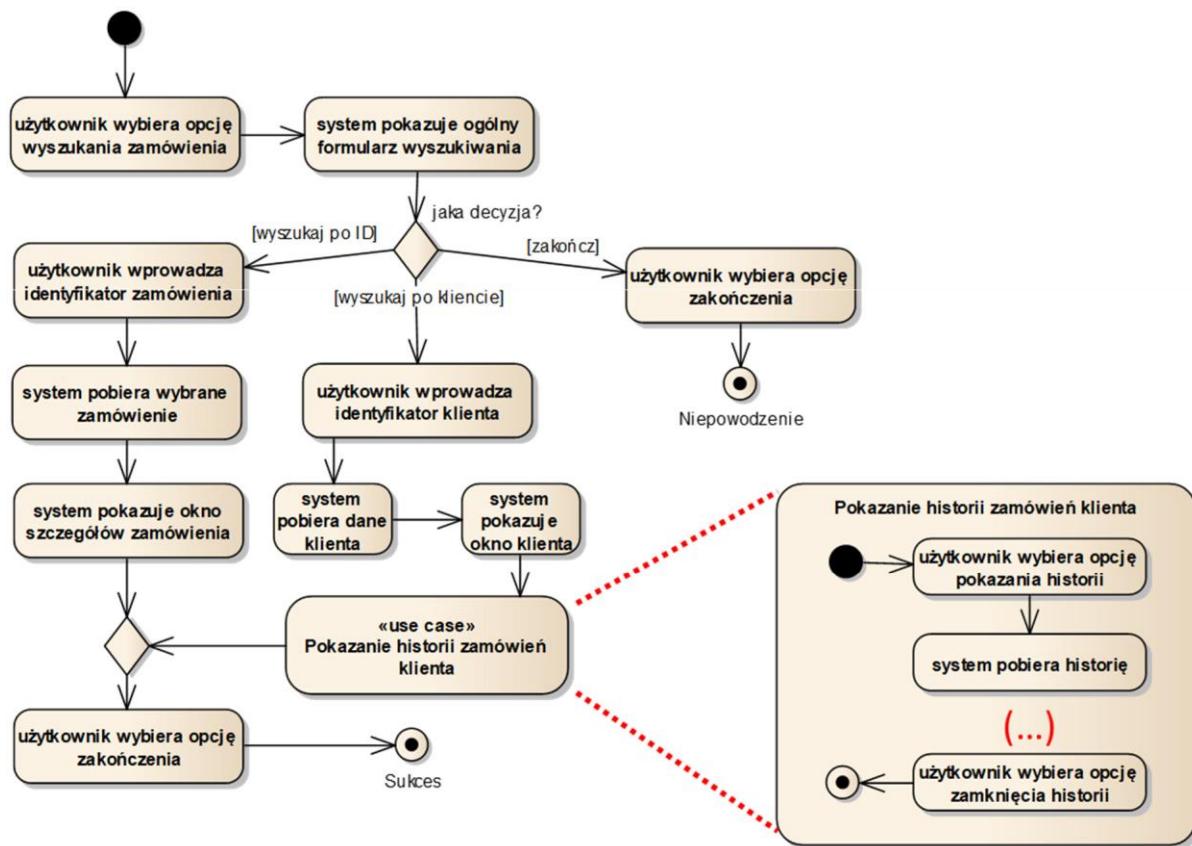


Rysunek 6.10: Działanie węzłów rozwidlenia (fork) i węzłów złączenia (join)

Działanie opisywane diagramem czynności kończy się w momencie dotarcia jednego z żetonów do **węzła końcowego**. W tym momencie pozostałe akcje są przerywane (jeśli działają) i działanie

całego systemu opisywanego diagramem jest przerywane. Istnieją również specjalne węzły końcowe, tzw. węzły końca przepływu (ang. Flow Final Node). Dotarcie do takiego węzła nie kończy działania całego systemu, a jedynie danej gałęzi – usuwany jest jedynie żeton, który dotarł do takiego węzła. Węzły końca przepływu reprezentowane są za pomocą okręgu ze znakiem „X” w środku. Zwróćmy uwagę na to, że z naturalnych względów, z węzłów końcowych nie mogą wychodzić żadne przepływy.

Bardziej skomplikowane sieci akcji możemy dzielić na mniejsze elementy. W tym celu, możemy definiować tzw. **węzły czynności** (ang. Activity Node). Czynność jest podobna do akcji w tym sensie, że definiuje jakąś operację wykonywaną przez obiekty danego systemu. Różnica polega na tym, że czynność nie jest operacją elementarną, ale może składać się z wielu akcji. Dla węzła czynności możemy narysować diagram czynności, który definiuje te akcje oraz odpowiednią sieć przepływów. Odpowiedni przykład zawiera rysunek 6.11. Węzłem czynności jest węzeł „Pokazanie historii zamówień klienta”. Dla tego węzła zdefiniowany został dodatkowy diagram, pokazany po prawej stronie rysunku, co ilustrują grube kropkowane linie (linie te nie są elementem notacji języka UML). Żeton, który dociera do węzła czynności natychmiast przekazywany jest do węzła początkowego zawartego w tym węźle diagramu czynności. W efekcie, realizowane są kolejne akcje zgodnie z definicją tego diagramu, aż do osiągnięcia jednego z węzłów końcowych.



Rysunek 6.11: Przykładowy węzeł czynności wraz z opisującym go diagramem

Zwróćmy uwagę na to, że diagram na rysunku 6.11 ilustruje jedno z zastosowań diagramów czynności – definiowanie scenariuszy przypadków użycia. Możemy ten diagram porównać ze scenariuszami przypadków użycia „Pokazanie zamówienia” oraz „Pokazanie historii zamówień klienta” przedstawionymi w poprzedniej części rozdziału. Jego zawartość informacyjna odpowiada scenariuszom w notacji czysto tekstopowej. Dla wielu osób wersja graficzna w postaci diagramu czynności jest czytelniejsza i pozwala na lepsze zrozumienie logiki danego przypadku użycia.

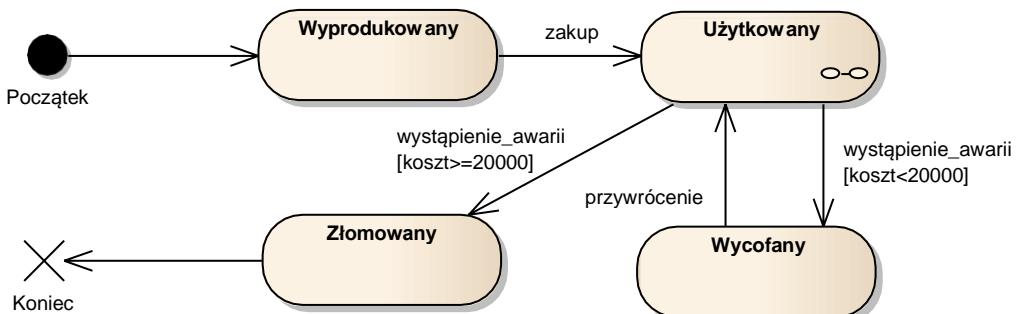
6.3. Model maszyny stanów

Bardzo podobnym w swojej notacji do modelu czynności jest model maszyny stanów. Obydwa rodzaje diagramów posługują się notacją grafu. Podstawowa różnica jest taka, że węzły w diagramach maszyny stanów definiują stany, a nie akcje. Sytuację, w której bardzo przydatne byłoby stworzenie diagramu maszyny stanów ilustruje rysunek 6.12. Obiekty klasy „Samochód” mogą przybierać jeden z czterech stanów. Stany te definiowane są typem wyliczeniowym „Status Samochodu” i reprezentowane poprzez atrybut „status” klasy „Samochód”. Zmiana stanów samochodu może nastąpić w trakcie wykonywania różnych operacji. Na rysunku zostały zdefiniowane trzy takie operacje. Z diagramu nie wynika jednak, które operacje i w jaki sposób mogą zmieniać wartości atrybutu „status”. Potrzebujemy zatem diagramu, który pokaże dynamikę zmiany stanów.



Rysunek 6.12: Przykładowa klasa z definicją stanów

Odpowiedni diagram jest przedstawiony na rysunku 6.13. **Stany** (ang. State) reprezentowane są za pomocą ikony prostokąta z zaokrąglonymi rogami. Każdy stan określa pewną stabilną konfigurację systemu (np. obiektu), która może trwać w czasie. Między stanami prowadzimy przejścia (ang. Transition) oznaczane strzałkami. Przejścia mogą być opisane przez podanie tzw. **wyzwalacza** (ang. Trigger). Wyzwalacz odpowiada konkretnemu zdarzeniu, które powoduje zmianę stanu. W naszym przykładzie, przejścia między stanami posiadają wyzwalacze, których nazwy odpowiadają operacjom klasy „Samochód”. Zwróćmy też uwagę na to, że dane zdarzenie może powodować przejścia między różnymi stanami.



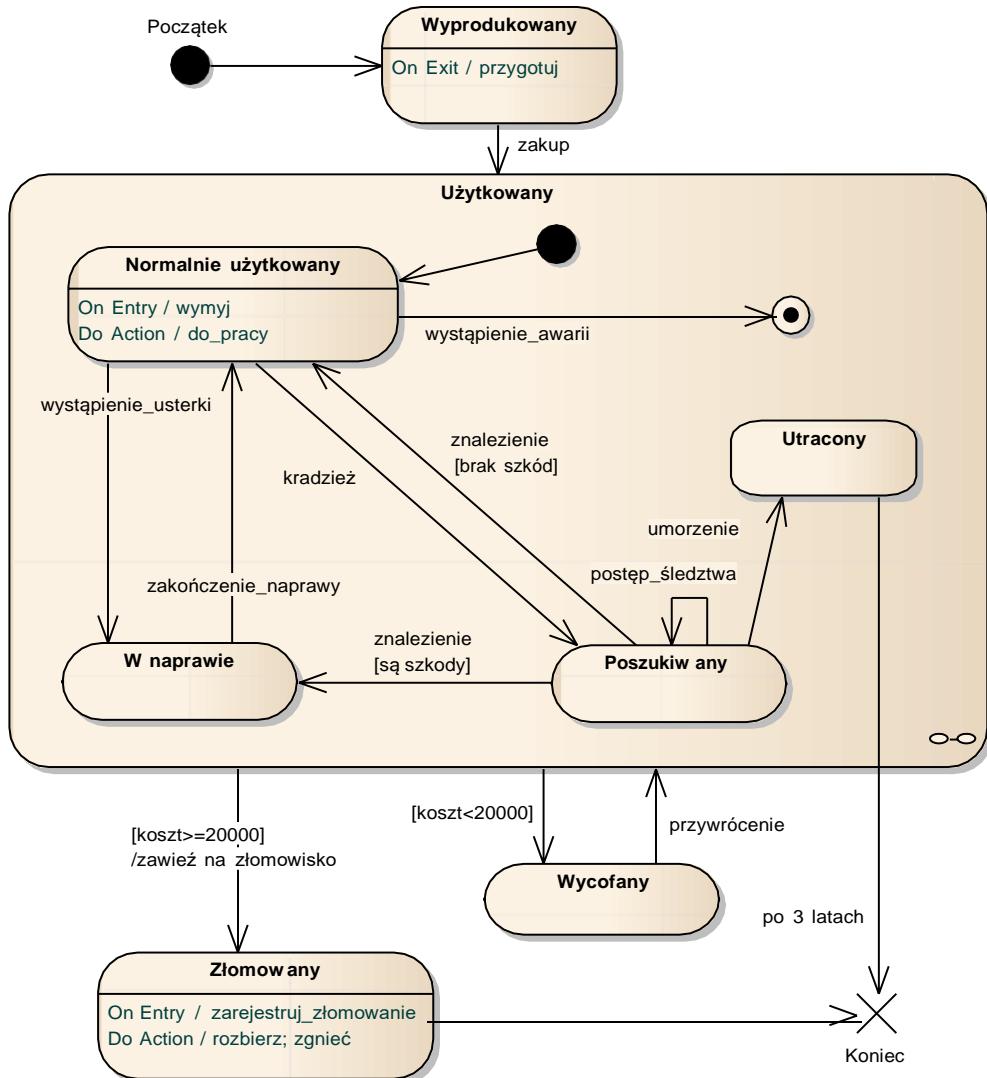
Rysunek 6.13: Przykładowy diagram maszyny stanów

Wyzwalacz może być dodatkowo wyposażony w warunek, zgodnie ze standardową notacją warunków w języku UML (napis w nawiasach kwadratowych). W przypadku wystąpienia zdarzenia określonego wyzwalaczem, dodatkowo badany jest warunek. Przejście między stanami następuje jedynie w razie spełnienia tego warunku. Przykładowo, zgodnie z diagramem na rysunku 6.13 nadziejście zdarzenia „wystąpienie_awarii” może spowodować zmianę stanu na „Złomowany” lub „Wycofany” w zależności od spełnienia odpowiednich warunków określonych w wyzwalaczach.

Na diagramie maszyny stanów występują również tzw. pseudostany. Najczęściej używanymi pseudostanami są pseudostan początkowy oraz pseudostan terminalny. **Pseudostan początkowy** (ang. Initial Pseudostate) oznacza stan, w którym rozpoczyna się działanie maszyny stanów (np. stan obiektu zaraz

po jego utworzeniu). Na diagramie, taki pseudostan połączony jest zawsze jednym przejściem z jednym stanem właściwym. Z diagramu na rysunku 6.13 wynika, że po utworzeniu obiektu klasy „Samochód” znajdzie się on w stanie „Wyprodukowany”. Analogicznie, **pseudostan terminalny** (ang. Terminal Pseudostate) oznacza możliwe stany, w których znajduje się maszyna stanów (np. obiekt) przez jej skasowaniem (zakończeniem działania). W naszym przykładzie, obiekt klasy „Samochód” jest kasowany w stanie „Złomowany”. Do pseudostanu terminalnego może prowadzić wiele przejść ze stanów właściwych.

Dla ułatwienia zrozumienia bardziej złożonych maszyn stanów możemy zastosować tzw. stany złożone. Stan złożony sam w sobie stanowi maszynę stanów, zawartą w maszynie, w której taki stan został umieszczony. Przykład stanu złożonego widzimy na rysunku 6.14. Wewnątrz stanu złożonego możemy umieścić „wewnętrzna” maszynę stanów. Maszyna taka posiada stany wraz z przejściami, a także pseudostan początkowy. Wejście do stanu złożonego (tutaj: stan „Użytkowany”) powoduje uruchomienie wewnętrznej maszyny stanów i przejście do stanu określonego pseudostanem początkowym (tu: stan „Normalnie użytkowany”). Wyjście ze stanu złożonego może nastąpić na dwa sposoby. Pierwszy sposób polega na utworzeniu przejścia z jednego ze stanów wewnętrznych stanu złożonego do stanu (lub pseudostanu) na zewnątrz stanu złożonego. Przykładem jest przejście między stanem „Ultraony” a pseudostanem „Koniec” na rysunku 6.14. Drugi sposób polega na przejściu wewnętrznej maszyny do tzw. **pseudostanu końcowego** (ang. Final Pseudostate), który oznaczamy za pomocą małej czarnej kropki z obwódką (jak węzeł końcowy w diagramach czynności). W tym wypadku, wyjście ze stanu złożonego następuje jednym z nienazwanych przejść z tego stanu do innego stanu. Jeśli takich przejść jest więcej niż jedno, powinny być one opatrzone warunkami. Przykładem na rysunku 6.14 jest przejście między stanem „Użytkowany”, a stanami „Złomowany” i „Wycofany”.



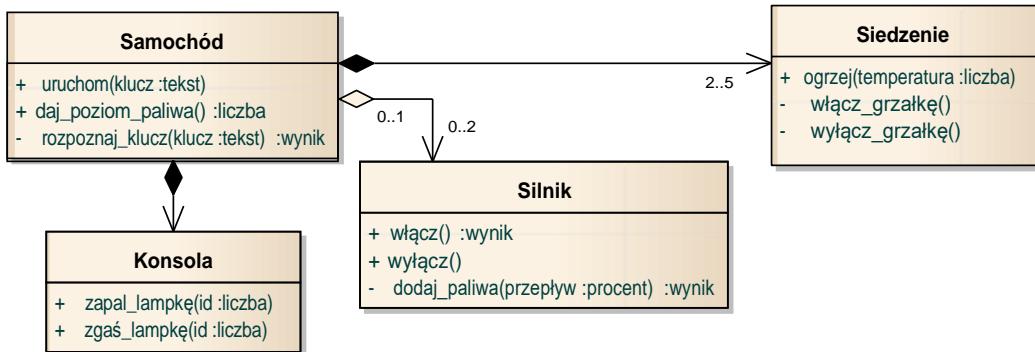
Rysunek 6.14: Przykładowy diagram maszyny stanów ze stanem złożonym i akcjami stanów

Oprócz wyzwalaczy, na diagramach maszyny stanów możemy oznaczać czynności wykonywane podczas zmiany stanów. Do każdego stanu możemy przypisać akcje wykonywane podczas wchodzenia do danego stanu, podczas wychodzenia z tego stanu oraz podczas trwania w tym stanie. Są to odpowiednio **akcje na wejściu** (oznaczane jako „On Entry”), **akcje na wyjściu** (oznaczane jako „On Exit”) oraz **czynności stanu** (oznaczane jako „Do Action”). Ponadto, każdy wyzwalacz powiązany z przejściem może być wyposażony w **akcję przejścia**. Wszystkie powyższe akcje oznaczane są poprzez podanie ich nazwy poprzedzonej znakiem ukośnika „//” (np. „//zarejestruj_złomowanie”, „//zwieź na złomowisko”).

6.4. Model sekwencji

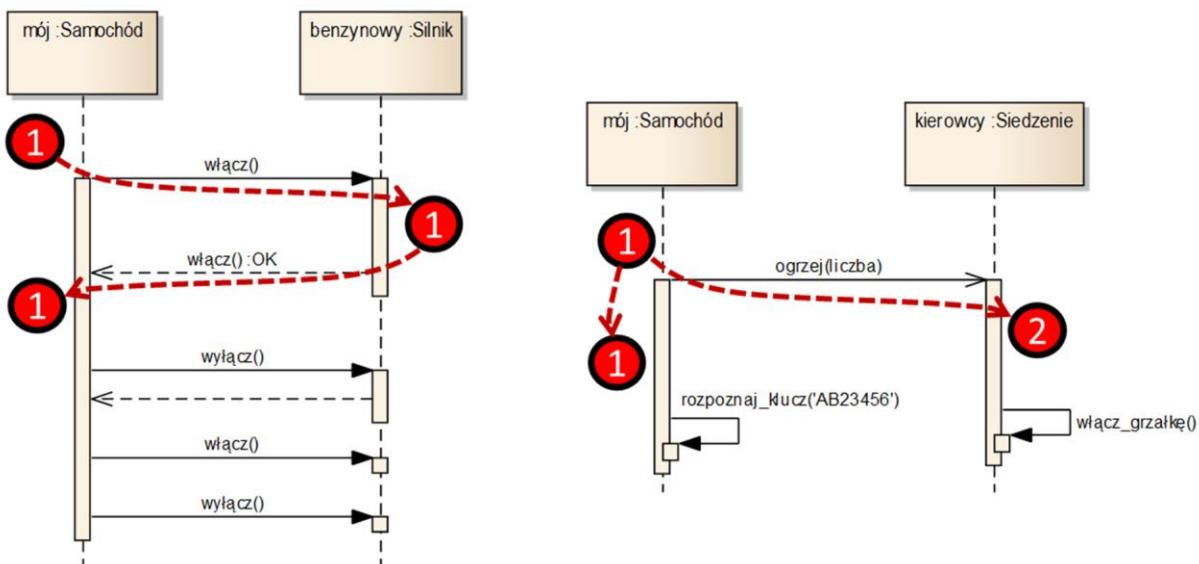
Przedstawione wyżej modele czynności reprezentują procesy jako ciągi akcji wykonywanych przez różne obiekty. Często jednak przydatne jest przedstawienie procesu jako sekwencji komunikatów wymienianych między obiektami. Do tego celu możemy wykorzystać modele sekwencji. Modele takie zwyczaj tworzymy w kontekście konkretnej struktury systemu wyrażonej modelem klas lub modelem komponentów. Komunikaty wymieniane między obiektami mogą odpowiadać konkretnym operacjom klas lub interfejsów komponentów. Ponadto, wymiana komunikatów powinna się odbywać zgodnie z nawigowalnością relacji między klasami lub komponentami.

Na rysunku 6.15 widzimy diagram klas, który posłuży nam do wyjaśnienia modelu sekwencji. Diagram przedstawia fragment modelu samochodu wraz z jego składowymi (silnikami, siedzeniami i konsolą). Zwróćmy uwagę na to, że relacje klasy „Samochód” z klasami składowymi są nawigowalne – skierowane w kierunku klas składowych. Ponadto, warto zauważyć, że część operacji klas jest operacjami prywatnymi. Te szczegóły diagramu klas są istotne z punktu widzenia możliwości tworzenia modelu sekwencji.



Rysunek 6.15: Diagram klas jako podstawa modelu sekwencji

Podstawowymi elementami diagramów sekwencji są **linie życia** (ang. Lifeline) oraz **komunikaty** (ang. Message), co ilustruje rysunek 6.16. Linia życia reprezentuje działanie jednego obiektu. Rysujemy ją jako pionową przerywaną linię zakończoną u góry ikoną obiektu. Na rysunku 6.16 widzimy cztery linie życia, przy czym dwie z nich dotyczą obiektu „mój” (typu „Samochód”). Między liniami życia umieszczamy komunikaty, które oznaczane są jako strzałki (najczęściej poziome, czasami rysowane skosem w dół). Kolejność przesyłania komunikatów wynika z kolejności ich umieszczenia na diagramie, który czytamy z góry na dół. Przesłanie komunikatu w jednego obiektu do drugiego powoduje wykonanie jakieś operacji przez obiekt docelowy. Wykonanie operacji zaznaczamy tzw. **belką wykonania** (ang. Execution Specification). Belka taką rysowana jest jako wąski prostokąt nałożony na linię życia obiektu, do którego kierowany jest komunikat. Przykładowo, komunikat „włącz” kierowany jest do obiektu „benzynowy: Silnik”. Po otrzymaniu takiego komunikatu, obiekt ten podejmuje działanie zaznaczone odpowiednią belką wykonania.



Rysunek 6.16: Podstawowe elementy notacji diagramów sekwencji

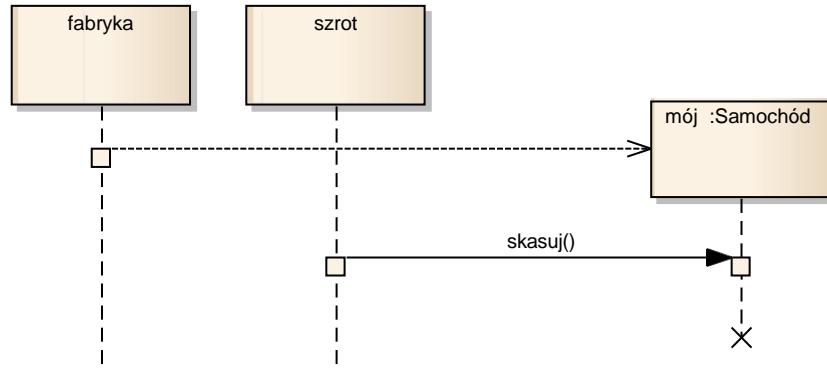
Zasadniczo wyróżniamy trzy rodzaje komunikatów. **Komunikaty synchroniczne** (ang. Synchronous Message) oznaczane są strzałką z wypełnionym grotem. Komunikaty takie najczęściej powiązane są z **komunikatami zwrotnymi**, oznaczanymi strzałką z pustym grotem, pisaną linią przerywaną. Znaczenie komunikatów synchronicznych i zwrotnych wyjaśnia diagram po lewej stronie rysunku 6.16. Wysłanie komunikatu synchronicznego „włącz()” powoduje przekazanie sterowania z obiektu „mój:Samochód” do obiektu „benzynowy:Silnik”, co symbolizuje żeton „1”. Drugi z tych obiektów podejmuje działanie („włącza się”), a pierwszy przechodzi w stan oczekiwania. Po wykonaniu operacji przez silnik, sterowanie (żeton „1”) wraca do samochodu. Oznaczane to jest komunikatem zwrotnym „włącz():OK”. Po zwróceniu mu sterowania, samochód może podjąć dalsze działania, w tym np. przesyłać kolejne komunikaty. Jak widzimy, dwa obiekty synchronizują swoje działania, gdyż jeden czeka na zakończenie działania drugiego. Stąd pochodzi nazwa komunikatów synchronicznych. Zauważmy, że komunikaty synchroniczne nie wymagają umieszczania odpowiadających im komunikatów zwrotnych. Dla komunikatów synchronicznych, zwrot sterowania na końcu belki wykonania następuje zawsze, niezależnie od istnienia komunikatu zwrotnego (patrz dwa ostatnie komunikaty „włącz()” i „wyłącz()”).

Treść komunikatów na diagramach sekwencji ma ustaloną składnię. **Etykieta** komunikatu synchronicznego (ogólnie – wszystkich komunikatów oprócz zwrotnych) zawiera jego nazwę (np. „włącz”) oraz opcjonalną listę argumentów wejściowych w nawiasach. Etykieta komunikatu zwrotnego również zawiera nazwę, która powinna odpowiadać nazwie odpowiedniego komunikatu synchronicznego, a także opcjonalnie – listę argumentów wyjściowych i wartość zwracaną (np. „OK”). Jeśli obiekt docelowy komunikatu synchronicznego posiada klasę, to etykieta komunikatu powinna odpowiadać jednej z operacji tej klasy. Tę zasadę możemy zaobserwować porównując rysunki 6.16 i 6.15. Klasa „Silnik” posiada publiczne operacje „włącz()” i „wyłącz()”, dzięki czemu możliwe jest kierowanie do obiektów tej klasy komunikatów przez obiekty zewnętrzne (tu: obiekt klasy „Samochód”). Zwróćmy szczególną uwagę na to, że aby możliwe było przesyłanie komunikatów między obiektami różnych klas, między tymi klasami powinna istnieć relacja nawigowalna. Przykładem jest relacja agregacji, która jest nawigowalna od klasy „Samochód” do klasy „Silnik”. Jednocześnie, zauważmy, że nawigowalność dotyczy jedynie możliwości przesłania komunikatu synchronicznego, a nie komunikatu zwrotnego. Nawigowalność nie dotyczy bowiem kierunku przesyłania danych, a jedynie możliwości dostępu do uruchamiania odpowiednich operacji.

Trzecim rodzajem komunikatów są **komunikaty asynchroniczne** (ang. Asynchronous Message), które oznaczamy strzałką z pustym grotem. Znaczenie komunikatów asynchronicznych wyjaśnia diagram po prawej stronie rysunku 6.16. Wysłanie komunikatu „ogrzej(liczba)” powoduje powstanie dwóch ścieżek sterowania (wątków), oznaczonych żetonami „1” i „2”. Obydwie ścieżki działają równolegle, co oznacza, że obydwa obiekty mogą wykonywać swoje operacje w tym samym czasie. W przeciwnieństwie do sytuacji z komunikatem synchronicznym, tutaj obiekt „mój:Samochód” nie czeka na zakończenie działania operacji „ogrzej()”. W naszym przykładzie, obiekt „kierowcy:Siedzenie” uruchamia operację „włącz_grzałkę()”, a w tym samym czasie obiekt „mój:Samochód” uruchamia operację „rozpoznaj_klucz()”. Przy okazji widzimy notację komunikatów kierowanych przez obiekty do samych siebie. Zauważmy, że takie komunikaty mogą odpowiadać operacjom prywatnym odpowiednich klas (patrz rysunek 6.15). Ważną cechą komunikatów asynchronicznych jest brak związanych z nimi komunikatów zwrotnych. Wynika to z tego, że po utworzeniu nowej ścieżki sterowania (w naszym przykładzie oznaczonej „2”) nie następuje powrót sterowania.

Na diagramach sekwencji możemy również definiować cykl życia obiektów. Obiekty mogą być tworzone i niszczone w wyniku przesyłania komunikatów od innych obiektów. Przykład takich komunikatów widzimy na rysunku 6.17. Pierwszy z komunikatów jest **komunikatem utworzenia** (ang. Create Message). Oznaczamy go strzałką przerywaną prowadzącą od jakiejś linii życia (tu: „fabryka”) do ikony

obiektu rozpoczynającej inną linię życia (tu: „mój:Samochód”). Ikona obiektu jest rysowana na poziomie komunikatu, co ma symbolizować utworzenie obiektu w wyniku tego komunikatu. Warto zwrócić uwagę na to, że jeśli linia życia zaczyna się na samej górze diagramu, to zakładamy powstanie danego obiektu wcześniej (przed wykonaniem danego diagramu sekwencji). Zakończenie życia (zniszczenie) obiektu zaznaczamy znakiem „X” umieszczonym na końcu linii życia. Najczęściej, obiekt kończy życie w wyniku otrzymania komunikatu, co ilustruje rysunek 6.17 (komunikat „skasuj”).

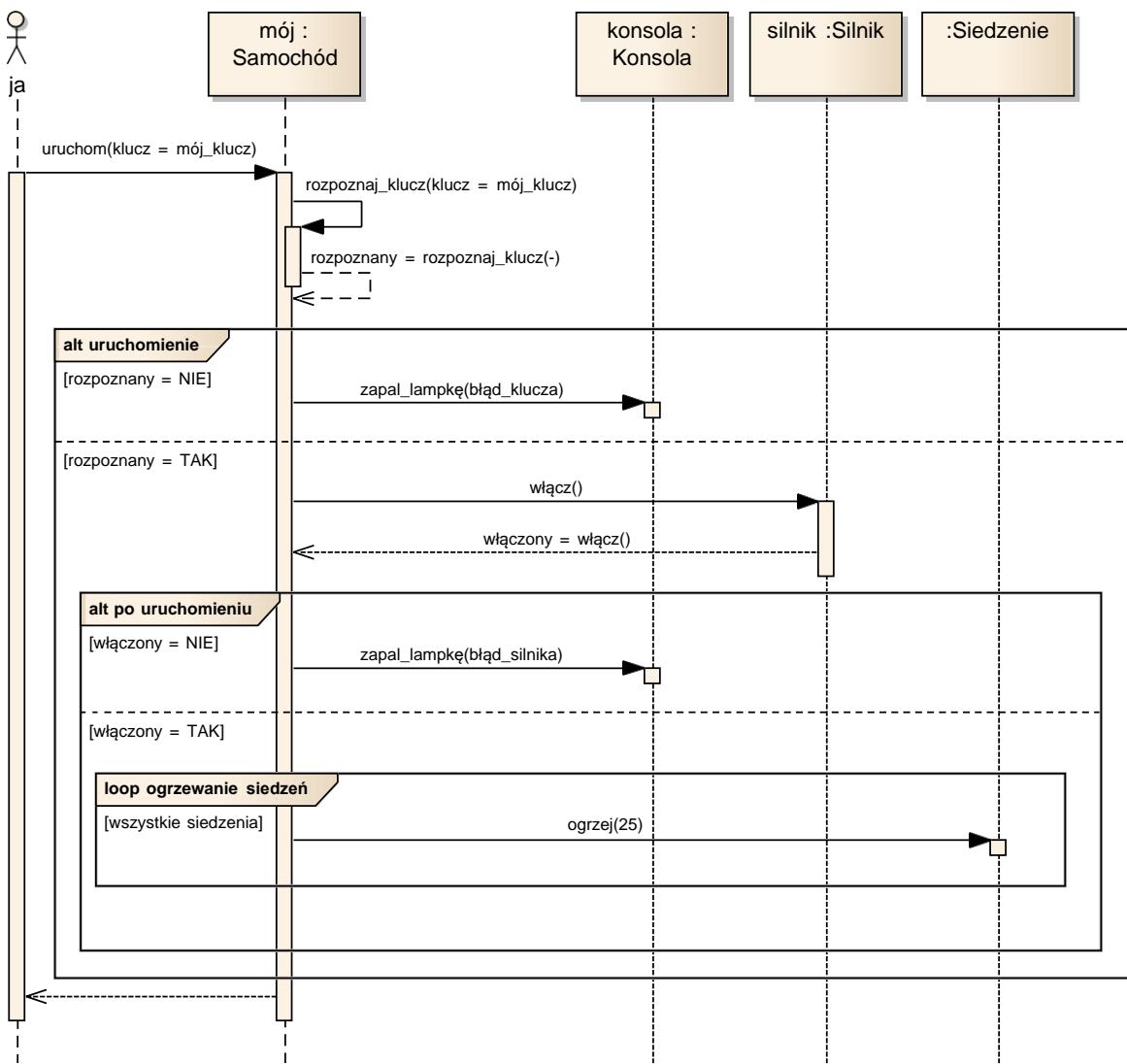


Rysunek 6.17: Tworzenie i kasowanie obiektów na diagramie sekwencji

Na diagramach sekwencji możemy modelować przebiegi alternatywne oraz pętle. Używamy wtedy tzw. **fragmentów łączonych** (ang. Combined Fragment) wyrażanych za pomocą prostokątnych ramek obejmujących odpowiednie komunikaty. Typ ramki określony jest w polu znajdującym się w lewym górnym rogu ramki. Najczęściej stosowane są ramki typu „alt”, „opt” i „loop”. Po nazwie typu ramki można umieścić jej nazwę.

Przykłady ramek „alt” i „loop” widzimy na rysunku 6.18. Najbardziej zewnętrzna jest ramka „alt” o nazwie „uruchomienie”. Ramka taka podzielona jest przerywanymi poziomymi liniami na kilka części (tutaj – na dwie). Każda część oznaczona jest warunkiem umieszczonym w nawiasach kwadratowych. Działanie ramki polega na tym, że wykonywane są komunikaty zawarte w obszarze, dla którego spełniony jest warunek. W naszym przykładzie, przed wykonaniem ramki wykonywana jest przez samochód operacja „rozpoznaj_klucz”. Wynik tej operacji (zmienna „rozpoznany”) jest sprawdzany przez warunki ramki „uruchomienie”. Jeśli spełniony jest pierwszy warunek („rozpoznany = NIE”), wykonywana jest operacja „zapal_lampkę(błąd_klucza)”. Jeśli spełniony jest drugi warunek, wykonywana jest sekwencja komunikatów zawartych w drugiej części ramki. Podobnie działa ramka „alt” o nazwie „po uruchomieniu”. Tym, razem, badana jest wartość wyniku operacji „włącz” wykonywanej przez silnik.

Ostatnią (tutaj – najbardziej wewnętrzną) ramką na rysunku 6.18 jest ramka „loop” (pętla) o nazwie „ogrzewanie siedzeń”. Ramka taka oznacza wykonanie zawartej w niej sekwencji komunikatów dopóki spełniony jest określony warunek. W naszym przykładzie, warunek oznacza wykonanie pętli „dla wszystkich siedzeń” (zgodnie z diagramem klas, może ich być od 2 do 5). Oznacza to, że do każdego obiektu klasy „Siedzenie” przesyłany jest komunikat „ogrzej”. Zwrócmy uwagę, że w tym przypadku linia życia nie ma nazwy (przed dwukropkiem), a jedynie typ („Samochód”).



Rysunek 6.18: Przykładowy diagram sekwencji z ramkami „alt” i „loop”

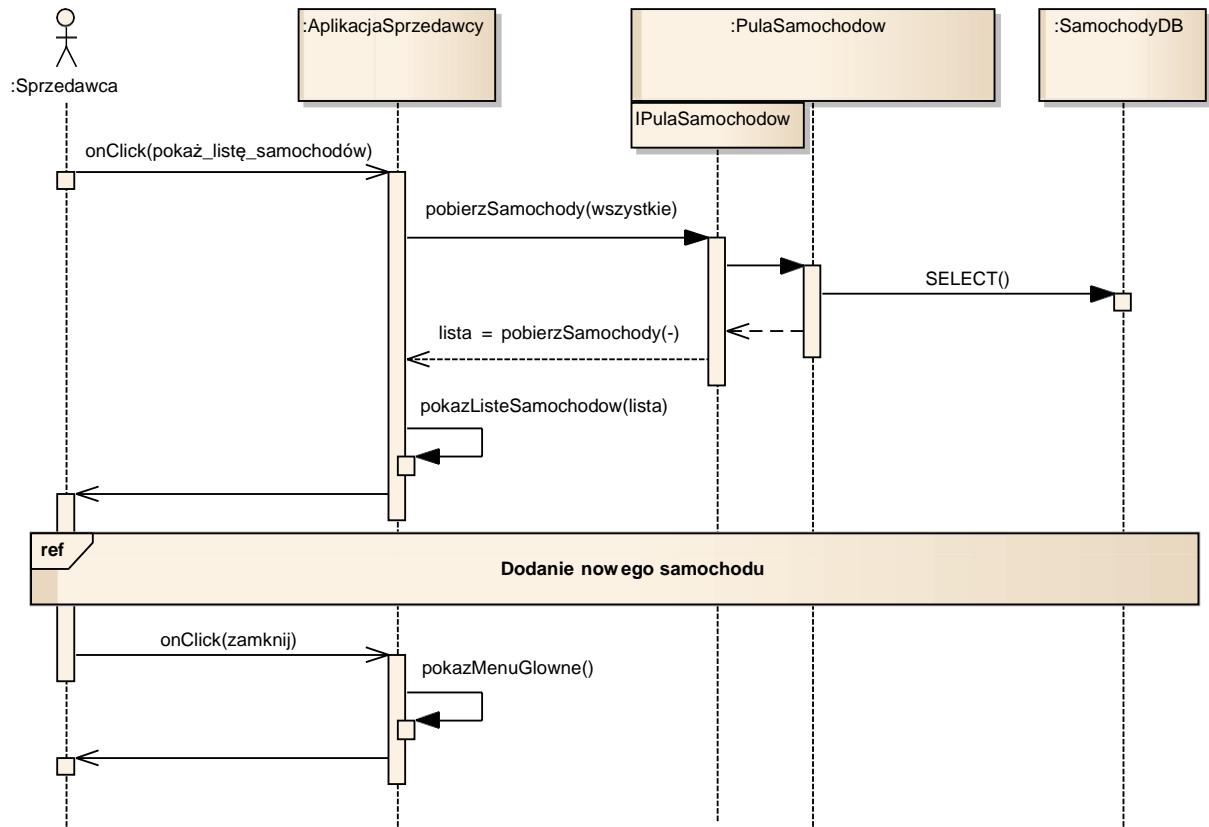
Na koniec omawiania rysunku 6.18 zwróćmy jeszcze uwagę, że linia życia może dotyczyć obiektu klasy aktora. W takiej sytuacji, linia życia oznaczana jest na górze ikoną aktora z odpowiednią nazwą. Linie życia dla aktorów występują najczęściej na diagramach sekwencji, które stanowią realizację przypadków użycia. Aby to wyjaśnić posłużymy się zatem takim diagramem, przedstawionym na rysunku 6.19. Pokażemy dwa diagramy, które realizują dwa przykładowe przypadki użycia połączone relacją «invoke». Pozwoli to nam opisać kilka kolejnych elementów diagramów sekwencji. W szczególności, pokażemy w jaki sposób tworzyć diagramy sekwencji na poziomie komponentów.



Rysunek 6.19: Diagram przypadków użycia realizowany diagramami sekwencji

W poprzednich przykładach, diagramy sekwencji były tworzone na poziomie modelu klas. Często jednak korzystne jest pokazanie dynamiki wymiany komunikatów między komponentami, m.in. za pośrednictwem ich interfejsów. Odpowiedni przykład widzimy na rysunku 6.20. Diagram ten stanowi realizację przypadku użycia „Pokazanie listy samochodów”. Jest on oparty na modelu komponentów

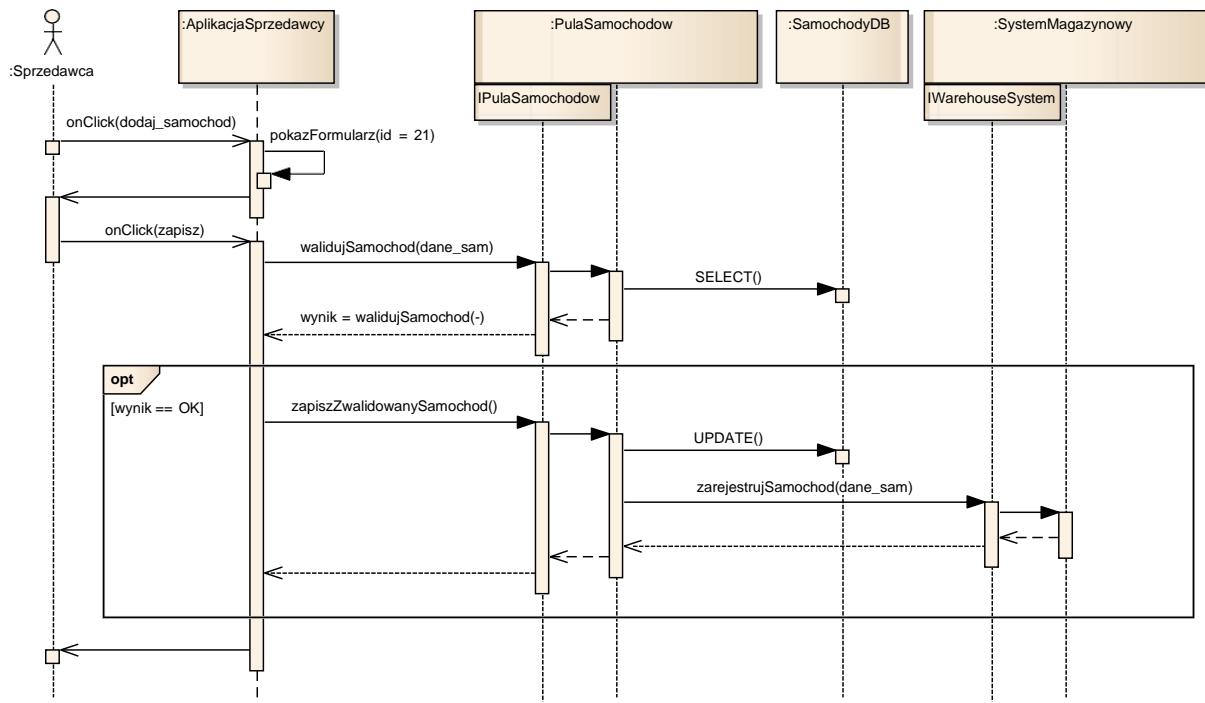
zawartym w poprzednim rozdziale (por. odpowiednie rysunki). Etykiety komunikatów odpowiadają operacjom interfejsów przedstawionych na diagramie definiującym komponent „PulaSamochodow”. Istotnym elementem notacyjnym jest linia życia interfejsu „IPulaSamochodow”, która jest „zagnieżdżona” w linii życia obiektu komponentu „PulaSamochodow”.



Rysunek 6.20: Realizacja przypadku użycia „Pokazanie listy samochodów”

Sekwencja rozpoczyna się od przesłania do obiektu komponentu „AplikacjaSprzedawcy” asynchronicznego komunikatu „onClick”, który odpowiada zdarzeniu wybrania przez aktora (sprzedawcę) odpowiedniej opcji („pokaż listę samochodów”) w menu. Następnie, za pośrednictwem interfejsu, do obiektu „:PulaSamochodow” przesyłany jest komunikat „pobierzSamochody”. W ramach realizacji tej operacji, następuje przesłanie komunikatu do obiektu bazowanego „:SamochodyDB”. Ponieważ komponent ten nie posiada interfejsu, komunikat (zapytanie SQL) jest przesyłany bezpośrednio do obiektu. Po zakończeniu wykonywania operacji „pobierzSamochody”, zwracana jest odpowiednia lista. Jest ona następnie wyświetlana na ekranie poprzez wywołanie wewnętrznej operacji komponentu „AplikacjaSprzedawcy” (komunikat „pokazListeSamochodow”). Potem, sterowanie przekazywane jest (asynchronicznie) z powrotem do aktora. Może on np. wybrać opcję „zamknij” (kolejny komunikat „onClick”), co spowoduje pokazanie menu głównego i zakończenie działania tego wykonania przypadku użycia.

Charakterystycznym elementem na rysunku 6.20 jest ramka typu „ref” (referencja) z etykietą „Dodanie nowego samochodu”. Jest to tzw. **użycie interakcji** (ang. Interaction Use). Ramka ta oznacza odniesienie do innego diagramu interakcji (tu: sekwencji). W naszym przykładzie, jest to diagram pokazany na rysunku 6.21, który stanowi realizację drugiego przypadku użycia z rysunku 6.19. Znaczenie tej ramki referencji jest takie, że cały diagram z rysunku 6.21 jest niejako wstawiany w miejsce ramki.



Rysunek 6.21: Realizacja przypadku użycia „Dodanie nowego samochodu”

Rysunek 6.21 ilustruje zastosowanie ramki „**opt**” (opcja). Jest to niejako wariant ramki „**alt**” zawierający tylko jeden warunek. Zawartość ramki wykonywana jest tylko w przypadku spełnienia warunku. W przeciwnym razie cała zawartość ramki jest pomijana w wykonaniu danej sekwencji.

Analizując diagram na rysunku 6.21, jak również wszystkie diagramy poprzednie, warto zwrócić uwagę na ciągłość przepływu sterowania. W typowych zastosowaniach, komunikaty powinny tworzyć spójną sekwencję, gdzie kolejne komunikaty wynikają bezpośrednio z poprzednich. W szczególności, należy zwrócić uwagę, że komunikaty zwrotne muszą zwracać sterowanie do tego samego obiektu, który wysłał poprzedzający komunikat synchroniczny. Ponadto, komunikaty nie powinny pojawiać się „znikąd”, ale mieć swoje źródło w belce wykonania, która wynika z poprzednich komunikatów w sekwencji. Inaczej mówiąc, komunikaty nie powinny występować w miejscach, w których dany obiekt nie otrzymał wcześniej sterowania.

Więcej przykładów diagramów sekwencji znajdziemy w module III – w rozdziałach dotyczących projektowania systemu.

Zadania

Zadanie 1

Stwórz diagram przypadków użycia dla systemu obsługi dziekanatu obejmujący przypadki związane z wystawianiem ocen. Diagram powinien zawierać co najmniej trzy przypadki użycia, co najmniej dwóch aktorów i co najmniej jedną relację pomiędzy przypadkami zgodną z wybraną konwencją («invoke» lub «include»/«extend»).

Zadanie 2

Uzupełnij diagram przypadków użycia z rysunku 6.19 o dwa nowe przypadki użycia, z których co najmniej jeden jest połączony relacją «invoke» z jednym z już istniejących.

Zadanie 3

Stwórz diagram czynności obrazujący proces wypłaty pieniędzy z bankomatu. Wykorzystaj zarówno węzły decyzyjne jak i belki fork/join.

Zadanie 4

Stwórz diagram maszyny stanów obrazujący możliwe zmiany stanu skupienia wody. Nazwij odpowiednio przejścia między stanami.

Zadanie 5

Stwórz diagram sekwencji obrazujący przykładowy przebieg uzgadniania trójetapowego stosowanego przy nawiązywaniu połączenia w protokole TCP.

Słownik pojęć

Akcja

Węzeł w modelu czynności, który definiuje elementarną operację wykonywaną w ramach opisywanego procesu. Akcja oznaczana jest ikoną prostokąta z zaokrąglonymi rogami, wewnętrz umieszczana jest treść akcji.

Aktor

Klasa obiektów spoza modelowanego systemu wchodząca z tym systemem w interakcje. Obiekty te mogą być osobami, urządzeniami lub zewnętrznym systemami informatycznymi. Możemy również powiedzieć, że aktor definiuje rolę graną przez obiekty zewnętrzne w stosunku do danego systemu. Aktor jest reprezentowany przez ikonę człowieka narysowaną prostymi kreskami.

Belka wykonania

Oznaczenie wykonania jakiejś operacji przez obiekt po otrzymaniu komunikatu. Belka wykonania rysowana jest jako wąski prostokąt nałożony na linię życia danego obiektu.

Extend

Relacja oznaczająca możliwość wplecenia treści scenariuszy przypadku użycia rozszerzającego w treść scenariuszy przypadku użycia rozszerzanego. Możliwość wplecenia może podlegać określonym warunkom, które powinny być dołączone do specyfikacji danej relacji. Relacja jest rysowana jak relacja zależności ze stereotypem „extend”.

Fragment łączony

Ramka na diagramie sekwencji, która oznacza pewien fragment diagramu podlegający specjalnemu traktowaniu. Możliwe są m.in. fragmenty oznaczające alternatywę („alt”), opcję („opt”) lub petlę („loop”).

Include

Relacja oznaczająca bezwarunkowe włączenie w treść scenariuszy jednego przypadku użycia, treści scenariuszy innego. Relacja jest rysowana jak relacja zależności ze stereotypem „include”.

Invoke

Relacja między przypadkami użycia mająca semantykę podobną do semantyki wywołania procedury. Zawsze jest skierowana od przypadku wywołującego do przypadku wywoływanego. Wywołanie może być bezwarunkowe lub możemy dla niego określić warunek. Relacja jest rysowana jak relacja zależności ze stereotypem „invoke”.

Komunikat asynchroniczny

Rodzaj komunikatu przesyłanego między obiektami, który nie powoduje zawieszenia działania obiektu wysyłającego komunikat. Komunikat asynchroniczny rysujemy jako strzałkę ze standardowym grotem (bez wypełnienia).

Komunikat synchroniczny

Rodzaj komunikatu przesyłanego między obiektami, który powoduje zawieszenie działania obiektu wysyłającego komunikat do czasu otrzymania komunikatu zwrotnego. Komunikat synchroniczny rysujemy jako strzałkę z wypełnionym grotem.

Komunikat zwrotny

Rodzaj komunikatu powiązanego z komunikatem synchronicznym, który jest przesyłany jako odpowiedź (odwrotnie niż komunikat synchroniczny). Komunikat zwrotny rysujemy jako strzałka przerywana.

Linia życia

Linia na diagramie sekwencji reprezentująca działanie jednego obiektu. Linia życia rysowana jest pionową przerywaną kreską.

Przypadek użycia

Definicja klasy zachowań modelowanego systemu (tzw. podmiotu), prowadzących do osiągnięcia obserwowalnego, istotnego rezultatu dla jakiegoś aktora (lub aktorów). Przypadek użycia jest rysowany jako elipsa z nazwą.

Pseudostan początkowy

Oznaczenie stanu, w którym rozpoczyna się działanie maszyny stanów. Pseudostan początkowy oznaczany jest jako czarne kółko wskazujące na stan początkowy.

Pseudostan terminalny

Oznaczenie możliwych stanów, w których znajduje się maszyna stanów przez jej skasowaniem (zakończeniem działania). Pseudostan terminalny oznaczany jest jako duży znak „X”.

Punkt rozszerzenia

Miejsce w scenariuszu przypadku użycia, w którym następuje wplecenie elementów scenariuszy innego przypadku użycia.

Rozwidlenie

Węzeł w modelu czynności, który posiada jeden przepływ wchodzących oraz wiele przepływów wychodzących. Po dotarciu żetony do przepływu wchodzącego jest on natychmiast kopiowany i przekazywany do wszystkich przepływów wychodzących. Węzeł rozwidlenia rysowany jest w postaci belki.

Stan

Określenie pewnej stabilnej konfiguracji systemu (np. obiektu), która może trwać w czasie. Stan reprezentowany jest za pomocą ikony prostokąta z zaokrąglonymi rogami i nazwą.

Węzeł decyzyjny

Węzeł w modelu czynności, który posiada jeden przepływ wchodzących oraz wiele przepływów wychodzących. Po dotarciu żetony do przepływu wchodzącego jest on natychmiast przekazywany do jednego przepływu wychodzącego – spełniającego przypisany mu warunek. Węzeł decyzyjny rysowany jest w postaci rombu.

Węzeł końcowy

Węzeł w modelu czynności, który posiada jedynie przepływy wchodzące. Po dotarciu do takiego węzła żetonu, cały proces jest zatrzymywany (kasowane wszystkie żetony). Wariantem węzła końcowego jest węzeł końca przepływu, który kasuje jedynie docierający do niego żeton.

Węzeł scalenia

Węzeł w modelu czynności posiadający wiele przepływów wejściowych i jeden przepływ wyjściowy. Żeton docierający na jedno z wejść takiego węzła jest natychmiast transportowany na wyjście. Węzeł scalenia rysowany jest w postaci rombu.

Złączenie

Węzeł w modelu czynności posiadający wiele przepływów wejściowych i jeden przepływ wyjściowy. Po dotarciu żetonów na wszystkie wejścia tego węzła, są one łączone i jeden żeton jest natychmiast transportowany na wyjście. Węzeł złączenia rysowany jest w postaci belki.

Co trzeba zapamiętać

Model przypadków użycia

Za pomocą diagramów przypadków użycia możemy opisywać zewnętrzny sposób zachowania się systemu, tzn. jego funkcjonalność z punktu widzenia środowiska (głównie – użytkowników). Model ten zawiera przypadki użycia realizowane na rzecz aktorów. W modelu przypadków użycia uwzględniamy również relacje między przypadkami użycia, które oznaczają wzajemne włączanie, rozszerzanie lub wywoływanie funkcjonalności.

Model czynności

Diagramy czynności stanowią grafy skierowane opisujące sieci akcji wraz z odpowiednimi przepływami sterowania między akcjami. Model ten realizuje częstą potrzebę podczas modelowania systemów, jaką jest przedstawienie jakiegoś procesu w postaci schematu blokowego. Oznacza to konieczność zbudowania modelu składającego się z akcji oraz przepływów sterowania między akcjami, czyli modelu

czynności. Modele czynności mogą opisywać algorytmy, procesy biznesowe, procesy fizyczne lub scenariusze przypadków użycia, które też tworzą pewnego rodzaju proces. W modelu czynności występują węzły sterujące (decyzyjne, rozwidlenia itd.) umożliwiające zmianę przepływu sterowania.

Model maszyny stanów

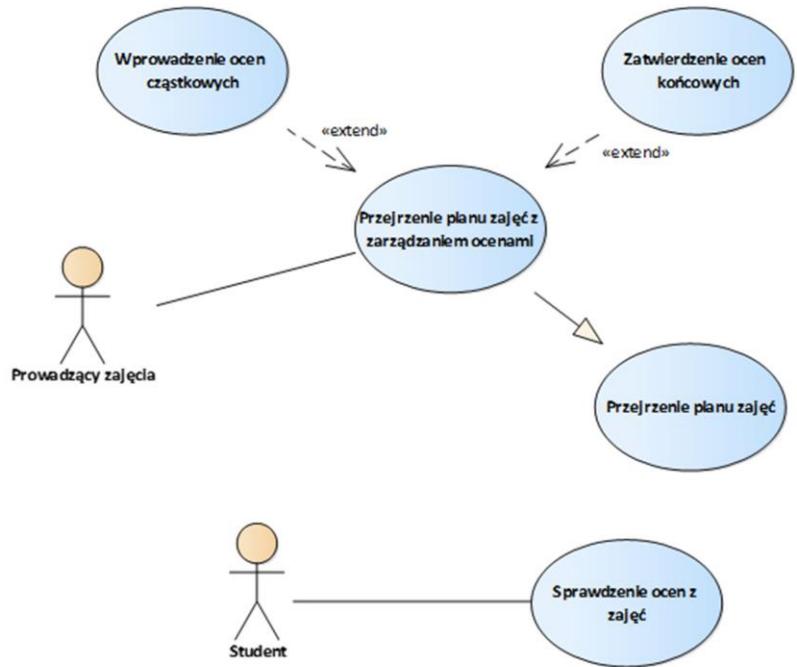
Diagramy maszyny stanów służą do pokazywania sposobu zachowania się jakiegoś elementu struktury systemu (np. klasy obiektów) w postaci przejść pomiędzy jego stanami, czyli stabilnymi konfiguracjami. Model maszyny stanów jest bardzo podobny w swojej notacji do modelu czynności. Obydwa rodzaje diagramów posługują się notacją grafu. Podstawowa różnica jest taka, że węzły w diagramach maszyny stanów definiują stany, a nie akcje.

Model sekwencji

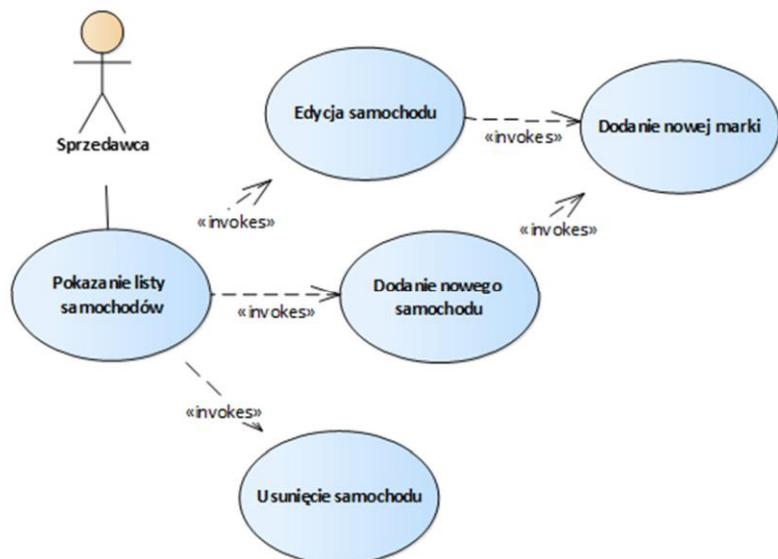
Diagramy sekwencji pokazują sekwencje komunikatów przesyłanych między obiekty w ramach działającego systemu. Komunikaty oznaczają polecenia wykonania określonych usług, opisują przepływ sterowania między obiekty oraz przepływ danych (parametry komunikatów). Modele sekwencji zwyczaj tworzymy w kontekście konkretnej struktury systemu wyrażonej modelem klas lub modelem komponentów. Komunikaty wymieniane między obiekty mogą odpowiadać konkretnym operacjom klas lub interfejsów komponentów. Ponadto, wymiana komunikatów powinna się odbywać zgodnie z nawigowalnością relacji między klasami lub komponentami.

Rozwiązania zadań

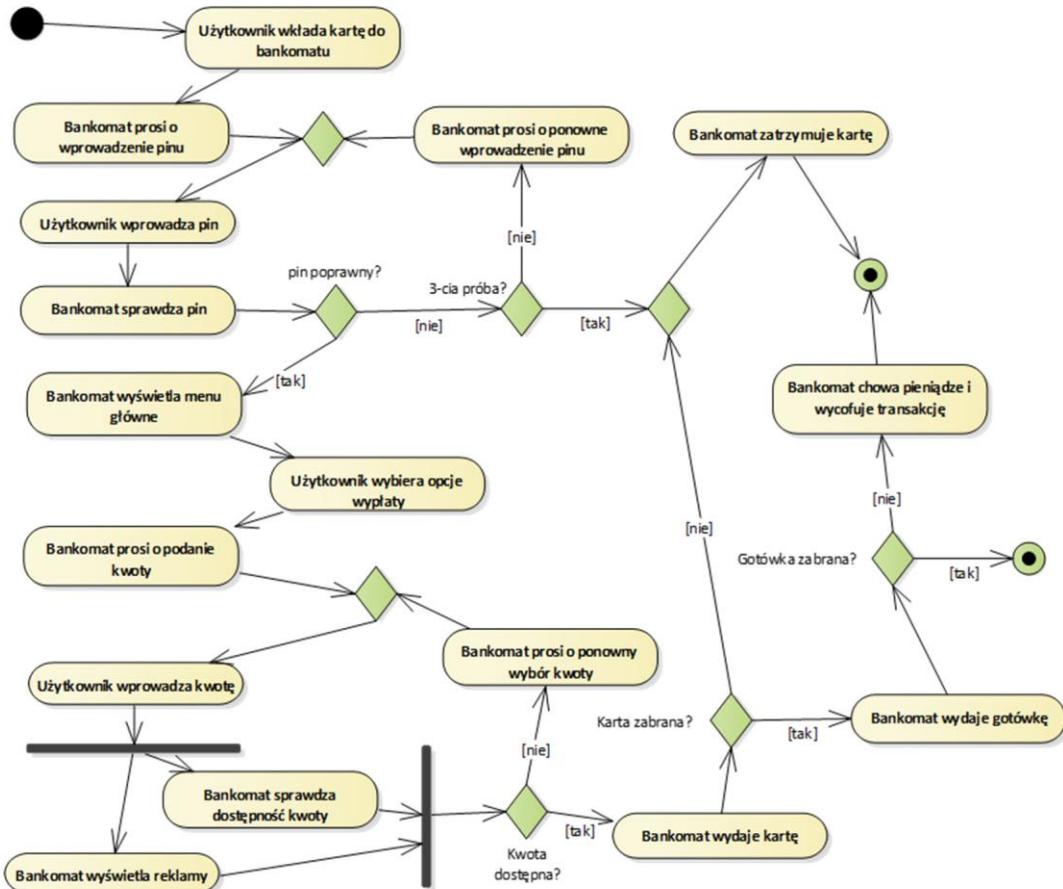
Rozwiązanie zadania 1



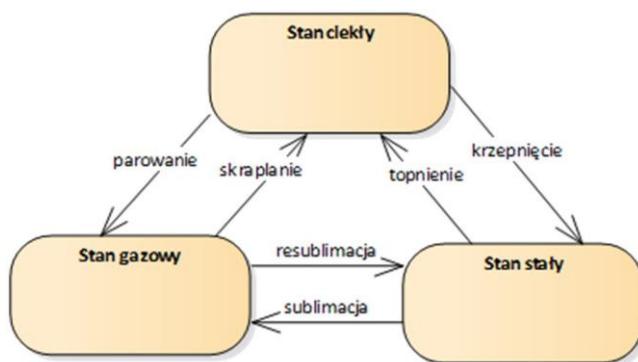
Rozwiązanie zadania 2



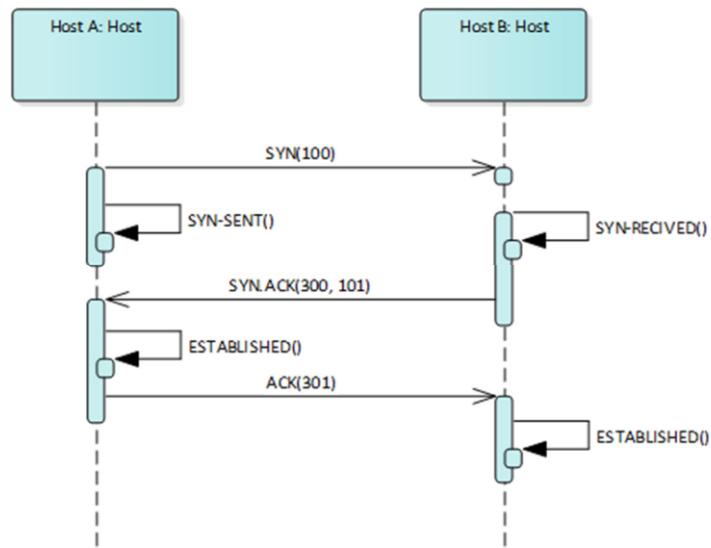
Rozwiązań zadania 3



Rozwiązań zadania 4



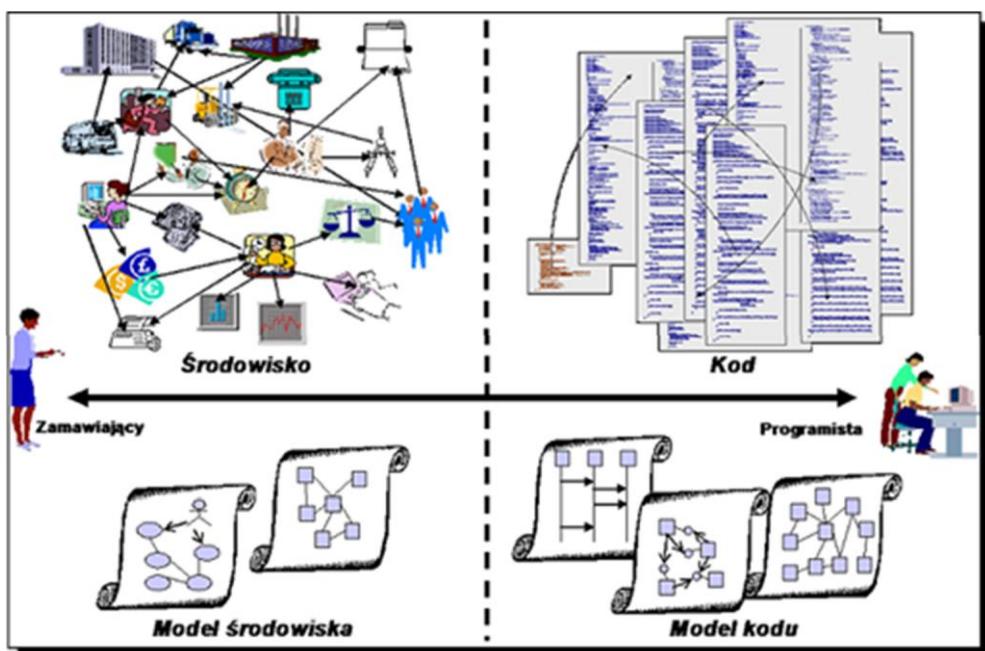
Rozwiązanie zadania 5



7. Wprowadzenie do inżynierii wymagań

7.1. Rola wymagań w inżynierii oprogramowania

Celem każdego projektu konstrukcji oprogramowania jest wykonanie systemu oprogramowania spełniającego wymagania sponsora projektu w określonych ramach czasowych i budżetowych. Techniki inżynierii oprogramowania mają na celu pomóc w pokonaniu niewątpliwej złożoności współczesnych systemów oprogramowania. Problemem jest niewątpliwie rozmiar kodu, który niejednokrotnie sięga setek tysięcy instrukcji (nawet dla systemów o średnich rozmiarach). Jednak, inżynieria oprogramowania to nie tylko programowanie i panowanie nad złożonością kodu. Zadaniem inżynierii oprogramowania jest bowiem także zarządzanie złożonością potrzeb klienta (zamawiającego system oprogramowania). Zostało to zilustrowane na rysunku 7.1. Złożoność kodu w dużym stopniu zależy od złożoności potrzeb zamawiającego wynikającej ze złożonością środowiska (np. firmy). Jednocześnie należy zapewnić zgodność kodu z wymaganiami (potrzebami) zamawiającego oprogramowanie (klienta). Ważne jest dotarcie do rzeczywistych potrzeb. Powodzenie projektu równoznaczne jest ze spełnieniem potrzeb zamawiającego. Oczywiście, bardzo ważne jest, żeby zamawiany system został oddany na czas i jego koszt był zgodny z zaplanowanym budżetem. Jednak, najważniejsze jest, żeby system działał tak, jak tego wymaga zamawiający.



Rysunek 7.1: Zależność kodu od potrzeb zamawiającego

Podstawowym pojęciem inżynierii wymagań jest oczywiście „wymaganie”. Przed przystąpieniem do omawiania inżynierii wymagań musimy zatem zdefiniować to pojęcie.

Wymaganie to jako własność produktu końcowego (systemu oprogramowania), którą musi on posiadać, aby spełnić oczekiwania zamawiającego. Właściwością tą może być określony sposób funkcjonowania systemu lub cecha jakościowa narzucona na system. Mówiąc, że system spełnia wymagania, jeśli zostało potwierdzone, że posiada wszystkie właściwości określone tymi wymaganiami.

Na tej podstawie możemy również określić, czym jest **inżynieria wymagań**. Jest to dyscyplina inżynierii oprogramowania, która obejmuje działania polegające na zbieraniu, analizowaniu, negocjowaniu i specyfikowaniu (zapisywaniu) wymagań. Głównym produktem tej dyscypliny jest specyfikacja wymagań, która jest podstawą efektywnego kosztowo wytwarzania systemu oprogramowania zgodnego z oczekiwaniemi zamawiającego.

Zgodnie z definicją, wymagania odpowiadają na najistotniejsze w całym procesie budowy oprogramowania pytanie, „jaki system mamy zbudować?” W dobrze zorganizowanym procesie twórczym, wymagania sterują konstrukcją systemu. Wszystkie działania związane z projektowaniem, implementacją oraz wdrożeniem systemu są podporządkowane spełnieniu wymagań. Ostatecznym potwierdzeniem spełnienia wymagań jest przejście odpowiedniego zestawu testów akceptacyjnych. Dobre skonstruowany system spełnia wszystkie testy przeprowadzone przy udziale zamawiającego. Kluczem tutaj jest potwierdzenie przez zamawiającego, że wyrażone wymaganiem potrzały zostały spełnione.

Można zatem powiedzieć, że jakość wymagań jest kluczowym elementem powodzenie każdego projektu konstrukcji systemu oprogramowania. Źle sformułowane wymagania są bardzo częstą przyczyną niepowodzeń. Ogólnie, można powiedzieć, że dobre sformułowanie wymagań polega na dobrym odzwierciedleniu rzeczywistych potrzeb zamawiającego. W szczególności, dobrej jakości wymagania powinny charakteryzować się kilkoma podstawowymi cechami.

Wymagania powinny być **kompletne**. Oznacza to, że powinny one obejmować cały zakres potrzeb zamawiającego opisany w niezbędnych szczegółach. Aby zapewnić kompletność wymagań należy nieustannie zadawać sobie oraz zamawiającemu pytanie, czy czegoś nie pomineliśmy. Można tutaj pomagać sobie stosowaniem odpowiednich technik odkrywania wymagań, jak burze mózgów lub warsztaty interaktywne. Kompletność specyfikacji wymagań oznacza również uwzględnienie wszystkich obszarów wymagań, o których będziemy mówili w dalszej części tego rozdziału.

Drugą istotną cechą wymagań jest ich **jednoznaczność i poprawność**. Takie wymagania powinny definiować jeden możliwy zakres systemu, i nie pozostawiać pola do różnych interpretacji. Aby zapewnić jednoznaczność i poprawność, należy stale upewniać się, że wymagania są przejrzyste dla zamawiającego, ale również – że są zrozumiałe dla deweloperów. Należy stale dążyć do uzyskania informacji zwrotnej, czy dobrze rozumiemy to, co przekazuje nam zamawiający. W cyklu iteracyjnym dobrym testem na przejrzystość wymagań jest zrealizowanie pierwszego przyrostu systemu. Jeśli klient potwierdzi, że spełnia on określony wymaganiem zakres w stopniu dobrym, to mamy dobre potwierdzenie jednoznaczności wymagań.

Wymagania powinny być również **spójne**, czyli niesprzeczne. Eliminacja sprzeczności polega na ciągłym wzajemnym porównywaniu wymagań ze sobą, szczególnie wymagań różnych rodzajów. Sprzeczności często prowadzą do problemów podczas realizacji systemu. Mogą one powstać w wyniku tego, że specyfikacja jest tworzona przez zespół i brak wewnętrznego porozumienia w zespole (w tym, w zespole reprezentującym zamawiającego). Dobrym sposobem na eliminację sprzeczności jest ciągła aktualizacja wspólnego słownika. Jednoznaczne słownictwo pozwala uniknąć nieporozumień i zapewnić spójność różnych obszarów specyfikacji wymagań. Ważne jest szczególnie zwrócenie uwagi na wymagania jakościowe oraz ograniczenia techniczne. Niektóre ograniczenia (np. konieczność stosowania określonego sprzętu) mogą w ewidentny sposób stać w sprzeczności z wymaganiami jakościowymi (np. wydajność przetwarzania danych).

Kolejna cecha dobrej specyfikacji wymagań to **możliwość zarządzania**. Wymagania powinny być podzielone na dobrze określone jednostki, które mogą sterować procesem twórczym. Powinny one być na tyle małe, aby można było je przydzielać do wykonania w stosunkowo krótkich odcinkach czasu (np. w ciągu jednej iteracji/sprincie). Jednocześnie, poszczególne wymagania powinny mieć przypisane

odpowiednie atrybuty pozwalające na efektywne przydzielania ich do wykonania w trakcie projektu. Takie atrybuty mogą obejmować m.in. unikalny identyfikator, priorytet dla użytkownika, poziom trudności technicznej, osobę odpowiedzialną oraz wersję. Łatwość zarządzania to również łatwość śledzenia produktów wynikających z wymagań. Wymagania na różnych poziomach powinny z siebie w jednoznaczny sposób wynikać. To samo dotyczy również produktów projektowych oraz kodu. Każda zmiana wymagania powinna w jednoznaczny sposób prowadzić do innych produktów (a ostatecznie – kodu), które wymagają zmiany.

Bardzo istotną, można powiedzieć – kluczową cechą wymagań jest ich **testowalność** (mierzalność). Wymaganie, którego nie jesteśmy w stanie przetestować jest nic niewarte. Stosunkowo prosto jest sformułować testy dla wymagań funkcjonalnych. Zazwyczaj polegają one na opracowaniu odpowiednich scenariuszy testowych. Sprawa jest trudniejsza w przypadku wymagań jakościowych. Tutaj konieczna jest indywidualizacja podejścia do testowania takich wymagań w zależności od ich szczegółowogo rodzaju. Konieczne jest opracowanie odpowiednich metryk, które pozwolą na jednoznaczne potwierdzenia spełnienia takich wymagań. O metrykach mówimy w sposób bardziej szczegółowy w następnym rozdziale niniejszego modułu.

Warto na koniec rozważyć o jakości wymagań zauważyc, że spełnienie tak określonych wymagań oznacza zadowolonego zamawiającego. Warto też zauważyc, że często system spełnia wszystkie wymagania, a klient (zamawiający) nie jest z niego zadowolony. Jest to najczęściej właśnie wynikiem niekompletnych, mało precyzyjnych, niejednoznacznych czy niemierzalnych wymagań.

Aby sformułować dobrej jakości wymagania musimy uzyskać dostęp do odpowiednich źródeł informacji. Można tutaj wyróżnić źródła osobowe (ludzi) oraz źródła nieosobowe (np. dokumenty). W szczególności warto wziąć pod uwagę następujące źródła wymagań:

- Istniejące procesy – opisy procesów biznesowych obowiązujące w danej organizacji,
- Docelowe procesy – specyfikacja zmodyfikowanych procesów biznesowych, biorących m.in. pod uwagę zastosowanie nowego systemu,
- Podręczniki użytkownika – specyfikacja opisująca funkcjonalność istniejących systemów,
- Zamawiający – osoby decydujące o dokonaniu zmian w organizacji i budowie nowego (lub rozbudowanego) systemu,
- Użytkownicy – osoby, które będą bezpośrednio pracować z systemem.

Ważne jest, aby źródłem wymagań były osoby (zamawiający, użytkownicy) dobrze umocowane i reprezentatywne. W szczególności mogą to być kluczowi użytkownicy (np. kierownicy zespołów, doświadczeni konsultanci wewnętrzni, mentorzy) oraz kluczowi sponsorzy projektu (kierownicy wysokiego szczebla, którym zależy na powodzeniu projektu).

Pisząc wymagania należy pamiętać o tym, że będą z nich korzystać bardzo różnorodne grupy osób. Możemy tu wyróżnić osoby po stronie zamawiającego (np. firma zamawiająca), jak i po stronie wykonawcy (np. firma wykonująca system). Rolą osób specyfikujących wymagania jest pogodzenie oczekiwani tak różnorodnych grup w stosunku do specyfikacji. Warto zauważyc, że dobrą praktyką byłoby, aby osoby formułujące specyfikację wymagań nie były zależne zarówno od zamawiającego, jak i wykonawcy. Dzięki temu, specyfikacja nie byłaby obarczona określonym skrzywieniem spowodowanym punktem widzenia specyfikującego wymaganie. Niestety, jest to stosunkowo rzadko spotykana sytuacja. Niezależnie jednak od tego, należy brak uwagi różnorodność odbiorców.

- Grupy osób po stronie zamawiającego:
 - Sponsorzy – podejmują decyzję o potrzebie zamówienia systemu,

- Współpracownicy zamawiającego (klienci, dostawcy, ...) – będą musieli współpracować z systemem budowanym dla zamawiającego,
 - Przygotowujący przetargi – muszą zorganizować przetarg na budowę systemu zgodnego z wymaganiami,
 - Użytkownicy końcowi – będą bezpośrednio korzystać z systemu.
- Grupy osób po stronie wykonawcy:
 - Przygotowujący oferty – muszą złożyć ofertę na budowę systemu zgodnego z wymaganiami,
 - Deweloperzy – muszą zaprojektować i zrealizować zadany system,
 - Przygotowujący testy akceptacyjne – muszą napisać testy, które zweryfikują spełnienie wymagań,
 - Kierownicy projektów – muszą pokierować realizacją systemu tak, aby był zgodny z wymaganiami.

Może się wydawać, że zadanie napisania zrozumiałej dla wszystkich tych grup specyfikacji wymagań wydaje się być zadaniem karkołomnym. Wynika to z różnego przygotowania technicznego tych osób, jak i bardzo różnej wiedzy na temat budowy systemów oprogramowania. Osoby bez przygotowania informatycznego często nie rozumieją sformułowań „branżowych”. Z drugiej strony, informatycy nie zawsze rozumieją terminologię danej dziedziny problemu, którą posługują się zamawiający (np. terminologię finansową). Stosowanie dobrych praktyk inżynierii wymagań pozwala jednak pokonać ten problem lub w istotny sposób ograniczyć jego skutki.

Bardzo istotnym składnikiem inżynierii wymagań jest **zarządzanie wymaganiami**. Polega na ciągłym upewnianiu się, że rozwiązujeśmy właściwy problem i budujemy właściwy system. Dzieje się to poprzez systematyczne podejście do: 1) zbierania, 2) organizowania, 3) dokumentowania, 4) wykorzystywania, zmieniających się wymagań na system oprogramowania. Zarządzamy wymaganiami ponieważ nie są one oczywiste i pochodzą z wielu źródeł. Poza tym, zarządzanie dotyczy różnych rodzajów wymagań oraz różnych poziomów ich szczegółowości. W typowym systemie oprogramowania możemy wyróżnić setki lub nawet tysiące wymagań, które musimy w odpowiedni sposób uporządkować. Należy również zawsze pamiętać, że wymagania podlegają ciągłym zmianom, wynikającym np. ze zmian otoczenia biznesowego lub warunków w ramach organizacji zamawiającej oprogramowanie.

Kluczowe w zarządzaniu wymaganiami jest umiejętnie podzielenie całego zbioru wymagań klienta na odpowiednie jednostki, czyli pojedyncze wymagania. Każde wymaganie to jedna charakterystyka systemu, który mamy zbudować. Aby móc efektywnie zarządzać wymaganiami musimy je mieć w jednoznaczny sposób identyfikować. Wymaganie powinno mieć nazwę (jedno krótkie zdanie) oraz treść (reprezentację, w postaci kilku akapitów tekstu, czy niedużego modelu graficznego). Wymagania powinny także mieć odpowiedni zbiór atrybutów, które ułatwiają zarządzanie nimi. Bardzo pomocne jest, na przykład, nadanie wymaganiom numerów identyfikacyjnych, priorytetów, czy określenie osób za nie odpowiedzialnych.

Bardzo istotne jest również utrzymywanie śladu między wymaganiami. Ponieważ wymagania mogą mieć różne poziomy szczegółowości, należy utrzymywać związek między wymaganiami ogólnymi i wynikającymi z nich wymaganiami szczegółowymi. Również bardzo ważne jest utrzymywanie relacji między różnymi wymaganiami na tym samym poziomie szczegółowości (np. wyjaśnianie wszystkich pojęć w słowniku). W utrzymywaniu śladu pomaga wyraźne określenie poziomów i rodzajów wymagań o czym mówimy w dalszej części tego rozdziału.

Zarządzanie wymaganiami to również zarządzanie zmianami wymagań. Jest to nieodłączny element każdego projektu. Zmiany wymagań wynikają ze zmieniających się warunków prawnych, biznesowych

czy organizacyjnych. Zmiany mogą też wynikać z niezdecydowania klienta lub niezrozumienia jego potrzeb. Mogą też wynikać z polityki wewnętrz organizacji zamawiającej oprogramowanie. Niezależnie od przyczyn, należy podkreślić, że praktycznie nie istnieją nietrywialne projekty, podczas których nie dochodzi do zmian wymagań. Niestety, czasami nawet najmniejsza zmiana sformułowania wymagań może rodzić bardzo poważne konsekwencje dla realizowanego systemu. Na przykład, zmiana treści z „1 milisekunda” na „10 milisekund” może powodować kilkukrotny wzrost kosztów końcowego systemu. Dlatego też, każda zmiana wymagań powinna być traktowana z bardzo dużą uwagą. Warto wprowadzić odpowiednie procedury, regulujące tryb wprowadzania zmian wymagań (proces kontroli zmian). Również w projektach, w których stosowane są zwinne (agile) metodyki twórcze, należy dbać o stałą kontrolę zakresu systemu i oceniać wpływ dokonywanych zmian na ten zakres.

Bardzo istotnym składnikiem zarządzania wymaganiami jest dokumentowanie wymagań. Specyfikacje wymagań tradycyjnie sporządza się w formie dokumentów (głównie tekstowych). Warto jednak stosować nowoczesne metody specyfikowania wymagań w formie modeli graficznych. Współczesne narzędzia wspomagające modelowanie wymagań umożliwiają automatyczną generację dokumentacji w formie dokumentu. Jednocześnie, narzędzia te ułatwiają utrzymywanie śladu między wymaganiami (poziom wymagań zamawiającego i poziom wymagań oprogramowania) oraz śledzenie realizacji wymagań przez system (projekt systemu).

Podsumowując rolę inżynierii wymagań można podkreślić, że powinniśmy stosować dobre praktyki tej dyscypliny, ponieważ: 1) wymagania nie są oczywiste i mogą pochodzić z wielu źródeł, 2) wymagania są różnego rodzaju i mają różnych poziom szczegółowości, 3) wymagań jest bardzo dużo i trudno je wszystkie objąć umysłem, 4) wymagania zależą od siebie nawzajem i od innych produktów procesu wytwarzania oprogramowania, 5) wymagania mają różne właściwości, np. znaczenie dla systemu, 6) wymagania muszą pogodzić różne zainteresowane strony, 7) wymagania się zmieniają (niestety!).

7.2. Specyfikowanie środowiska systemu

W dzisiejszych czasach elementem niemal każdej organizacji biznesowej jest jakiś system oprogramowania. Systemy oprogramowania wspierają co najmniej część działalności przedsiębiorstw, a w wielu przypadkach są fundamentem tej działalności, jak np. w przypadku handlu elektronicznego. Wdrożenie w przedsiębiorstwie nowego lub rozbudowanie dotychczasowego systemu oprogramowania często powoduje istotne zmiany w sposobie jego organizacji oraz sposobie realizacji procesów biznesowych. Wprowadzenie takich zmian jest zazwyczaj kosztowne. Nowy lub przebudowany system powinien zatem, w najbardziej zbliżony do optymalnego sposób wspierać zmieniony sposób funkcjonowania przedsiębiorstwa, aby uzyskane dzięki temu korzyści przewyższały poniesione nakłady. Podstawową miarą jakości budowanego systemu oprogramowania będzie stopień w jakim wspiera on funkcjonowanie organizacji w jej środowisku biznesowym. Dlatego też, zanim rozpoczęmy etap analizy i formułowania wymagań dla systemu, musimy poznać elementy środowiska biznesowego przedsiębiorstwa, dla którego ten system ma być zbudowany. W szczególności, powinniśmy dobrze zrozumieć procesy w nim zachodzące.

Różnego rodzaju organizacje i przedsiębiorstwa funkcjonujące w określonych środowiskach biznesowych również stanowią fragmenty otaczającego nas świata. Podstawowe elementy opisu praktycznie każdej organizacji biznesowej to:

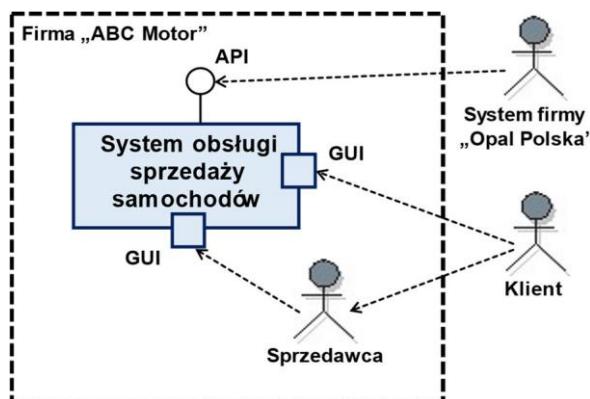
- Struktura organizacyjna – jednostki organizacyjne, pracownicy (stanowiska pracy), współpracownicy (klienci, dostawcy),
- Produkty pracy – surowce, towary, podzespoły, dokumenty, itp.,

- Systemy informatyczne – systemy biurowe, systemy zarządzania produkcją, systemy on-line, itd.

Aby zrozumieć daną organizację biznesową powinniśmy stworzyć jej model (patrz moduł II). Możemy powiedzieć, że model jest miniaturową kopią rzeczywistego biznesu. Model może dotyczyć stanu obecnego lub stanu postulowanego, jaki chcemy osiągnąć po dokonaniu zmian organizacyjnych. Najczęściej, pożądane jest, aby system informatyczny dostosować do biznesu, a nie odwrotnie – dostosować biznes do gotowego systemu obsługi przedsiębiorstwa. Oczywiście podczas szczegółowej analizy danego przedsiębiorstwa pod kątem stworzenia odpowiedniego systemu informatycznego, często dostrzega się potrzebę przemodełowania jego struktury organizacyjnej lub sposobu funkcjonowania w celu optymalizacji oraz w celu zapewnienia lepszego sprzęgu z tworzonym systemem.

Dokonując analizy przedsiębiorstwa, pierwszym pytaniem, na które powinniśmy odpowiedzieć jest: gdzie przebiega granica naszego biznesu? Innymi słowy – czym zajmuje się modelowana organizacja, a co jest poza zakresem jej działalności? Modelowanie może przebiegać na różnym poziomie szczegółowości – możemy modelować całą firmę lub tylko jakiś jej fragment, np. dział czy placówkę.

Definicję zakresu biznesu zaczynamy od definicji jego otoczenia. **Otoczenie biznesu** definiujemy jako zbiór wszystkich jednostek współpracujących z biznesem. Jeśli skupiamy się tylko na wybranym fragmencie biznesu, otoczeniem dla niego stają się też pozostałe działy przedsiębiorstwa. Tymi jednostkami współpracującymi mogą być klienci, podwykonawcy, dostawcy podzespołów, spedytorzy, zleceniodawcy, instytucje państwa, agenci, itp. Biznes może również współpracować bezpośrednio z zewnętrznymi systemami informatycznymi. Oznacza to, że dana organizacja biznesowa może posiadać wiele kanałów współpracy z jednostkami zewnętrznymi, w tym za pośrednictwem swoich systemów informatycznych. Na rysunku 7.2 widzimy odpowiedni przykład. Organizacją biznesową jest firma prowadząca sieć salonów sprzedaży samochodów. Jej otoczenie stanowią klienci oraz producenci oprogramowania, a dokładniej – systemy informatyczne tych producentów (tu np.: „System firmy Opal Polska”). Otoczenie może komunikować się z firmą na trzy sposoby. Pierwszy sposób jest sposobem tradycyjnym i polega na kontaktach międzyludzkich. Na przykład klient komunikuje się ze sprzedawcą (pracownikiem firmy). Drugi kanał jest kanałem elektronicznym, gdzie firma udostępnia (tu np. klientowi) odpowiedni interfejs użytkownika (GUI). Ostatni kanał jest również kanałem elektronicznym, w którym firma udostępnia zewnętrznym systemom odpowiedni interfejs programistyczny (API). Oznacza to zatem, że znacząca część współpracy naszej firmy z otoczeniem przebiega przy udziale systemu oprogramowania.



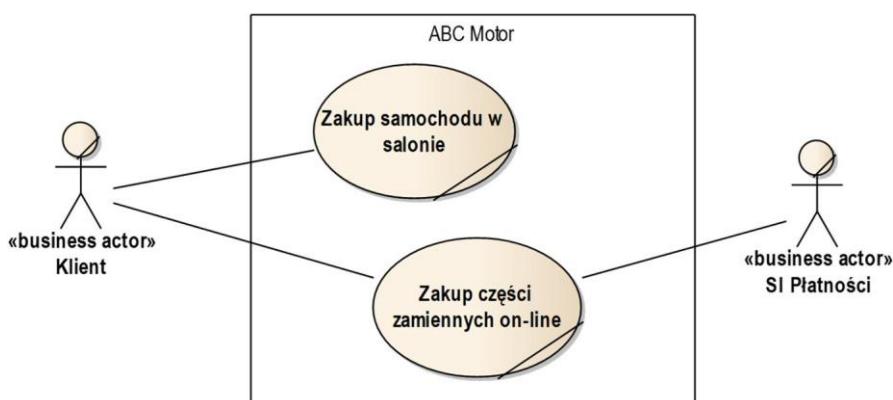
Rysunek 7.2: Określenie otoczenia biznesu oraz otoczenia systemu informatycznego

Aby opisać taką współpracę powinniśmy przede wszystkim zidentyfikować i opisać przebieg procesów biznesowych. Modelowanie tych procesów jest ważnym etapem na drodze do stworzenia

kompletnego opisu funkcjonowania przedsiębiorstwa. Żeby zrozumieć pojęcie procesu biznesowego należy zrozumieć samo pojęcie **procesu**. Jego ogólna definicja określa go jako serię wzajemnie powiązanych zadań, które przekształcają dane wejście w wyjście. Należy zwrócić uwagę na dwa osobne aspekty wynikające z tej definicji. Pierwszy związany jest z samym wyróżnianiem danego procesu oraz jego potencjalnymi interakcjami z innymi procesami. Z tego punktu widzenia proces posiada wejście, wyjście oraz określone granice wyznaczające co konkretnie on obejmuje. Jako przykład, rozważmy proces wymiany kół w samochodzie z zimowych na letnie w warsztacie samochodowym. Wejściem do tego procesu będzie samochód z zimowymi kołami i komplet kół letnich. Wyjściem będzie samochód z letnimi kołami i komplet kół zimowych. Granice procesu będące natomiast definiowały zakres czynności podejmowanych przez mechaników w warsztacie w celu wymiany kół. Drugi aspekt dotyczy wewnętrznej struktury procesu. W celu jej pokazania trzeba dokładnie określić w jaki sposób tworzące go zadania są ze sobą powiązane i w jaki sposób prowadzi to do uzyskania oczekiwanej wyjścia na podstawie wejścia. Typowo w tym celu poszczególne zadania przedstawiane są w postaci sekwencji uporządkowanej w czasie. **Proces biznesowy** jest szczególnym rodzajem procesu, który wytwarza wartość dla jego odbiorcy. Innymi słowy wartość wyjścia z procesu jest większa niż jego wejście z punktu widzenia danego biznesu. Podsumowując, można powiedzieć, że proces biznesowy:

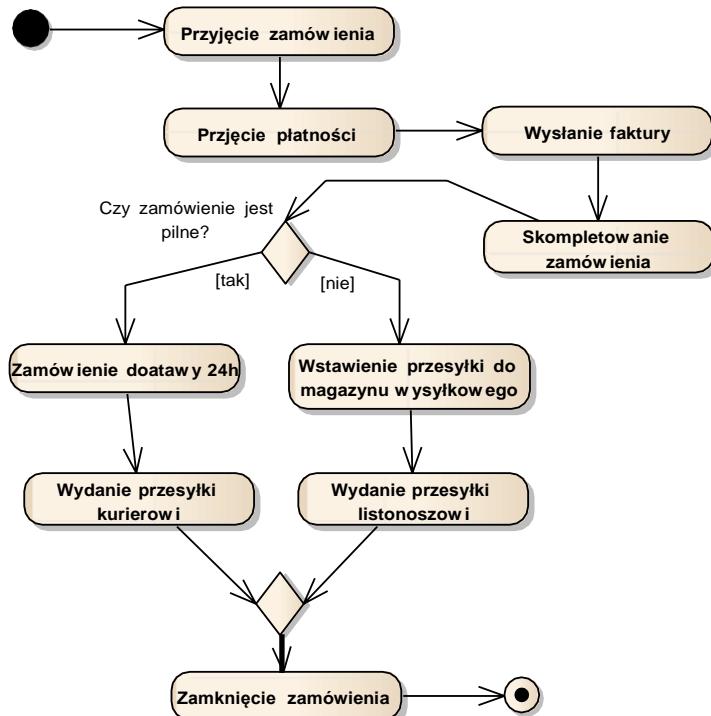
- posiada jasno określone granice, wejście i wyjście,
- składa się z sekwencji uporządkowanych na przestrzeni czasu czynności,
- tworzy wartość dodaną dla określonego odbiorcy (beneficjenta procesu).

Istnieje wiele sposobów modelowania procesów biznesowych, w tym dedykowane do tego notacje. Często stosowanym w praktyce językiem jest język BPMN (Business Process Modeling Notation). Jest to specjalizowany język służący głównie definiowaniu przebiegu procesów biznesowych. Innym często stosowanym podejściem jest zastosowanie języka uniwersalnego, jakim jest język UML (patrz moduł II, modelowanie dynamiki systemu). Model czynności zawarty w języku UML posiada elementy, które są w znacznym stopniu analogiczne do elementów języka BPMN. Zaletą takiego rozwiązania jest możliwość łatwej synchronizacji modeli procesów biznesowych z innymi modelami (modelami wymagań i modelami projektowymi). W celu określenia zakresu modelowanych procesów biznesowych możemy na przykład wykorzystać diagramy przypadków użycia. Przykładowy taki diagram widzimy na rysunku 7.3. Są tutaj zdefiniowane dwa wybrane procesy biznesowe („Zakup samochodu w salonie” i „Zakup części zamiennych on-line”), zamodelowane jako **przypadki użycia biznesu**. Dodatkowo, na rysunku zdefiniowano dwóch współpracowników biznesu w formie aktorów biznesowych. Model ten jest analogiczny do modelu przypadków użycia systemu, omówionych w module II. Różnica polega na tym, że modelowanie dokonywane jest na poziomie systemu biznesowego.



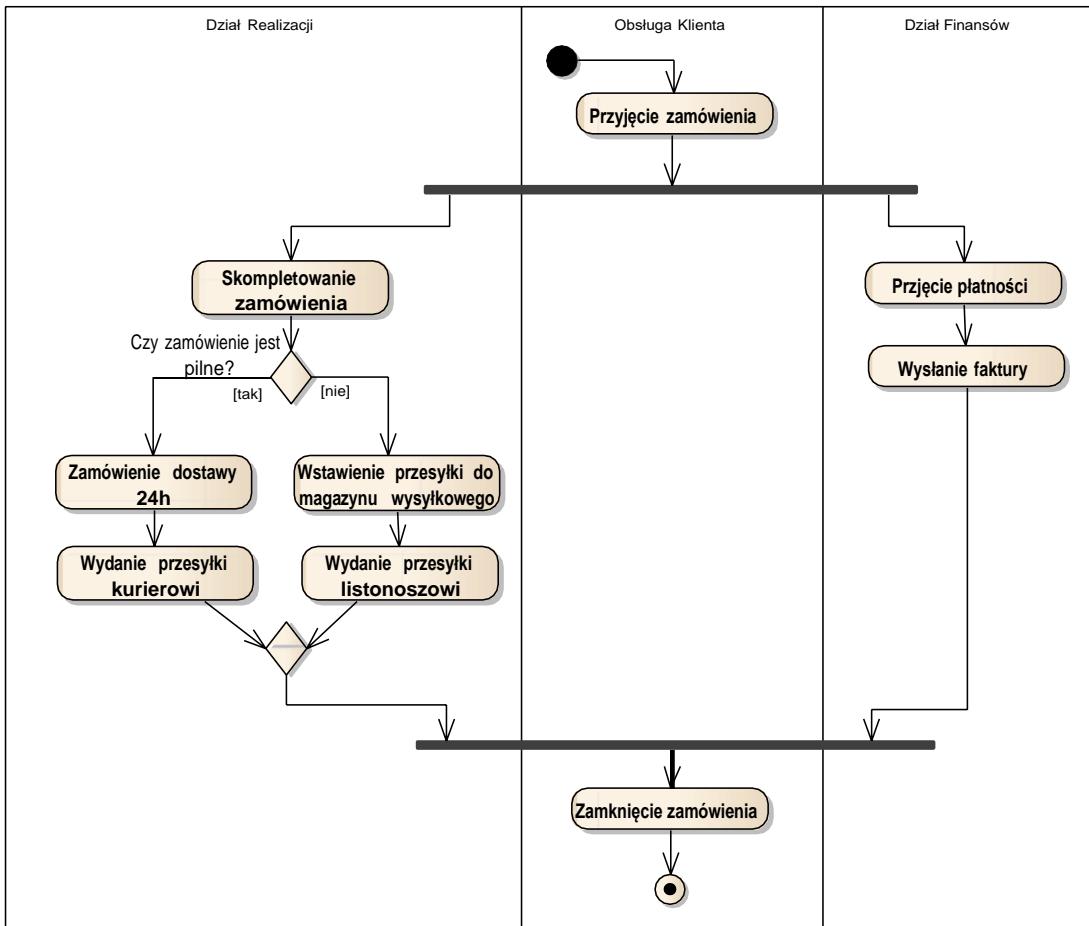
Rysunek 7.3: Procesy biznesowe modelowanie jako przypadki użycia biznesu

Przypadek użycia biznesu możemy opisać wykorzystując **diagram czynności**. Przykładowy diagram modelujący proces biznesowy „Zakup części zamiennych on-line” przedstawiony jest na rysunku 7.4. Pokazuje on przebieg obsługi zakupu z punktu widzenia modelowanej organizacji, ilustrując wszystkie konieczne do tego kroki. Jak widzimy, przebieg procesu różnicowany jest w zależności od tego czy zamówienie jest pilne czy nie. Przedstawiamy tutaj dosyć prosty proces, gdyż szczegółowe omówienie zasad modelowania biznesu leży poza zakresem niniejszych materiałów. Warto jednak zauważyć, że ciągi akcji tworzących proces biznesowy są często na tyle skomplikowane, że zazwyczaj konieczne jest stosowanie całej notacji modelu czynności, włączając w to węzły decyzyjne, węzły synchronizacji, a także przepływy obiektów.



Rysunek 7.4: Diagram czynności opisujący proces biznesowy

Diagram na rysunku 7.4 można uszczegółowić oraz zoptymalizować. Brakuje na nim m.in. informacji, kto jest odpowiedzialny za wykonanie poszczególnych akcji. Po szczegółowej analizie można również zaprojektować proces, w którym część czynności będzie wykonywana równolegle. Zmodyfikowany diagram czynności widzimy na rysunku 7.5. Wykorzystuje on konstrukcję języka UML relatywnie często stosowaną przy opisie procesów biznesowych – tzw. **tory** (ang. Swimplane). Pozwalają one pokazać podmioty odpowiedzialne za konkretne działania w ramach procesu, czy to na poziomie konkretnych osób, czy np. na poziomie działów przedsiębiorstwa. Dodatkowo, widzimy tutaj zastosowanie belek synchronizacji do pokazania ciągów akcji wykonywanych równolegle.



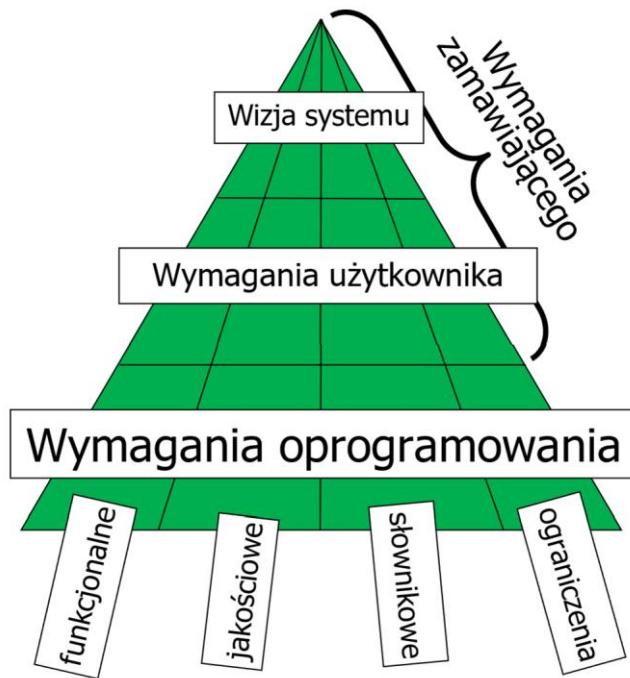
Rysunek 7.5: Przykład diagramu czynności z wykorzystaniem torów

7.3. Struktura specyfikacji wymagań – rodzaje wymagań

Jak wskazują badania sondażowe, najczęściej stosowanym narzędziem do specyfikowania wymagań jest zwykły procesor tekstu. Bardzo często, formą specyfikacji wymagań jest monolityczny dokument tekstowy, w którym nie są w jednoznaczny sposób wyodrębnione konkretne wymagania. Brakuje określenia zależności między wymaganiami, ich atrybutów czy historii dokonanych zmian. Ogólnie – brakuje systematycznej struktury zgodnej z dobrymi praktykami inżynierii wymagań. Korzystanie z takiego dokumentu podczas analizy i projektowania, a także implementacji i testów, jest niezmiernie trudne. Opisy różnych aspektów systemu są w nim przemieszane, a kolejne modyfikacje powodują dezaktualizację niejawnie związanych ze sobą części specyfikacji. Ponadto, niedostateczny formalizm zapisu wymagań sprawia, że cechy systemu są opisane w sposób niejasny, niejednoznaczny i pozbawiony precyzji.

Dla powodzenia projektu konstrukcji oprogramowania kluczowe jest zatem przedstawienie zgromadzonych wymagań w systematyczny sposób, zapewniający spełnienie omówionych wyżej kryteriów jakościowych. Aby opis przyszłego systemu był użyteczny dla różnych grup uczestniczących w projekcie, niezbędne jest systematyczne podejście do procesów związanych z tworzeniem specyfikacji wymagań. Procesy te obejmują zbieranie, organizację, dokumentowanie, modyfikację, śledzenie zmian oraz wykorzystanie wymagań. Efektywność tych działań uwarunkowana jest zapewnieniem właściwej formy i struktury specyfikacji wymagań. Dobre praktyki inżynierii wymagań wskazują na potrzebę jednoznacznego podziału specyfikacji zarówno pod kątem poziomu abstrakcji (szczegółowości) wymagań jak i ich

rodzaju. Ponadto, bardzo istotne jest precyzyjne określenie powiązań między wymaganiami różnego poziomu i rodzaju. Dobra zbudowana specyfikacja wymagań powinna mieć strukturę **piramidy**, przedstawionej na rysunku 7.6. Szczyt takiej piramidy stanowi ogólny opis systemu – jego wizja. Elementy wizji systemu stanowią punkt wyjścia do sformułowania wymagań użytkownika określających zakres budowanego systemu. Podstawą piramidy i jednocześnie jej największym fragmentem są wymagania oprogramowania, wyrażające wszystkie szczegółowe potrzeby zamawiającego. Na wszystkich poziomach piramidy formułujemy wymagania czterech rodzajów: wymagania funkcjonalne, jakościowe, słownikowe, oraz ograniczenia środowiskowe i techniczne. Poniżej omówimy wszystkie te elementy piramidy wymagań.



Rysunek 7.6: Organizacja specyfikacji wymagań

Należy tutaj podkreślić, że specyfikacja wymagań nie musi (a często – nie powinna) być tworzona jako jeden gruby dokument na początku całego procesu wytwarzania oprogramowania. W metodach迭代 (iteracyjnych) (zarówno zwinnych jak i sformalizowanych), wymagania są zbierane i formułowane przez cały okres projektu. Wymagania mogą podlegać zmianom, zatem specyfikacja wymagań powinna być produktem „żyjącym” przez cały czas projektu. W wielu projektach konieczne jest jednak sformułowanie formalnej umowy między zamawiającym a wykonawcą systemu. Kluczowym elementem takiej umowy jest **specyfikacja wymagań zamawiającego** (np. w zamówieniach publicznych spełniająca rolę specyfikacji istotnych warunków zamówienia). Z punktu widzenia struktury wymagań, taka specyfikacja powinna objąć dwie górne warstwy piramidy (patrz rysunek 7.6).

- wizja systemu pełni rolę określenia ogólnych cech systemu w ścisłym powiązaniu z potrzebami biznesowymi zamawiającego,
- wymagania użytkownika pełnią rolę specyfikacji potrzebnej do określenia rozmiaru systemu i nakładu pracy potrzebnego na jego zbudowanie.

W ramach wymagań zamawiającego, zamawiający przedstawia swoje rzeczywiste potrzeby związane z wspieraną przez system dziedziną swojej działalności. Jednocześnie, określa zakres systemu na tyle jednoznacznie, żeby możliwe było dokonanie wyceny budowanego systemu przez wykonawcę.

Zwróćmy uwagę na to, że do określenia zakresu systemu nie są konieczne wszystkie szczegółowe wymagania (np. kolor ekranów czy kolejność nawigacji między oknami). Dlatego też, umieszczenie szczegółowych wymagań oprogramowania w specyfikacji wymagań zamawiającego należy traktować jako błąd. Takie wymagania są często zmieniane wielokrotnie w trakcie projektu i nakład pracy związany z ich stworzeniem na samym początku projektu będzie zazwyczaj stracony. W dalszej części omawiamy poszczególne składniki piramidy wymagań. Więcej szczegółów zostanie przedstawionych w kolejnym rozdziale niniejszego modułu.

Wizja systemu ma charakter bardzo ogólny i nie powinna mieć charakteru wiążącego w kontaktach między zamawiającym a wykonawcą systemu. Celem jej jest sformułowanie intencji przyświecających rozwojowi systemu i wyszczególnienie problemów odkrytych w modelu biznesu zamawiającego. Kompletna wizja systemu powinna zawierać:

- opis problemu biznesowego, który jest podstawą dla powstania nowego systemu; oznacza to m.in. identyfikację głównych obszarów biznesu zamawiającego, które powinny uzyskać wsparcie w wyniku zbudowania nowego systemu,
- przedstawienie interesariuszy powstającego systemu, czyli osób bezpośrednio i pośrednio zaинтересowanych powstaniem systemu; są to m.in.: kluczowi użytkownicy, ich reprezentanci oraz kadra kierownicza, zamawiający system, osoby finansujące powstanie systemu lub w inny sposób związane z nim w aspekcie ekonomicznym, personel zaangażowany w tworzenie systemu,
- ogólne cechy i funkcje systemu oraz słownik pojęć biznesowych,
- najważniejsze ograniczenia środowiskowe i techniczne przyszłego systemu,
- opcjonalnie - motto projektu, czyli krótki „slogan” oddający główny cel powstania produktu informatycznego.

Wymagania użytkownika definiują system w sposób na tyle szczegółowy, aby potencjalnie stanowić mniej lub bardziej formalną podstawę do kontroli postępów prac, w tym – zawarcia kontraktu na wykonanie systemu. Typowo, na wymagania użytkownika składają się na nie wymagania funkcjonalne wyrażone przez uporządkowaną strukturę przypadków użycia, definicje aktorów, słownik pojęć wykorzystanych podczas formułowania wymagań, istotne wymagania jakościowe oraz ograniczenia środowiskowe przyszłego systemu.

Na poziomie wymagań użytkownika konieczne jest wyraźne wydzielenie struktury i podział na pakiety wymagań. Konieczne jest wyraźne rozróżnienie między wymaganiami funkcjonalnymi i jakościowymi oraz ich pakietyzacja. Wymagania funkcjonalne określamy stosując notacje charakterystyczne dla wybranych metodyk twórczych. Najczęściej stosowanymi notacjami są przypadki użycia systemu (język UML) oraz historie użytkownika. Z uwagi na liczbę wymagań funkcjonalnych konieczne jest zazwyczaj podzielenie ich na pakiety, np. ze względu na użytkowników lub obszarów funkcjonalnych. Dla wymagań jakościowych musimy na tym poziomie wymagań sformułować precyzyjne metryki. Umożliwi to – po zawarciu kontraktu – odpowiednią kontrolę spełnienia tych wymagań.

W trakcie prac na systemem, wymagania oprogramowania podlegają dokładnej specyfikacji – uszczegółowieniu. Odpowiada za to najbardziej szczegółowy poziom piramidy wymagań, czyli **wymagania oprogramowania**. Precyza określenia wymagań oprogramowania powinna zapewniać jednoznaczność implementacji systemu przez zespół deweloperski. Na ich podstawie, zespół powinien móc zaprojektować i zakodować wszystkie komponenty systemu. Oznacza to, że w ramach specyfikowania wymagań oprogramowania konieczne jest szczegółowe ustalenie z zamawiającym scenariuszy zachowania systemu, wyglądu interfejsu użytkownika oraz struktury przetwarzanych przez system danych wraz z opisem algorytmów przetwarzania danych. Często konieczne jest również doprecyzowanie cech

jakościowych systemu, np. poprzez dokładne określenie czasów odpowiedzi dla poszczególnych interakcji użytkownika z systemem.

Wymagania oprogramowania bazują na wymaganiach „odkrytych” i zapisanych jako wymagania zamykającego. Szczegółowa struktura tych wymagań zależy od rodzaju wymagania bazowego. Na przykład, w przypadku wymagań funkcjonalnych, najczęstszą notacją są szczegółowe scenariusze interakcji między jednostkami na zewnątrz systemu a systemem. W ten sposób powinny być systematycznie opisane wszystkie realizowane przypadki użycia lub historie użytkownika. Podobnie, słownik dziedziny sformułowany na poziomie wymagań użytkownika powinien ulec przekształceniu w model danych, który bardzo szczegółowo określa klasy danych, atrybuty i ich typy, a także rodzaje relacji.

Na wszystkich poziomach piramidy wymagań powinien być wyraźnie zachowany podział między różnymi rodzajami wymagań (patrz rysunek 7.6). Rodzaje wymagań stanowią „kolumny” piramidy, gdyż na kolejnych poziomach stanowią jednolitą całość, opisywaną w sposób coraz bardziej szczegółowy. Najczęściej spotykanym podziałem wymagań jest podział na wymagania funkcjonalne i jakościowe. Czasami jako rodzaj wymagań jakościowych wymienia się ograniczenia środowiskowe i techniczne, ale tutaj traktujemy je jako osobny rodzaj wymagań. Bardzo istotny jest również słownik (wymagania słownikowe), który spaja całość specyfikacji poprzez zastosowanie wspólnej terminologii.

Wymagania funkcjonalne określają sposób zachowania się systemu w interakcji z jednostkami na zewnątrz tego systemu – użytkownikami (ludźmi) oraz systemami zewnętrznymi. W ramach specyfikacji wymagań funkcjonalnych powinniśmy odpowiedzieć na podstawowe pytanie: „co system ma robić?” W szczególności, specyfikacja powinna określać m.in. usługi dostarczane przez system, sposób reakcji na komunikaty dochodzące spoza systemu, sposób zachowania w określonych sytuacjach i przy aktualnym stanie systemu. Zestaw wymagań funkcjonalnych określa zakres funkcjonalności konieczny do zrealizowania przez zespół budujący system oprogramowania. Dobrą praktyką jest również określenie funkcjonalności, które nie będą realizowane lub zostaną zrealizowane w kolejnych projektach.

Podstawowym sposobem wyrażania wymagań funkcjonalnych jest podział całej funkcjonalności systemu na jednostki o dobrze określonym celu dla użytkowników systemu. Takimi jednostkami mogą być przypadki użycia lub historie użytkownika, formułowane zazwyczaj na poziomie wymagań użytkownika. Na poziomie wizji systemu w zupełności wystarczające jest stosowanie prostych zdań oznajmujących bez zachowania konkretnej notacji. Na poziomie wymagań oprogramowania, szczegóły interakcji systemu możemy opisać scenariuszami oraz scenopisami. Wszystkie te notacje omówione są w kolejnym rozdziale.

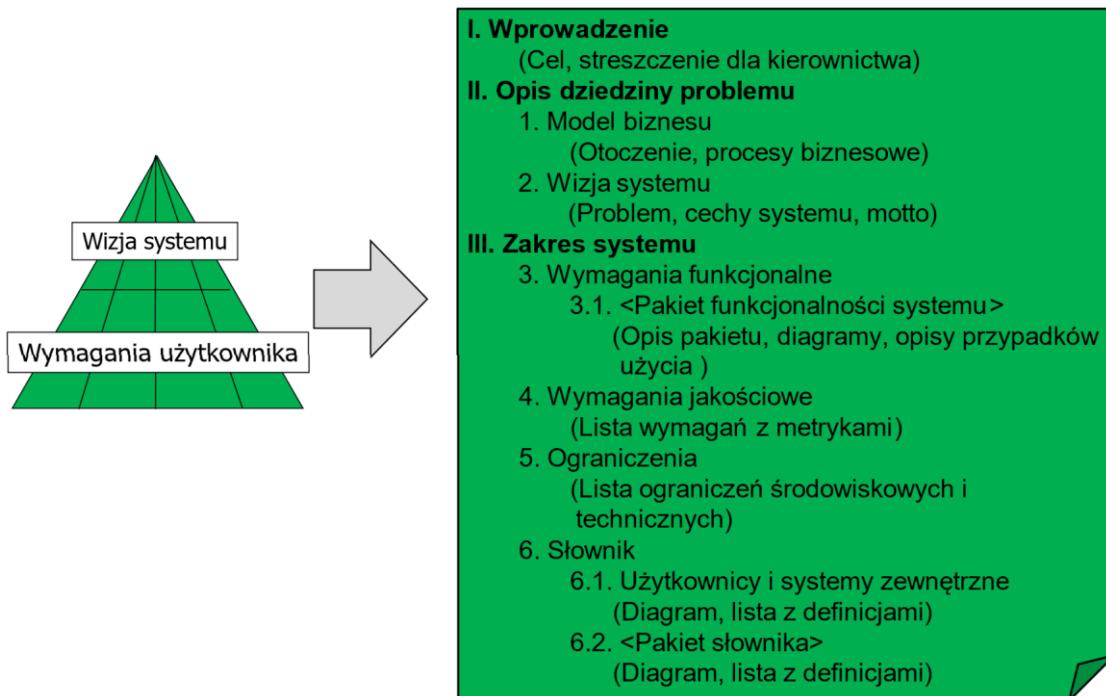
Wymagania jakościowe inaczej nazywamy wymaganiami pozafunkcjonalnymi. Stanowią one charakterystyki systemu określające sposób oceny działania systemu. O ile wymagania funkcjonalne definiują to, „co system powinien robić”, o tyle wymagania jakościowe definiują to, „jak system powinien coś robić”. System może realizować wszystkie wymagania funkcjonalne, lecz nadal nie spełniać oczekiwów zamawiającego. Może np. działać zbyt wolno, być zawodny lub nie spełniać kryteriów bezpieczeństwa. Wymagania jakościowe dzielimy na różne kategorie, określające różne charakterystyki jakościowe budowanego systemu.

Bardzo istotne dla określenia wymagań jakościowych jest przedstawienie sposobu ich weryfikacji (testowania). Musimy zapewnić mierzalność wymagań jakościowych, czyli zdefiniować sposób oceny spełnienia wymagań zgodnie z możliwie najbardziej obiektywnymi kryteriami. W tym celu konieczne jest zdefiniowanie metryk wymagań, które opiszą sposób pomiaru oraz oczekiwane wyniki. Warto też zwrócić uwagę na to, że wymagania jakościowe „przecinają” wymagania funkcjonalne. Jedno wymaganie jakościowe może mieć wpływ na sposób (i jednocześnie – koszt) realizacji wielu wymagań funkcjonalnych.

Ograniczenia środowiskowe i techniczne określają warunki narzucone budowanemu systemowi przez jego otoczenie oraz konieczne do zastosowania technologie. Ograniczenia nie są zatem wymaganiami w sensie potrzeb zamawiającego. Stanowią one wymagania w sensie narzuconych przez zamawiającego warunków, w jakich będzie tworzony i uruchamiany system. Te warunki mogą dotyczyć środowiska, w jakim będzie pracował system (np. wyposażenie stanowisk pracy, warunki pogodowe), technologii wymaganych podczas budowy systemu (np. posiadane licencje na system bazy danych) lub jego działania (np. posiadany przez zamawiającego sprzęt).

Wymagania słownikowe definiują zakres pojęć i danych, które są używane w ramach dziedziny problemu obejmującej budowany system. Słownik powinien zawierać na bieżąco aktualizowane definicje wszystkich pojęć używanych w ramach wymagań funkcjonalnych, jakościowych oraz ograniczeń. Definicje te wykonywane są na różnym poziomie szczegółowości. Na poziomie wizji, często wystarczy krótkie 1-2 zdaniowe wyjaśnienie znaczenia danego pojęcia. Na poziomie wymagań użytkownika konieczne jest określenie co najmniej najważniejszych atrybutów pojęć oraz wyspecyfikowanie relacji między pojęciami. Wymagania oprogramowania wprowadzają konieczność zdefiniowania wszystkich szczegółów dotyczących struktur danych, czyli stworzenie szczegółowego modelu danych. Do formułowania wymagań słownikowych możemy w najprostszym przypadku użyć formuły słownika językowego – listy pojęć z ich definicjami. Często jednak o wiele korzystniejsze jest zastosowanie notacji graficznej, np. diagramów klas języka UML. Takie notacje zapewniają większą precyzję i lepsze zrozumienie przedstawianej dziedziny problemu.

Zbiór wszystkie wymagań stanowi specyfikację wymagań, która może być wyrażona w sposób mniej lub bardziej sformalizowany. Często wymagane jest stworzenie formalnego dokumentu, który będzie stanowił załącznik do umowy miedzy zamawiającym z wykonawcą. Przykładową strukturę takiego dokumentu **specyfikacji wymagań zamawiającego** przedstawia rysunek 7.7. Struktura ta jest zgodna ze strukturą dwóch górnych warstw piramidy. Jeśli jest to wymagane, dokument rozpoczyna się od wprowadzenie, które pokrótce opisuje motywację powstania, cel oraz strukturę. Kolejna część zawiera opis dziedziny problemu, na który składają się model biznesu oraz wizja systemu. Ostatnią częścią jest zakres systemu, który dzielimy w sposób wyraźny na cztery omawiane wyżej rodzaje wymagań. Biorąc pod uwagę liczbę wymagań, może być konieczny ich podział na pakiety. Dotyczy to szczególnie wymagań funkcjonalnych i słownikowych.



Rysunek 7.7: Przykładowa struktura dokumentu specyfikacji wymagań zamawiającego

Warunki projektu mogą również narzucać tworzenie dokumentów wymagań już po zawarciu kontraktu. Struktura takich dokumentów może być rozszerzeniem struktury specyfikacji wymagań zamawiającego, przedstawionej na rysunku 7.7. Jak wiemy, wymagania zamawiającego są uszczegóławiane w ramach specyfikowania wymagań oprogramowania. Dlatego też, dokument **specyfikacji wymagań oprogramowania** może zawierać uszczegółowienie elementów opisujących zakres systemu. W rezultacie, rozdział 3 naszej przykładowej struktury specyfikacji wymagań będzie rozszerzony o szczegółowe scenariusze przypadków użycia (lub historii użytkownika). W rozdziale 6 możemy dodać nowy pakiet, przedstawiający słownik elementów interfejsu użytkownika. Rozdziały 4 i 5 mogą być uzupełnione o kolejne wymagania jakościowe i ograniczenia, które np. wynikają z analizy scenariuszy i scenariuszy.

Na koniec warto jeszcze raz podkreślić, że sposób tworzenia specyfikacji wymagań silnie zależy od warunków w projekcie, a szczególnie – metodyki twórczej. Różne organizacje posiadają różne wzorce specyfikacji wymagań. Specyfikacje mogą być tworzone w oparciu o różnego rodzaju narzędzia dedykowane do zarządzania wymaganiami, czy też narzędzia modelowania (np. dla języka UML). W projektach prowadzonych metodą zwinnymi zazwyczaj nie tworzy się dokumentów w formie przedstawionej powyżej. Należy jednak pamiętać, że niezależnie od sposobu prowadzenia projektu, pominięcie niektórych wymagań może prowadzić do istotnych problemów lub wręcz niepowodzenia projektu. Dlatego też warto jest traktować wzorce specyfikacji wymagań jako podstawę zapewnienia ich kompletności i spójności.

Zadania

Zadanie 1

Mamy zadany proces biznesowy w ramach określonej dziedziny problemu (patrz rozdział 1, sekcja 1.5). Proszę przedstawić w dowolnej formie sekwencję czynności wykonywaną w ramach tego procesu. Proszę uwzględnić alternatywne przebiegi procesu.

Zadanie 2

Mamy zadaną dziedzinę problemu (patrz rozdział 1, sekcja 1.5). Proszę napisać kilka wymagań funkcjonalnych dla systemu budowanego w ramach zadanej dziedziny. Wymagania proszę zapisać w sposób ogólny – na poziomie wizji systemu.

Zadanie 3

Mamy zadaną dziedzinę problemu (patrz rozdział 1, sekcja 1.5). Proszę napisać kilka wymagań jakościowych dla systemu budowanego w ramach zadanej dziedziny. Wymagania proszę zapisać w sposób ogólny – na poziomie wizji systemu. W miarę możliwości proszę określić liczbowo parametry jakościowe dla wymagań.

Słownik pojęć

Otoczenie biznesu (środowiska)

Zbiór wszystkich jednostek współpracujących z biznesem (środowiskiem).

Proces

Seria wzajemnie powiązanych zadań, które przekształcają dane wejście w wyjście.

Proces biznesowy

Szczególny rodzaj procesu, który wytwarza wartość dla jego odbiorcy. Proces biznesowy posiada jasno określone granice, wejście i wyjście, składa się z sekwencji uporządkowanych na przestrzeni czasu czynności oraz tworzy wartość dodaną dla określonego odbiorcy (beneficjenta procesu).

Wymaganie

Właściwość produktu końcowego (systemu oprogramowania), którą musi on posiadać, aby spełnić oczekiwania zamawiającego. Właściwością tą może być określony sposób funkcjonowania systemu lub cecha jakościowa narzucona na system. Mówiąc się, że system spełnia wymagania, jeśli zostało potwierdzone, że posiada wszystkie właściwości określone tymi wymaganiami.

Co trzeba zapamiętać

Wymagania w inżynierii oprogramowania

Wymagania odpowiadają na najistotniejsze w całym procesie budowy oprogramowania pytanie, „jaki system mamy zbudować?” W dobrze zorganizowanym procesie wytwarzającym, wymagania sterują konstrukcją systemu. Jakość wymagań jest kluczowym elementem powodzenie każdego projektu konstrukcji systemu oprogramowania. Wymagania powinny być kompletne, jednoznaczne, poprawne, spójne (niesprzeczne), możliwe do zarządzania oraz testowalne. Aby sformułować dobrej jakości wymagania musimy uzyskać dostęp do odpowiednich źródeł informacji. Można tutaj wyróżnić źródła osobowe (ludzi) oraz źródła nieosobowe (np. dokumenty). Pisząc wymagania należy pamiętać o tym, że będą z nich korzystały bardzo różnorodne grupy osób. Możemy tu wyróżnić osoby po stronie zamawiającego (np. firma zamawiająca), jak i po stronie wykonawcy (np. firma wykonująca system).

Specyfikowanie środowiska (biznesu)

Wdrożenie w przedsiębiorstwie nowego lub rozbudowanie dotychczasowego systemu oprogramowania często powoduje istotne zmiany w sposobie jego organizacji oraz sposobie realizacji procesów biznesowych. Dlatego też, zanim rozpoczęniemy etap analizy i formułowania wymagań dla systemu, musimy poznać elementy środowiska biznesowego przedsiębiorstwa, dla którego ten system ma być zbudowany. W szczególności, powinniśmy dobrze zrozumieć procesy w nim zachodzące, a w tym – jego strukturę organizacyjną, produkty pracy i systemy informatyczne. Procesy biznesowe modelujemy np. w językach BPMN lub UML. Proces biznesowy w języku UML możemy zamodelować jako przypadek użycia biznesu opisany modelem czynności.

Struktura specyfikacji wymagań

Dla powodzenia projektu konstrukcji oprogramowania kluczowe jest przedstawienie zgromadzonych wymagań w systematyczny sposób, zapewniający spełnienie kryteriów jakościowych. Dobrze zbudowana specyfikacja wymagań ma strukturę piramidy z trzema poziomami – wizję systemu, wymaganiami użytkownika oraz wymaganiami oprogramowania. Wyróżniamy cztery podstawowe typy wymagań (kolumny piramidy) – wymagania funkcjonalne, wymagania jakościowe, wymagania słownikowe oraz ograniczenia. Specyfikacje wymagań mogą być sformułowane na różnym poziomie formalności. Często wymagane jest utworzenie formalnego dokumentu – specyfikacji wymagań zamawiającego, który będzie składnikiem umowy między zamawiającym a wykonawcą. Struktura takiego dokumentu zależy od przyjętych konwencji, ale powinna obejmować wszystkie typy wymagań. Na poziomie specyfikacji wymagań oprogramowania, wymagania zamawiającego są dodatkowo uszczegóławiane.

Rozwiązania zadań

Rozwiązanie zadania 1

Dziedzina problemu – Wydział Ruchu Drogowego policji.

Proces biznesowy: Nałożenie mandatu w trybie zaocznym

1. Fotoradar rejestruje wykroczenie i przekazuje zdjęcie do systemu
2. System automatycznie identyfikuje pojazd i właściciela pojazdu na podstawie zdjęcia [zidentyfikowano]
3. System sprawdza, czy jest ustawiona komunikacja elektroniczna z właścicielem pojazdu [jest ustawiona]

4. System wysyła powiadomienie o wykroczeniu
5. Właściciel pojazdu przyjmuje mandat jako kierowca

→ Koniec

- 1.-4. jak wyżej
- 5'. Właściciel pojazdu wysyła informację o kierowcy pojazdu
- 6'. System wysyła proponowany mandat do kierowcy pojazdu
- 7'. Kierowca w systemie sprawdza szczegóły mandatu
- 8'. Kierowca przyjmuje mandat

→ Koniec

1.-4., 5'.-7' jak wyżej

8''. Kierowca nie przyjmuje mandatu

→ Koniec

Rozwiązanie zadania 2

Dziedzina problemu – system dla Wydziału Ruchu Drogowego policji.

FK1: System musi umożliwiać opłacanie mandatów

FK2: System powinien umożliwiać zalogowanie się poprzez profil zaufany

FK6: System powinien pokazywać aktualny stan konta punktów karnych

FK15: System powinien umożliwiać obsługę odwołań od mandatów (przez urzędników)

FK16: System musi automatycznie ustalać numer rejestracyjny na podstawie zdjęcia pojazdu

FK22: System musi automatycznie blokować prawo jazdy w systemie CEPiK

Rozwiązanie zadania 3

Dziedzina problemu – system dla Wydziału Ruchu Drogowego policji.

JK1: System musi być dostępny w trybie 24/7

JK3: Dopuszcza się maksymalnie 1 awarię klasy 1 na rok

JK4: Awaria klasy 1 nie może trwać dłużej niż 30 minut

JK7: System powinien mieć interfejs użytkownika, który użytkownicy ocenią jako przejrzysty i łatwy w obsłudze

JK8: System powinien umożliwiać zobaczenie mandatu max. 1 min po jego wystawieniu

JK9: System powinien być dostępny na urządzeniach mobilnych

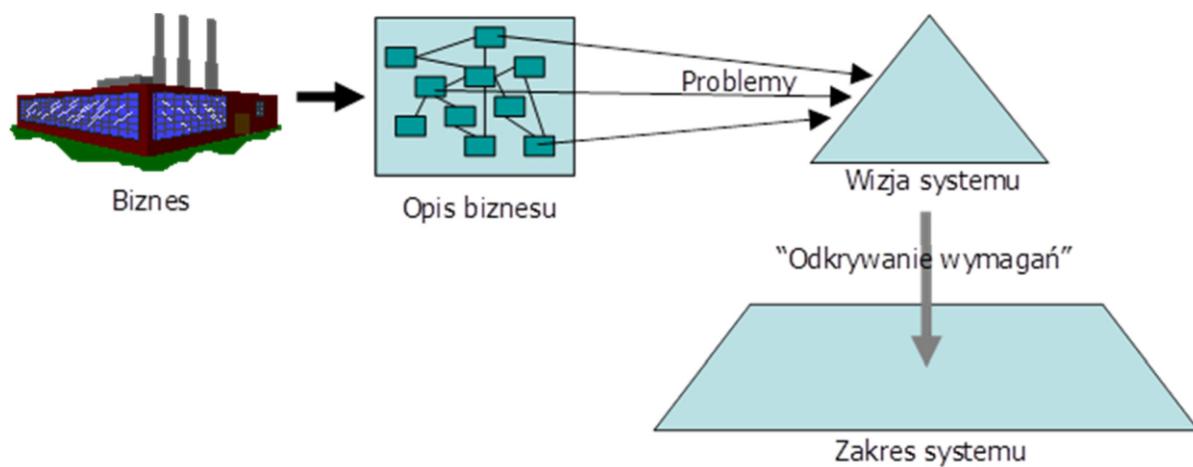
8. Podstawy specyfikowania wymagań

8.1. Specyfikowanie wizji systemu

Wizja systemu oprogramowania jest opisem potrzeb zamawiającego, dotyczącego systemu oprogramowania, przedstawiona na stosunkowo wysokim poziomie ogólności. Specyfikacja na poziomie wizji zbiera w jedną całość opis najważniejszych problemów będących rezultatem analizy biznesu (np. problemy dotyczące klientów firmy, określone przez dział marketingu) oraz wynikających z nich cech produktu software'owego. Dokument wizji może służyć za podstawę do dyskusji i podjęcia kluczowych decyzji odnośnie budowy systemu. Może też służyć do ujednolicenia rozumienia cech systemu, poprzez uzyskanie porozumienia między wszystkimi uczestnikami przedsięwzięcia, przede wszystkim między zamawiającym system (sponsorem) a wykonawcą. Wizja systemu powinna dawać przede wszystkim odpowiedzi na następujące pytania:

- Po co budujemy nowy system?
- Jakie problemy naszego biznesu ten system rozwiąże?
- Dla kogo jest ten system?
- Jakie cechy ma mieć ten system, aby pokonać zidentyfikowane problemy?

Odpowiedzi na powyższe pytania szukamy przede wszystkim poprzez analizę modelu biznesu, co ilustruje rysunek 8.1. Dokument wizji będzie efektywny, jeśli scharakteryzuje system z różnych perspektyw, w skrócone, czytelnej i zrozumiałej dla wszystkich uczestników przedsięwzięcia formie. Wizja systemu jest przedmiotem zainteresowania we wczesnych etapach realizacji przedsięwzięcia. Jej przygotowanie, choć nie jest wiążące dla wykonawcy systemu, przynosi wymierne korzyści w kolejnych etapach. Na jej podstawie odkrywane są bardziej szczegółowe wymagania dotyczące zakresu, pozwalające dokładnie wyznaczyć zakres przyszłego systemu (patrz ponownie rysunek 8.1).



Rysunek 8.1: Wizja systemu jako istotny element podejmowania decyzji

Wizja systemu, mimo swej ogólności, powinna posiadać konkretną strukturę. Struktura ta powinna uwzględniać możliwość podejmowania racjonalnych decyzji w początkowych fazach całego procesu budowy systemu oprogramowania. Typowo, wizja zawiera stwierdzenie problemu, opis interesariuszy (grup zainteresowanych), podstawowe cechy systemu i ograniczenia, a także – niekiedy – motto definiujące zasadniczy cel budowy systemu.

Pierwszym krokiem podczas analizy zasadności budowy systemu oprogramowani powinno być zdefiniowanie problemów, które ewentualny system mógłby rozwiązać. Pozwala to na wypracowane wspólnego w ramach organizacji stanowiska odnośnie obszarów, które wymagają wsparcia przy pomocy nowego oprogramowania. Opis problemu możemy sformułować w postaci **stwierdzenia problemu** (ang. problem statement). Powinno ono odpowiadać na pytanie „dlaczego potrzebujemy nowego oprogramowania?”. Stwierdzenie problemu ma postać zwięzkiego i jednoznacznego opisu, który identyfikuje zagadnienie biznesowe stawiane przed budowanym systemem a także wskazuje ogólną metodę jego rozwiązania oraz potencjalne korzyści płynące z wprowadzenia takiego rozwiązania.

Dla wyrażenia stwierdzenia problemu możemy na przykład skorzystać ze wzorca. Poniższy wzorzec składa się z 5 części, które najpierw określają problem i jego implikacje, a następnie przedstawiają możliwe rozwiązanie problemu i korzyści z niego wynikające.

- Problem polega na ... [tu opis problemu wynikający z modelu biznesu]
- Problem dotyczy ... [tu lista interesariuszy, których problemem dotyczy]
- Rezultatem problemu jest ... [tu opis implikacji występowania problemu]
- Problem można rozwiązać budując system ... [tu opisowa nazwa systemu]
- Korzyści z rozwiązania to ... [tu lista korzyści wynikających ze zbudowania systemu]

Powyższy wzorzec może zachęcać do dosyć rozbudowanego opisu. Aby tego uniknąć, możemy go przekształcić w jedno złożone zdanie:

- Problem polegający na, dotyczący, którego rezultatem jest, można rozwiązać budując system, co spowoduje (korzyści)

Innym wariantem jest wykorzystanie dwóch zdań, z których pierwsze identyfikuje problem, a drugie opisuje jego rozwiązanie. Poniższy przykład przedstawia tego typu stwierdzenie problemu dla hipotetycznego systemu obsługi salonów samochodów.

- „*Problem polega na dużym obciążeniu sprzedawców czynnościami rejestracji sprzedaży samochodów, co dotyczy sprzedawców i klientów, a czego rezultatem jest przemęczenie i mała wydajność sprzedawców oraz niezadowolenie klientów z obsługi.*”
- „*Rozwiązaniem problemu jest budowa nowego systemu zarządzania sprzedażą samochodów, co spowoduje zmniejszenie obciążenia sprzedawców formalnościami i większe zaangażowanie w kontakty z klientem.*”

Opracowanie stwierdzenia problemu w podobnej formie, to bardzo prosty i skuteczny sposób na zrozumienie poglądu interesariuszy projektu na występujące w organizacji problemy oraz sposoby ich rozwiązania. Niestety, bardzo często pomija się sformułowanie problemu. Oznacza to, że kluczowe decyzje dotyczące potrzeby zamówienia systemu mogą być podejmowane bez rozpoznania rzeczywistych problemów, a jedynie np. na podstawie mody („inni już mają taki system”) lub intuicji („na pewno nam się to przyda”).

Zwróćmy uwagę, że istotnym elementem stwierdzenia problemu jest identyfikacja grup osób zainteresowanych systemem. Dlatego też, kolejnym elementem wizji systemu powinna być **specyfikacja interesariuszy**. Interesariuszami są wszystkie te osoby, na których wdrożenie nowego systemu będzie miało bezpośredni lub pośredni wpływ. Wpływ ten może być różnego rodzaju i może dotyczyć np. kwestii finansowych (zyski, wynagrodzenia) lub bytowych (czas pracy, jakość środowiska pracy). Możemy wyróżnić kilka typowych grup interesariuszy.

- Użytkownicy systemu – osoby, które będą się posługiwać systemem w różnym zakresie (na co dzień lub sporadycznie) i mający różne oczekiwania odnośnie cech systemu widocznych przez jego interfejs użytkownika,
- Klienci organizacji biznesowej – osoby, które współpracują z organizacją rozważającą budowę nowego systemu, których w jakiś sposób dotknie wdrożenie tego systemu – bezpośrednio (jako użytkowników) lub pośrednio (w kontaktach z pracownikami używającymi nowy system),
- Inwestorzy (np. akcjonariusze) – osoby, których zwrot z inwestycji zależy od efektywności danej organizacji biznesowej, która z kolei może zależeć od jakości nowego systemu oprogramowania,
- Zamawiający – osoby, które podejmują kluczowe decyzje dotyczące danej organizacji biznesowej (wyższa kadra zarządzająca), w tym – decyzje odnośnie zamawiania i wdrażania nowego oprogramowania,
- Dział IT – osoby, które będą utrzymywać nowy system (administrowały systemem).

Zwrócmy uwagę na to, że interesariusze mogą mieć bardzo zróżnicowane oczekiwania odnośnie nowego systemu i posiadać różny punkt widzenia na występujące w organizacji problemy. Najłatwiej jest skupić się na identyfikacji bezpośrednich użytkowników systemu oraz ich potrzeb. Nieco trudniejsza jest identyfikacja interesariuszy, którzy będą mieli jedynie pośredni kontakt z systemem. Tacy interesariusze często nie są bezpośredniimi użytkownikami, a jedynie system w pewnym stopniu wpływa na ich działalność. Nie należy jednak zaniedbywać wykonania specyfikacji interesariuszy. Mają oni często bardzo istotny wpływ na cały proces wytwarzania oprogramowania i identyfikacja kryteriów ich satysfakcji może być kluczowa dla jego powodzenia.

Podobnie jak dla stwierdzenia problemu, dla opisu grup interesariuszy możemy wykorzystać wzorzec. W poniższym przykładzie wzorca nacisk położony jest na dogłębne zrozumienie zakresu odpowiedzialności interesariuszy oraz kryteriów ich zadowolenia.

- Nazwa interesariusza i typ (np. „Sprzedawca – użytkownik”)
- Opis interesariusza – krótka charakterystyka danej grupy osób wraz z ew. podaniem kluczowych reprezentantów (np. „Pracownicy salonów zajmujący się sprzedażą samochodów. Doświadczeni reprezentanci: Jan Iksiński i Adam Ygrekowski”)
- Zakres obowiązków – czym zajmuje się interesariusz w ramach zadanej dziedziny problemu (np. „Oferowanie samochodów klientom, organizowanie jazd testowych oraz obsługa całego procesu sprzedawczego”)
- Kryteria zadowolenia – jakie czynniki mogą wpływać na zaakceptowanie nowego systemu oprogramowania (np. „Zmniejszenie obciążenia biurokratycznego w procesie sprzedawczym, odciążenie od ręcznego przygotowywania raportów, wydłużenie czasu dostępnego na czynności marketingowe”)
- Zadania w projekcie – jak dany interesariusz będzie uczestniczył w procesie zamawiania i budowy oprogramowania (np. „Wydelegowanie dwóch sprzedawców jako ekspertów sprzedaży, do określania zakresu systemu, szczegółowych wymagań oraz akceptacji końcowej”)

Po zidentyfikowaniu problemu i interesariuszy możemy zająć się określeniem **cech systemu**. Cechy systemu stanowią odpowiedź na potrzeby poszczególnych grup zainteresowanych systemem. Zgłaszą je najczęściej sami zainteresowani. Rolą inżyniera wymagań jest uporządkowanie tych cech oraz upewnienie się, że pokrywają one wszystkie zidentyfikowane problemy. Odkrywanie cech systemu może być efektem pracy grupowej, np. burz mózgów lub warsztatów wymagań. Cechy systemu określają dwa rodzaje wymagań: wymagania funkcjonalne i jakościowe.

Cechy są pojęciami na wysokim poziomie abstrakcji, i formułujemy je przy pomocy krótkich zdań w języku naturalnym. Dzięki temu są one użytecznym i wygodnym sposobem opisu funkcjonalności i jakości tworzonego systemu, bez wdawania się w szczegóły. Zapisywane są zwykle jako pojedyncze zdania lub krótkie akapity zawierające sformułowania typu „System musi pozwalać na...”, „System powinien wykonywać...”, „System powinien mieć...”, itp. Ważne jest, aby formułując cechy systemu korzystać ze słownictwa używanego przez interesariuszy. Pozwoli to na lepsze zrozumienie ich potrzeb i poprawi komunikację między różnymi grupami osób uczestniczących w przyszłym projekcie. Definicje poszczególnych pojęć powinniśmy umieścić w **słowniku**. Słownik na poziomie wizji nie musi być rozbudowany. W słowniku (np. w porządku alfabetycznym) umieszczamy używane w definicji cech systemu pojęcia, które mogą budzić wątpliwości. Każde takie pojęcie definiujemy w kilku zdaniach, uzgadniające definicję możliwe ze wszystkimi interesariuszami. Poniższy przykład ilustruje wykorzystanie słownika do wyjaśnienia pojęć występujących w opisie cech systemu.

- Cecha F007: System musi umożliwiać zarządzanie modelami samochodów.
- Cecha J023: System powinien przyspieszyć przyjmowanie samochodów na stan o co najmniej 30%.
- Model samochodu – specyfikacja obejmująca nazwę producenta i nazwę modelu (np. „Fiak Tipi” lub „Opek Astal”) oraz wszystkie parametry techniczne serii samochodów.
- Samochód – konkretny egzemplarz danego modelu samochodu, posiadający unikany numer pojazdu oraz określony zestaw dodatkowego wyposażenia.

Zwróćmy uwagę, że w opisach cech systemu wyróżniliśmy podkreśleniem pojęcia, które wymagają wyjaśnienia. Pojęcia te zostały zdefiniowane i umieszczone w słowniku. Można je następnie używać w opisach kolejnych cech systemu.

W powyższym przykładzie cechy systemu zostały oznaczone identyfikatorami. Przykładowo, cechy funkcjonalne są oznaczone identyfikatorem z przedrostkiem „F”, a cechy jakościowe – z przedrostkiem „J”. Identyfikator jest przykładem **atrybutu wymagania**. Atrybuty można nazwać meta-danymi wymagań. Są one używane do powiązania wymagań z danymi umożliwiającymi porządkowanie wymagań oraz zarządzanie projektem budowy oprogramowania. Na poziomie wizji systemu nadanie atrybutów może być praktykowane dla cech systemu. Na poziomie wymagań użytkownika i wymagań oprogramowania, atrybuty powinniśmy nadawać poszczególnym jednostkom wymagań, jak przypadki użycia czy historie użytkownika. Typowy zbiór atrybutów wymagań obejmuje następujące typy:

- Unikalny identyfikator,
- Typ (funkcjonalne, jakościowe, słownikowe, ograniczenie),
- Status – wartości statusu oznaczają kolejne kroki w procesie zbierania, zatwierdzania i realizacji wymagań (np. proponowane, zatwierdzone, zrealizowane),
- Waga dla zamawiającego – określa, w jakim stopniu dane wymaganie realizuje potrzeby zamawiającego, skala może mieć kilka poziomów, najczęściej od 3 do 10 (np. krytyczne, ważne, użyteczne),
- Trudność wykonania – określa poziom złożoności technicznej realizacji danego wymagania, skala zależy od użytej metodyki szacowania (np. duża, średnia, mała; 1, 2, 3, 5, 8, 13, ...),
- Ryzyko - poziom ryzyka związanego z niemożnością realizacji wymagania,
- Stabilność – na ile prawdopodobne jest, że dane wymaganie ulegnie zmianie,
- Osoba odpowiedzialna – kto odpowiada za sformułowanie wymagania.

Podczas specyfikowania wizji systemu należy również wziąć pod uwagę **ograniczenia**. Ograniczenie oznacza zmniejszenie stopnia swobody, którą mamy podczas tworzenia i dostarczania systemu. Ograniczenia mogą potencjalnie zmniejszyć możliwość dostarczenia rozwiązania a niektóre mogą nawet

wymusić zmianę technologii wytwarzania systemu. Dlatego też wszelkie ograniczenia należy zidentyfikować i opisać podając uzasadnienie wprowadzenie danego ograniczenia. Można wyróżnić wiele źródeł ograniczeń: techniczne, ekonomiczne, polityczne, prawne, środowiskowe i inne. Niektóre ograniczenia staną się wymaganiem stawianym systemowi. Inne będą miały wpływ na zasoby czy planowanie przedsięwzięcia.

Ostatnim, opcjonalnym lecz często użytecznym elementem wizji systemu jest **motto**. Motto ma postać jednego czy dwóch zdań, które prosto i efektywnie prezentują korzyści płynące z realizacji projektu. Ustala ono zwięzłe cel jaki przyświeca wszystkim uczestnikom projektu podczas dalszych prac. Spełnia zatem funkcję jednociągą. Motto można umieszczać na stronach tytułowych wszystkich dokumentów.

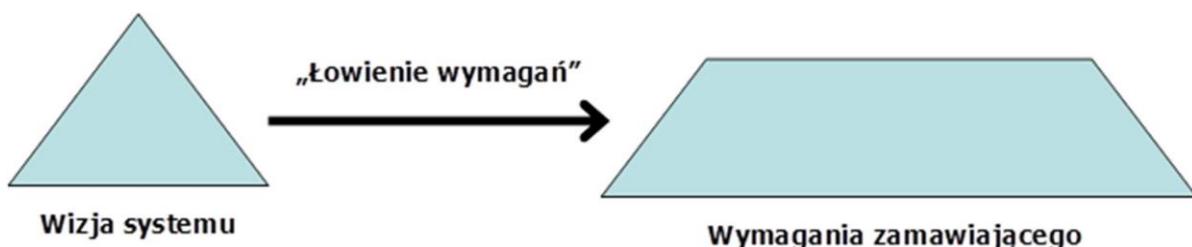
8.2. Specyfikowanie wymagań użytkownika

Wymagania użytkownika specyfikują wymagania na system w sposób wystarczający do określenia jego rozmiaru i nakładu pracy potrzebnej do jego zbudowania. Specyfikacja wymagań użytkownika powinna dostarczyć odpowiedzi na kluczowe w procesie zarządzania projektem pytania.

- Jak obszerna będzie funkcjonalność systemu?
- Jakie dane będzie przetwarzał system?
- Kto będzie się posługiwał systemem?
- Ile system będzie kosztował?

Współpraca twórców oprogramowania z zamawiającymi i użytkownikami, której wynikiem będzie sformułowanie wymagań użytkownika, powinna mieć charakter ciągły a nie jednorazowy. Wynika to z faktu, że wymagania użytkowników są zmienne. Gdy wprowadzamy użytkowników w dziedzinę potencjalnych rozwiązań ich problemów, często ich punkt widzenia i oczekiwania odnośnie do systemu ulegają zmianie. Zbieranie wymagań powinno odbywać się w cyklu iteracyjnym, co zapewni ciągłe wsparcie analityków, pozwali szybko reagować na zmienne potrzeby oraz zapobiegnie powstawaniu błędnych i niespójnych wymagań.

Współpracując z użytkownikami w celu pozyskiwania wymagań musimy cały czas mieć na uwadze wizję systemu. Na podstawie samej wizji systemu nie jesteśmy w stanie zawrzeć kontraktu, ze względu na jej wysoki poziom ogólności. Proces powstawania specyfikacji wymagań użytkownika możemy nazwać procesem „łowienia wymagań”, zilustrowanym na rysunku 8.2. Łowienie wymagań polega na aktywnym odkrywaniu szczegółowych potrzeb zamawiającego przy wykorzystaniu różnych technik. Jest to proces innowacyjny, w którym staramy się odkryć rzeczywiste potrzeby danej organizacji (ogólnie – odbiorców systemu) odnośnie przyszłego systemu (np. związane z usprawnieniem procesów wewnętrznych, konkurencyjności, czy po prostu – ułatwienia w wykonywaniu codziennych czynności). W procesie tym możemy wykorzystać takie techniki jak burze mózgów czy warsztaty wymagań.



Rysunek 8.2: Od wizji systemu do wymagań zamawiającego

Punktem wyjścia dla analityków oraz odbiorców przyszłego systemu przy formułowaniu wymagań użytkownika jest wizja systemu. Ważne jest, aby odkryte („wyłowane”) wymagania umożliwiały rozwiązanie zidentyfikowanych problemów i pokrywały niezbędne cechy systemu. Zidentyfikowane wymagania użytkownika dokumentujemy, aby mogły się stać podstawą do zawarcia kontraktu między zamawiającym a wykonawcą systemu, w tym - do określenia zasobów niezbędnych do realizacji systemu. Wymagania użytkownika powinny być zatem na tyle precyzyjne, aby nie było niedomówień co do zakresu systemu. Jednocześnie, powinny być na tyle ogólne, aby nie sugerować rozwiązań technicznych (dotyczących np. implementacji interfejsu użytkownika, baz danych, itp.), co jest dopiero zadaniem projektantów systemu.

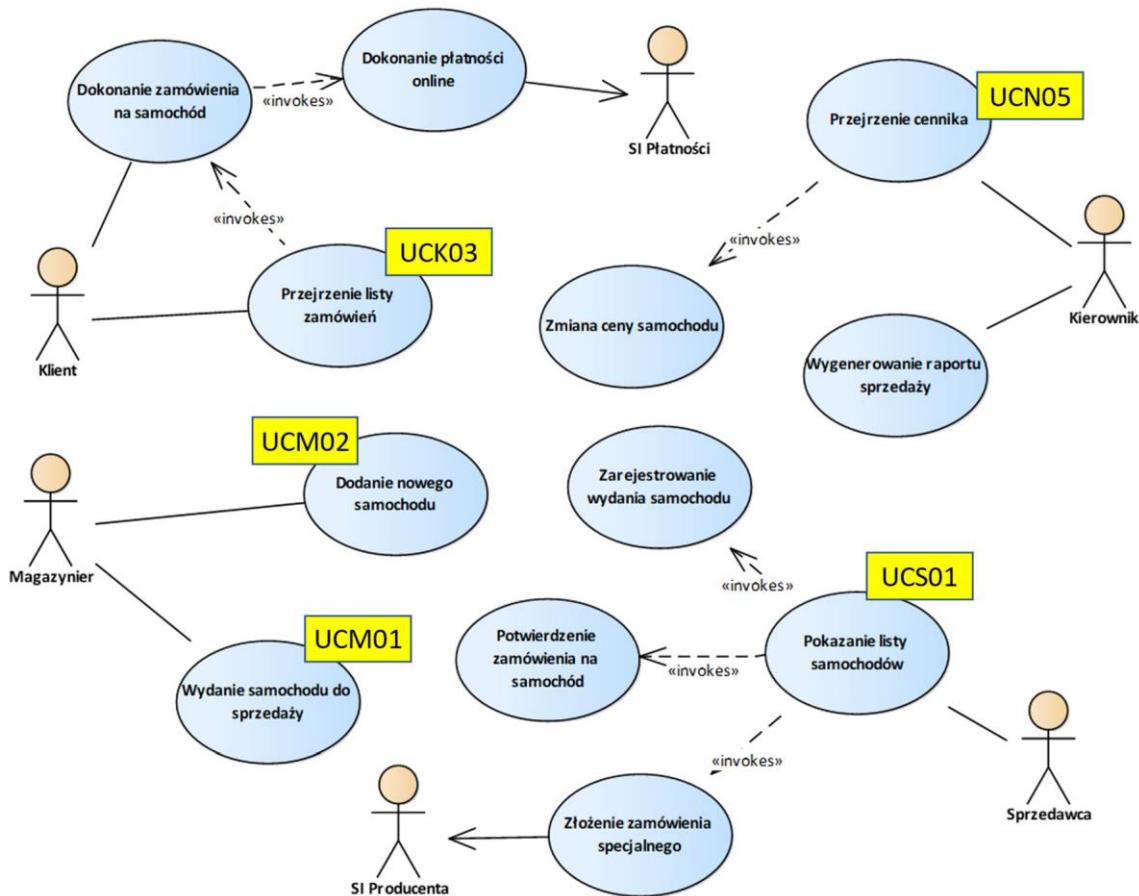
Zgodnie ze strukturą piramidy wymagań wprowadzoną w poprzednim rozdziale, na poziomie wymagań użytkownika formułujemy cztery rodzaje wymagań: wymagania funkcjonalne, jakościowe, słownikowe oraz ograniczenia. Poniżej przedstawimy podstawowe zasady formułowania wymagań użytkownika w tych czterech obszarach. Warto na wstępie zwrócić uwagę na to, że szczegółowe opisy wymagań użytkownika wykonywane są dopiero na poziomie wymagań oprogramowania. Ma to na celu uniknięcie podejmowania niepotrzebnych działań w sytuacji, kiedy wiele szczegółów może ulec zmianie w trakcie prac nad systemem (np. już po zawarciu formalnego kontraktu).

Jednym z najczęściej spotykanych sposobów zapisu **wymagań funkcjonalnych** są **przypadki użycia systemu**, omówione w module II (rozdział o specyfikowaniu dynamiki systemu). Notacja przypadków użycia ma tę zaletę, że jest łatwa do zrozumienia dla osób nie będących profesjonalnymi informatykami, np. dla zamawiających i użytkowników. Przypadki użycia są łatwe do opisania i dokumentowania. Dzięki temu analitycy ze strony twórcy systemu mogą pracować wspólnie z przyszłymi użytkownikami w celu zdefiniowania zachowania nowego systemu. Każdy użytkownik koncentruje się na możliwościach systemu wymaganych w celu sprawniejszego i wygodniejszego wykonywania swoich zadań biznesowych, przez co jego udział w specyfikacji wymagań przyczynia się do stworzenia systemu spełniającego w wysokim stopniu oczekiwania jego przyszłych użytkowników.

Podział wymagań na przypadki użycia, ułatwia zarządzanie zakresem systemu. Każdy przypadek użycia stanowi pewną zamkniętą, spójną całość. Dzięki temu, system można budować poprzez implementację kolejnych przypadków użycia. Zrealizowane przypadki użycia rozbudowują funkcjonalność systemu, którą użytkownik jest w stanie zweryfikować pod względem realizacji jego oczekiwani. Dzięki temu, twórcy systemu mogą szybko uzyskać informację zwrotną od użytkownika, dotyczącą poziomu akceptacji dla budowanego systemu. Przypadki użycia, pomimo swoich wielu zalet, są przede wszystkim odpowiednie do specyfikowania wymagań dla systemów interaktywnych. W systemach takich na pierwszy plan wsysuwa się konieczność określenia zasad interakcji systemu z jego użytkownikami. Systemami, dla których przypadki użycia mają ograniczone zastosowanie są takie, które mimo swojej złożoności mają niewielu użytkowników lub nie mają ich wcale oraz mają ubogie interfejsy zewnętrzne lub są zdominowane przez wymagania jakościowe. Są to np. systemy do obliczeń numerycznych, systemy sterujące procesami w czasie rzeczywistym, czy systemy wbudowane w urządzenia techniczne.

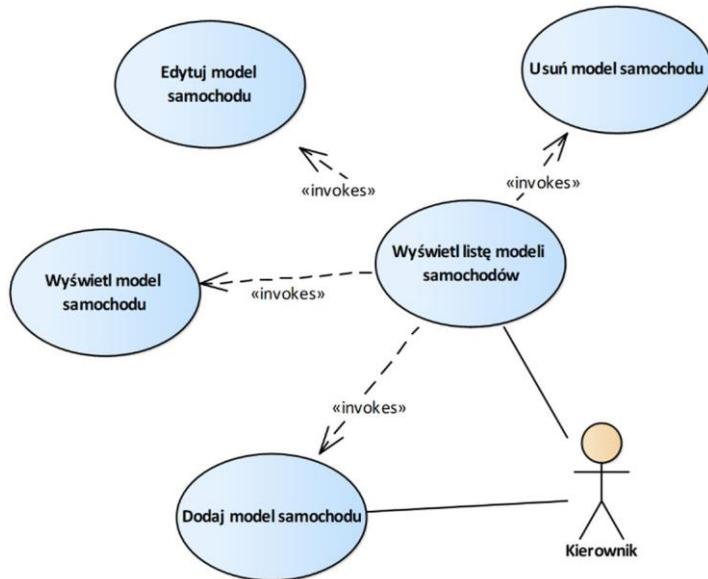
Punktem wyjścia do stworzenia modelu przypadków użycia systemu powinna być analiza cech systemu sformułowanych w wizji. Istotnym źródłem pomysłów na przypadki użycia może być również analiza procesów biznesowych. Dla typowego systemu oprogramowania, model przypadków użycia zawiera kilkadziesiąt a czasami kilkaset przypadków użycia. W takiej sytuacji, model należy podzielić na pakiety po kilka-kilkanaście przypadków użycia i dla każdego pakietu utworzyć osobny diagram. Na rysunkach 8.3 i 8.4 przedstawione są dwa przykładowe diagramy dla **systemu obsługi sprzedaży samochodów**. Każdy z przedstawionych przypadków użycia posiada ślad łączący go z odpowiednią cechą systemu oraz krokiem w procesie biznesowym. Niektóre z tych cech i fragmenty procesów zostały przedstawione w poprzedniej części tego rozdziału i w rozdziale poprzednim. Zwróciśmy uwagę na to, że przypadki użycia

mogą mieć nadane identyfikatory (np. „UCM02”), co często ułatwia komunikację w projekcie (patrz opisy metryk wymagań jakościowych na rysunku 8.7).



Rysunek 8.3: Przykładowe przypadki użycia dla systemu sprzedaży samochodów

Warto zwrócić uwagę na to, że diagram na rysunku 8.4 zawiera przypadki użycia realizujące pewien standardowy zestaw operacji na określonym elemencie danych (tu: model samochodu). Są to operacje typu CRUD (Create, Read, Update, Delete – utwórz, odczytaj, zaktualizuj, skasuj). Jest to sytuacja często spotykana w typowych systemach biznesowych i można ją uogólnić, tworząc tzw. **wzorzec przypadków użycia** (ang. use case pattern). W naszym przykładzie, zamiast frazy „model samochodu”, można umieścić inną, np. „użytkownik”, „pozycja magazynowa”, a ogólnie - „obiekt” (np. „Wyświetl listę obiektów”, „Dodaj obiekt”). Taki wzorzec można zatem zastosować w różnych dziedzinach poprzez prostą zamianę obiektu występującego w nazwach przypadków użycia.

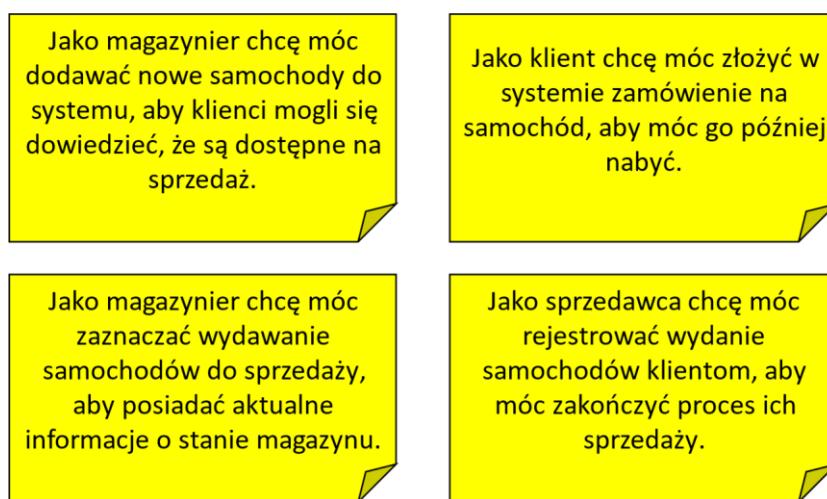


Rysunek 8.4: Przykład przypadków użycia realizujących operacje CRUD

Alternatywnym do przypadków użycia sposobem formułowania wymagań funkcjonalnych są **historie użytkownika**. Są one często używane w metodykach zwinnych (agile), co zostało omówione w module I. Historie użytkownika opisują oczekiwania klienta w stosunku do tworzonego systemu za pomocą krótkich zdań mieszczących się np. na kartkach z notesu. Zdanie stanowiące historię użytkownika powinno określać podmiot historii (użytkownik, który czegoś chce od systemu), cel biznesowy realizowany przez system i oczekiwany przez użytkownika, oraz motywację klienta stojącą za konkretnym oczekiwaniem. Do formułowania historii użytkownika można wykorzystać następujący prosty wzorzec:

- Jako <rodzaj użytkownika> chcę <jakiś cel>, aby <jakiś powód>

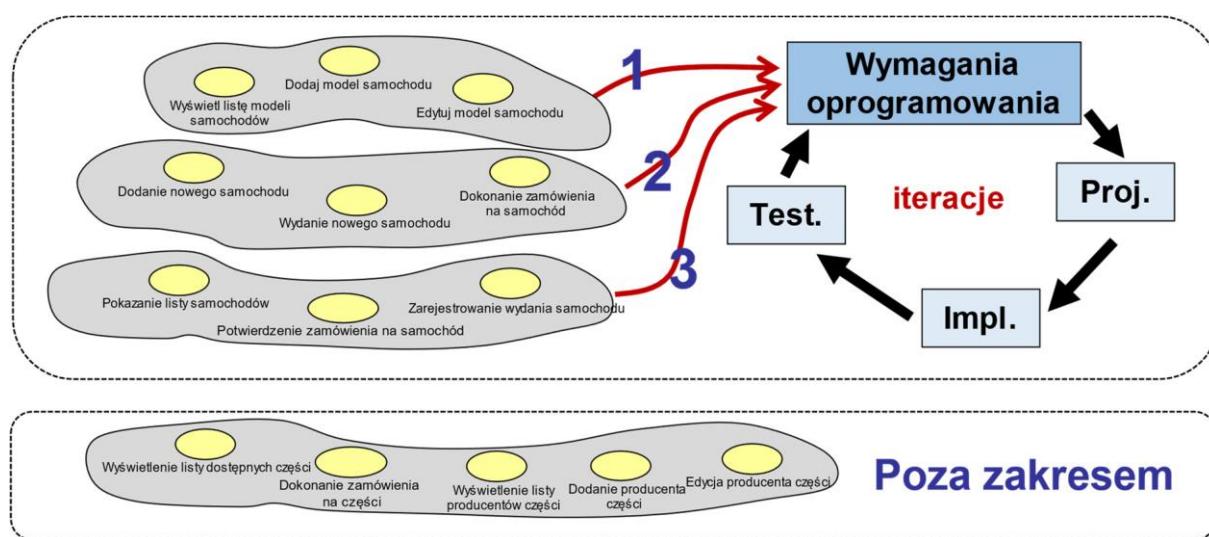
Szereg przykładowych historii użytkownika dla wcześniej przedstawionej wizji, napisanych zgodnie z powyższym wzorcem przedstawia rysunek 8.5. Zwróćmy uwagę na to, że historie użytkownika są analogiem przypadków użycia i można je użyć zamiennie. Informacja zawarta w modelu przypadków użycia (patrz rysunek 8.3) może być przedstawiona w postaci historii użytkownika (porównaj rysunek 8.5) i na odwrót. Różnica polega głównie na notacji (graficzna lub tekstowa) oraz nieco innym stylu zapisu wymagań. Dodatkowo, w treści historii użytkownika podajemy motywację, która nie występuje w modelu przypadków użycia (jeśli jest taka potrzeba, można ją zapisać w komentarzach).



Rysunek 8.5: Przykłady historii użytkownika

Przypadki użycia systemu i historie użytkownika są najczęściej używanymi jednostkami formułowania wymagań funkcjonalnych. Jednocześnie, mogą one pełnić rolę podstawowych jednostek **sterujących realizacją systemu**, w szczególności – definiować przyrosty systemu w iteracyjnym cyklu wytwarzania oprogramowania. Każdy przypadek użycia (historię użytkownika) można potraktować jako opis jednostkowego przyrostu funkcjonalności systemu. Stanowi on zamkniętą całość, której implementacja dostarcza użytkownikom systemu wymiernej korzyści. Możemy zatem oprzeć cykl budowy systemu na przypadkach użycia, rozbudowując system w każdej iteracji o funkcjonalność opisaną kolejnymi przypadkami. Taki cykl ilustruje rysunek 8.6. Przypadekom użycia zamawiający wspólnie z wykonawcami nadają odpowiednie priorytety. Na ich podstawie następuje przydział przypadków użycia do kolejnych iteracji. Odbywa się to na podstawie analizy ich atrybutów, a szczególnie – wagi dla zamawiającego i trudności wykonania (patrz poprzednia sekcja).

W pierwszej iteracji następuje realizacja pierwszej puli przypadków użycia. Są to przypadki opisujące najważniejszą funkcjonalność, jakiej zamawiający wymagają od systemu. Po zaimplementowaniu tych przypadków, system poddawany jest ocenie przez zamawiających. Na podstawie oceny zgłoszane są potrzeby ewentualnych zmian w wymaganiach. W drugiej iteracji realizowane są kolejne przypadki użycia. Jednocześnie uwzględnia się zmiany postulowane przez stronę zamawiającą. Po zrealizowaniu kolejnej puli przypadków użycia system ponownie podlega weryfikacji i cały cykl się powtarza. Oczywiście, wszelkie zmiany funkcjonalności są negocjowane z wykonawcą, tak aby z jednej strony kontrolować zakres projektu związany z czasem i kosztami jego wytwarzania, z drugiej zaś, aby stworzyć system spełniający potrzeby zamawiającego.



Rysunek 8.6: Iteracyjny cykl wytwórczy sterowany przypadkami użycia

Aby skutecznie sterować cyklem wytwórczym poprzez przypadki użycia, ważne jest odpowiednie określenie priorytetów. Umiejętnie wybranie przypadków użycia do realizacji w początkowych iteracjach jest kluczem do pokonania dwóch podstawowych zagrożeń: nietrafienie z wymaganiami w rzeczywiste potrzeby zamawiającego oraz nieodpowiednia architektura, która nie pozwala zbudować systemu realizującego potrzeby zamawiającego. Dlatego też, z jednej strony, powinniśmy wybrać przypadki użycia, które realizują najważniejsze dla zamawiającego potrzeby (atrribut „waga dla zamawiającego”). Z drugiej strony, powinny to być takie przypadki, które pozwolą w jak najszerzym stopniu zweryfikować architekturę całego systemu (atrribut „trudność wykonania”). Pośród wszystkich zaproponowanych przypadków użycia znajdą się też takie, które pozostaną poza zakresem systemu i – z racji określonego budżetu i ram czasowych czy ze względu na trudności techniczne – nie zostaną zrealizowane. Są to

zazwyczaj przypadki opisujące funkcjonalność, bez której system pozostanie nadal użyteczny dla zamawiającego.

Minimalny system, który jest jeszcze użyteczny dla zamawiającego nazywamy **minimalnie opłacalnym produktem** (ang. Minimum Viable Product, MVP). Projekt powinien być tak zaplanowany, aby jak najwcześniej (po jak najmniejszej liczbie iteracji) uzyskać taki właśnie system i poddać go ocenie. W ten sposób znacznie ograniczamy ryzyko niepowodzenia. Najbardziej krytyczne zagrożenia mogą być wcześniej wykryte i skorygowane – przed dokonaniem dużych nakładów pracy. Postęp prac jest mierzony rzeczywiście zaimplementowaną funkcjonalnością a w razie przerwania projektu możliwe jest wdrożenie niekompletnego systemu realizującego najważniejszą funkcjonalność. Pozostała funkcjonalność może być zrealizowana w późniejszym czasie lub przez innego twórcę oprogramowania.

Drugim kluczowym składnikiem specyfikacji wymagań zamawiającego jest specyfikacja **wymagań jakościowych**. Jak wiemy (patrz poprzedni rozdział), wymagania jakościowe mogą w bardzo znaczącym stopniu wpływać na architekturę systemu, a tym samym – na koszt jego wykonania. Bardzo istotne jest zatem, aby zapewnić kompletność i jednoznaczność tego rodzaju wymagań. Kontrolę kompletności może nam zapewnić zachowanie zgodności z określona taksonomią wymagań jakościowych. Istnieje wiele takich taksonomii, przy czym tutaj przestawimy chyba najbardziej szczegółową i wyczerpującą z nich, zawartą w normie ISO/IEC 25010:2011. Jest to standard przemysłowy, oparty na doświadczeniach z wielu rzeczywistych projektów. Zawiera ona nie tylko podział wymagań jakościowych na typy, lecz także sposoby ich weryfikacji (metryki). Model jakości opisany w normie ISO 25010 zawiera następujący podział rodzajów wymagań jakościowych:

- Przydatność funkcjonalna (ang. Functional suitability) – dotyczy spełniania przez system wyrażonych wcześniej potrzeb. Obejmuje między innymi wymagania dotyczące pokrycia oczekiwanych od systemów funkcji (kompletność funkcjonalna), dokładność z jaką te funkcje są spełniane (poprawność funkcjonalna) i stopnia w jakim ułatwiają one osiąganie użytkownikom ich celów (adekwatność funkcjonalna).
- Efektywność wydajnościowa (ang. Performance efficiency) – dotyczy wydajności osiąganej przez system przy założonych zasobach. Obejmuje między innymi wymagania dotyczące czasu działania, zużycia zasobów i wielkości zadań, z którymi musi radzić sobie system.
- Kompatybilność (ang. Compatibility) – dotyczy współpracy z innymi systemami oprogramowania. Obejmuje między innymi wymagania dotyczące wymiany informacji z innymi systemami i możliwości efektywnego działania współdzierając z nimi zasoby.
- Użyteczność (ang. Usability) – dotyczy dostosowania sposobu użycia systemu do potrzeb użytkowników. Obejmuje między innymi wymagania dotyczące łatwości nauki i używania systemu, zabezpieczenia przed błędami, estetyki i ułatwień dostępu. Specjalnym rodzajem wymagań w tej kategorii jest tzw. rozpoznawalność odpowiedniości, czyli łatwość z jaką użytkownicy są w stanie stwierdzić, czy system będzie przydatny dla ich celów.
- Niezawodność (ang. Reliability) – dotyczy zapewnienia określonego poziomu działania systemu przy założonych zasobach. Obejmuje między innymi wymagania dotyczące dojrzałości, dostępności, odporności na błędy i przywracania systemu do działania po awariach.
- Bezpieczeństwo (ang. Security) – dotyczy ochrony danych przez system. Obejmuje między innymi wymagania dotyczące zapewniania poufności i integralności danych oraz jednoznacznego rejestrowania wykonywanych przez system i użytkowników akcji.

- Łatwość utrzymania (ang. Maintainability) – dotyczy nakładów pracy koniecznych podczas utrzymywania systemu w eksploatacji. Obejmuje między innymi wymagania dotyczące zarówno dostosowania do tego celu samej implementacji systemu (reżywalność, testowalność), jak i łatwości znajdywania i naprawy defektów w trakcie jego działania.
- Przenośność (ang. Portability) – dotyczy możliwości działania oprogramowania na różnych środowiskach. Obejmuje między innymi wymagania dotyczące dostosowywania się oprogramowania do charakterystyk środowiska, nakładu pracy koniecznego do jego zainstalowania czy zdolności do zastąpienia innych aplikacji.

Z uwagi na wielość klasyfikacji i typów wymagań jakościowych nie istnieje jednolity, sformalizowany sposób ich zapisu. Najczęściej wymagania tego typu tworzone są jako paragrafy tekstu naturalnego (nie podlegającego gramatyce kontrolowanej). Czasami elementy specyfikacji wymagań jakościowych wskazują na wymagania funkcjonalne (np. przypadki użycia), których one dotyczą. Często jednak, wymagania jakościowe stanowią przekrój przez wiele wymagań funkcjonalnych lub dotyczą funkcjonalności całego systemu.

Aby móc sprawdzić stopień realizacji wymagań jakościowych należy określić dla nich metryki i związane z tym sposób testowania. Jest to problem dosyć złożony, gdyż wymagania opisujące charakterystykę pozafunkcjonalną systemu nie mają tak precyzyjnej formy jak wymagania funkcjonalne, które mogą w sposób precyzyjny (krok po kroku) określać działanie aplikacji. Dodatkowo, różne rodzaje wymagań jakościowych (patrz powyższa klasyfikacja) podlegają różnym miarom i metodom testowania.

Istotą testów cech jakościowych systemu jest jak największa obiektywizacja ich pomiaru. Pomiar można zdefiniować jako proces, w którym liczby lub znaki są przypisywane atrybutom obiektów świata rzeczywistego w taki sposób, aby opisać je zgodnie z jednoznacznie zdefiniowanymi zasadami. Pomiarom towarzyszą metryki – miary własności. Biorąc pod uwagę takie ogólne określenie pomiaru, można określić ogólny schemat definiowania metryk wymagań jakościowych. W skład metryki powinny wchodzić następujące elementy:

- nazwa wymagania i jego atrybuty,
- opis wymagania,
- sposób pomiaru (w tym: wymaganą liczbę wykonywanych pomiarów),
- zbiór możliwych wartości pomiaru (wraz z jednostką dla wartości liczbowych),
- oczekiwana (akceptowalna) wartość (w tym: charakterystykę błędów).

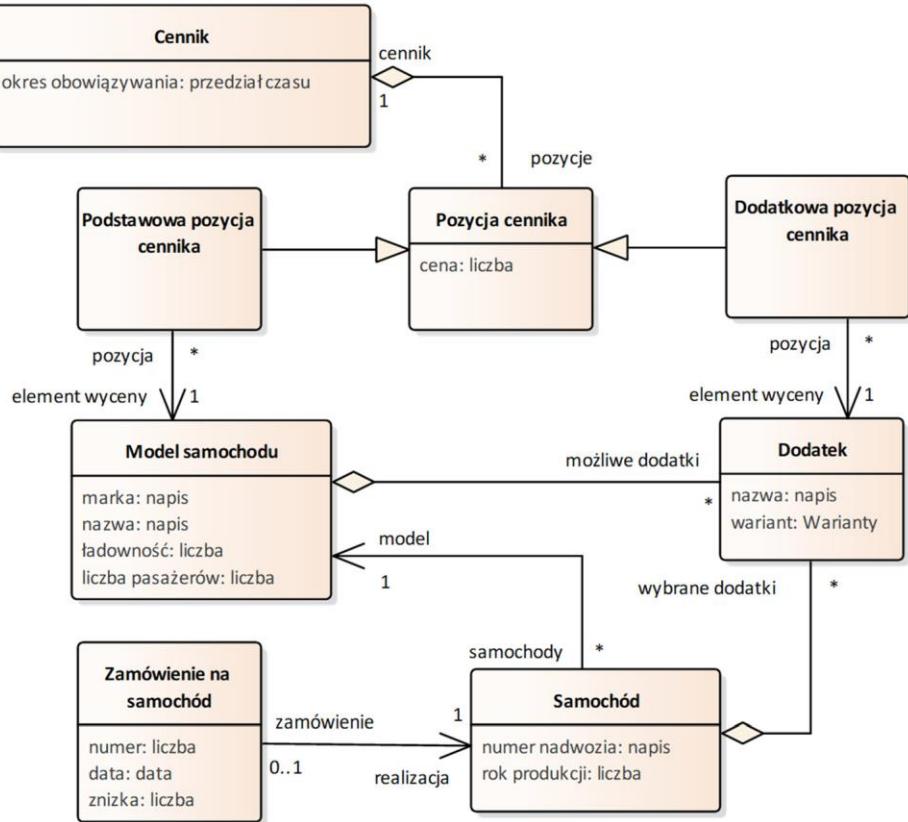
Przykładowo, możemy w oparciu o analizę działania przedsiębiorstwa dojść do wniosku, że wcześniej przedstawiona cecha J023 powinna się przekładać na wykonanie w systemie czynności związanych z dodaniem na stan nowego samochodu w czasie poniżej 2 minut. W takim wypadku opis odpowiedniego wymagania jakościowego wraz z jego metryką może wyglądać tak, jak przedstawiono na rysunku 8.7 – wymaganie JU038. Zwróćmy uwagę na to, że treść wymagania JU038 dotyczy czasu, ale nie jest to wymaganie rodzaju „Efektywność wydajnościowa (czas)”. Jest to wymaganie użyteczności, gdyż dotyczy sprawności posługiwania się systemem przez użytkowników po odpowiednim przeszkoleniu. Efektywności wydajnościowej dotyczy natomiast wymaganie JW121. Jest ono powiązane z tym samym przypadkiem użycia, co dla wymagania JU038, lecz tym razem opisuje sprawność działania (czas odpowiedzi) systemu.

<p>JU038: Czas dodawania nowego samochodu (użyteczność - operacyjność) Waga: ważne; Trudność: 5</p> <p>Opis Średni czas wykonania w systemie czynności w ramach przypadku użycia UCM02 nie powinien przekraczać 2 minut.</p> <p>Sposób pomiaru Zmierzenie średniego czasu, jaki zajmie magazynierowi dodanie nowego samochodu w systemie. Pomiaru dokonujemy na próbie 10 osób po 30 minutowym przeszkoleniu, z których każda przyjmie na stan po 10 samochodów.</p> <p>Zbiór możliwych wartości pomiaru 10 sekund – 10 minut</p> <p>Oczekiwana wartość Średnio < 2 minuty, najgorszy wynik < 4 minuty</p>	<p>JW121: Czas zapisu danych nowego samochodu (efektywność wydajnościowa - czas) Waga: istotne; Trudność: 2</p> <p>Opis Średni czas od momentu wprowadzenia danych samochodu do potwierdzenia ich dodania do systemu w ramach przypadku użycia UCM02 nie powinien być większy niż 0,5 sekundy</p> <p>Sposób pomiaru Zmierzenie średniego czasu, jaki zajmie systemowi dokonanie walidacji oraz zapisanie przez system danych nowego samochodu. Pomiaru dokonujemy na próbie 1000 uruchomień przypadku użycia dla poprawnych danych samochodu.</p> <p>Zbiór możliwych wartości pomiaru 0,01 sekundy – 10 sekund</p> <p>Oczekiwana wartość Średnio < 0,5 sekundy dla obciążenia systemu 1% Średnio < 1,0 sekundy dla obciążenia systemu 5%</p>
---	--

Rysunek 8.7: Przykłady wymagań jakościowych wraz z metrykami

W celu zapewnienia spójności specyfikacji wymagań konieczne jest zdefiniowanie używanego słownictwa. Na poziomie wymagań użytkownika definicje pojęć danej dziedziny problemu powinny być znacznie bardziej precyzyjne niż na poziomie wizji. Pojęcia umieszczały w **słowniku dziedziny**. Słownik dziedziny stanowi uporządkowany zbiór pojęć wraz z ich znaczeniami, za pomocą których można opisać wszystkie aspekty związane z określonym fragmentem rzeczywistości, a które są używane w specyfikacji wymagań użytkownika. Dzięki zastosowaniu centralnego słownika, jesteśmy w stanie uporządkować i ujednolicić używaną w specyfikacji terminologię, przy czym dotyczy to zarówno terminologii w wymaganiach funkcjonalnych i jakościowych. Pojęcia w słowniku tworzą sieć powiązań (relacji) wzajemnych, tworząc pewnego rodzaju mapę danej dziedziny problemu. Taką mapę najlepiej jest przedstawić w sposób wizualny, dzięki czemu dobrze są widoczne wszystkie zależności.

Przykładowa reprezentacja graficzna słownika została przedstawiona na rysunku 8.8. Zawiera ona wybrane pojęcia z dziedziny sprzedaży samochodów, czyli tej, której dotyczą wymagania przedstawione w poprzedniej części tego rozdziału. Jak widzimy, została tutaj użyta notacja języka UML. Nazwy klas odpowiadają nazwom pojęć danej dziedziny problemu. Zwróćmy uwagę na to, że nazwy te są zapisane w języku naturalnym, czyli np. posiadają spacje. Należy bardzo uważać na to, aby słownik zapisany w postaci modelu w języku UML nie był traktowany jako artefakt projektowy, czyli dotyczący opisu kodu systemu. Wszystkie elementy modelu powinny być zrozumiałe dla typowego użytkownika systemu, nie mającego przygotowania informatycznego. Na przykład, typy atrybutów powinny mieć nazwy potoczne w języku naturalnym (np. „liczba”, a nie „int” lub „float”). W razie wątpliwości, nazwy typów można na etapie wymagań użytkownika w ogóle pominąć. Bardzo istotne jest jednak określenie krotkości relacji, gdyż może ono mieć zasadniczy wpływ na ocenę pracochłonności wykonania systemu. Na przykład, istotną informacją może być to, że zamówienie na samochód zawsze dotyczy jednego samochodu (patrz krotkość „1” na rysunku 8.8), i nie może dotyczyć wielu samochodów (wymagana wtedy byłaby krotkość „*”). Na diagramach możemy również używać relacji agregacji (zawieranie) oraz relacji generalizacji (uogólnienie), które mogą posłużyć np. do przedstawienia taksonomii.



Rysunek 8.8: Przykładowa reprezentacja graficzna słownika dziedzinowego jako diagramu klas

Model dziedziny, zapisany np. jako model klas w języku UML stanowi istotną - trzecią kolumnę (wymagania słownikowe) w piramidzie wymagań na poziomie wymagań użytkownika. Tworzymy go równolegle z innymi wymaganiami już od samego początku cyklu życia oprogramowania. Rozbudowujemy go w trakcie specyfikowania zakresu systemu, a następnie uszczegóławiamy w ramach specyfikowania wymagań oprogramowania (patrz następna sekcja). Słownik jest podstawą sprawnej komunikacji wszystkich osób biorących udział w procesie wytwarzania oprogramowania – od zamawiających i przyszłych użytkowników, poprzez analityków, architektów i projektantów do programistów i osób odpowiedzialnych za testy.

8.3. Specyfikowanie wymagań oprogramowania

Wymagania oprogramowania stanowią uszczegółowienie wymagań użytkownika. Na tym poziomie piramidy wymagań, przypadki użycia zdefiniowane podczas formułowania wymagań użytkownika są opisywane w szczegółach, poprzez m.in. zdefiniowanie interakcji pomiędzy aktorami a systemem w postaci scenariuszy przypadków użycia. W miarę tworzenia scenariuszy uszczegóławiany i uzupełniany jest słownik dziedziny oraz ustalany jest wygląd interfejsu użytkownika. Powstają również tzw. scenariusze, które łączą opis działania przypadków użycia z wyglądem poszczególnych „scen”, czyli elementów wyświetlanych użytkownikowi. Kompletna specyfikacja wymagań oprogramowania zawiera również rozbudowaną specyfikację wymagań jakościowych, gdzie nowe lub zaktualizowane wymagania wynikają z uszczegółowienia wymagań funkcjonalnych i słownika. Źródłem dla wymagań oprogramowania jest zatem definicja zakresu systemu w postaci modelu przypadków użycia oraz powiązany z nimi słownik (model) dziedziny.

Podsumowując rolę wymagań oprogramowania, można powiedzieć, że dają one odpowiedź na trzy pytania:

- Jakie są szczegóły funkcjonowania systemu?
- Jakie dane będą przetwarzane przez system?
- Jakie inne szczegółowe cechy powinien mieć system?

Tego typu szczegóły są bardzo podatne na zmiany w miarę postępów prac nad systemem. Dlatego też, zazwyczaj formułuje się je bezpośrednio przed implementacją wybranego fragmentu funkcjonalności systemu. W cyklu iteracyjnym, dopiero po wybraniu zestawu przypadków użycia (lub historii użytkownika) do realizacji w danej iteracji, dokonuje się ich szczegółowego wyspecyfikowania. Piszymy wtedy szczegółowe scenariusze oraz projektujemy wygląd odpowiednich elementów interfejsu użytkownika. Ponadto, definiujemy dane, które będą przetwarzane przez system w ramach realizacji wybranego zestawu przypadków użycia. Szczegółowo opisujemy także, jakie dane będą wymieniane z użytkownikami i innymi systemami. Szczegółowy opisujemy składniki formularzy (pole, przyciski itp.), elementów graficznych (wykresy, diagramy, obrazy itp.), menu (opcje), okienek komunikatów, a także innych elementów interfejsu użytkownika. Tutaj również musimy ustalić wygląd elementów, np. kolor, położenie, rozmiar.

Scenariusze określają w sposób precyzyjny interakcję użytkownika z systemem prowadzącą do osiągnięcia określonego celu biznesowego. Scenariusze na poziomie wymagań oprogramowania powinny określać wszystkie zidentyfikowane alternatywne ścieżki wykonania przypadku użycia oraz obsługę sytuacji wyjątkowych (np. spowodowanych wystąpieniem błędu działania systemu).

Scenariusz przypadku użycia powinien spełniać trzy cechy:

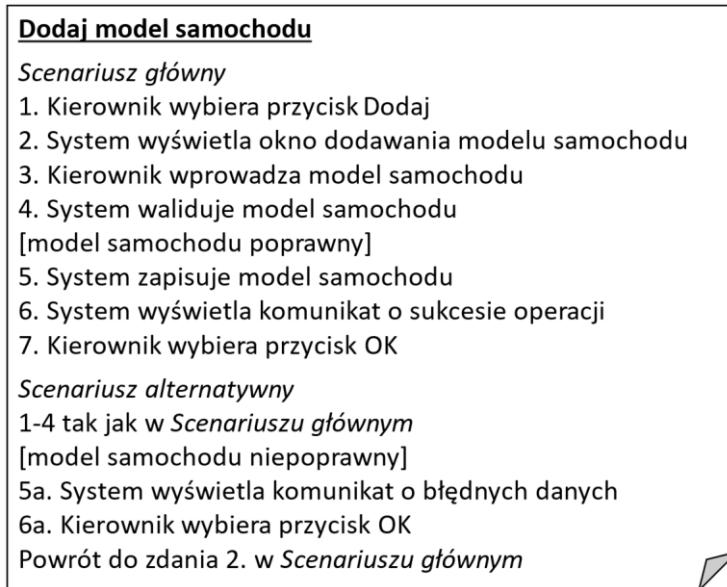
- opisywać sekwencję interakcji między aktorem i systemem,
- zaczynać się od interakcji aktora (w większości przypadków)
- kończyć się, kiedy zostanie osiągnięty cel przypadku użycia, lub kiedy na drodze do tego celu nastąpiła porażka.

Przypadek użycia jest zbiorem scenariuszy prowadzących do tego samego celu, w tym tych zakończonych porażką (rezygnacją, błędem lub sytuacją wyjątkową). Scenariusze przypadku użycia powinny definiować wszystkie alternatywne przebiegi obsługujące sytuacje wyjątkowe, z wyjątkiem tych najbardziej oczywistych, które mogą być opisane dla wielu przypadków użycia wspólnie (np. anulowanie wykonania operacji).

Scenariusz powinien posiadać swoją nazwę. W zależności od potrzeb, nazwa ta może być ogólna (np. „scenariusz główny”, „scenariusz alternatywny 1”) lub opisywać konkretną sytuację (np. „poprawna rejestracja samochodu”, „dane samochodu nieprawidłowe”). Pierwsze zdanie scenariusza jest w większości przypadków akcją inicjującą sekwencję współpracy pomiędzy użytkownikiem a systemem. Sekwencja taka jest zawsze inicjowana przez aktora dla głównych przypadków użycia (połączonych bezpośrednio z aktorem). Dla przypadków użycia połączonych relacjami z innymi przypadkami użycia możliwa jest sytuacja, kiedy pierwszą akcją jest akcja wykonywana przez system. Scenariusz może się zakończyć sukcesem (osiągnięciem celu biznesowego) bądź porażką (np. w wypadku wystąpienia sytuacji wyjątkowej).

Przykłady scenariuszy przypadku użycia spełniające powyższe zasady widzimy na rysunku 8.9. Stanowią one opis przypadku użycia „Dodanie nowego samochodu” (UCM02) z rysunku 8.3. Zdania scenariuszy w tym przykładzie pisane są w prostej kontrolowanej gramatyce, która z jednej strony jest wystarczająca do kompletnego opisania interakcji aktor-system, a z drugiej na tyle prosta, aby scenariusze były

przejrzyste i jednoznaczne. Oprócz zdań w gramatyce kontrolowanej opisujących sekwencje wymiany komunikatów pomiędzy systemem a użytkownikiem, w scenariuszu mogą występować również zdania sterujące, umożliwiające zmianę przepływu sterowania wewnętrz przypadku użycia (warunki, powroty) oraz pomiędzy różnymi przypadkami użycia (wywołania).



Rysunek 8.9: Przykładowe scenariusze przypadku użycia

Podstawowym typem zdania scenariusza w gramatyce kontrolowanej jest **zdanie typu POD(D)** (Podmiot, Orzeczenie, Dopełnienie bliższe, opcjonalne Dopełnienie dalsze). Zdanie POD(D) składa się z podmiotu określającego wykonawcę akcji (np. „system”), orzeczenia określającego wykonaną akcję (np. „zapisuje”) oraz dopełnień określających pojęcia, którego ta akcja dotyczy (np. „model samochodu”). Zdefiniowanie zdań w takiej ograniczonej gramatyce pozwala na łatwe wyróżnienie pojęć słownikowych (podmiotów i dopełnień) oraz powiązanych z nimi operacji (orzeczeń). Zdania POD(D) odpowiadają pojedynczym komunikatom wymienianym pomiędzy użytkownikiem a systemem. Zdania takie numerujemy w celu podkreślenia ich kolejności w sekwencji interakcji. Zwróćmy uwagę na to, że podmiotem zdań POD(D) jest zawsze system bądź aktor (z reguły aktor główny, lecz może to być również aktor pomocniczy biorący udział w interakcji).

Zdania typu POD(D) możemy podzielić na trzy kategorie ze względu na nadawcę i odbiorcę komunikatu:

- zdania aktor-system; są to komunikaty od aktora do systemu (z reguły wprowadzenie danych bądź naciśnięcie przycisku),
- zdania system-aktor; są to komunikaty od systemu do aktora, np. wyświetlenie formularza bądź okienka komunikatu,
- zdania system-system; opisują wewnętrzne operacje systemu jak np. walidację danych wprowadzonych przez użytkownika lub wykonanie przetwarzania danych.

Warto zwrócić uwagę, że scenariusze nie zawierają zdań aktor-aktor. Wynika to z faktu, że komunikacja między aktorami nie jest elementem specyfikowania wymagań na system oprogramowania. Takie zdania opisywałyby akcje, które nie mają bezpośredniego wpływu na działanie systemu. W scenariuszach opisujemy jedynie interakcje, które wymagają udziału systemu i skutkują odpowiednią logiką jego działania.

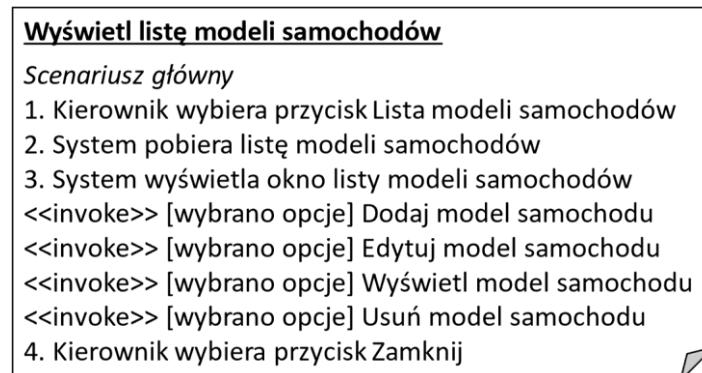
Zdania sterujące warunkowe pozwalają na kontrolowanie przebiegu scenariuszy przypadków użycia. Użycie zdania warunkowego pozwala na rozgałęzienie sterowania w zależności od spełnienia bądź

niespełnienia danego warunku. We naszym przykładzie (rysunek 8.9) po zdaniu nr 4 („System waliduje model samochodu”) może nastąpić różny przebieg dalszego ciągu interakcji. W zależności od wyniku tej operacji, wykonanie może przebiec zgodnie ze scenariuszem głównym bądź alternatywnym. Jeśli warunek jest spełniony, wykonywane są kolejne kroki danego scenariusza głównego. Jeśli warunek nie jest spełniony, należy sprawdzić warunki występujące po tym samy zdaniu w innych scenariuszach. Sterowanie interakcją podejmuje wtedy ten scenariusz, dla którego spełniony jest odpowiedni warunek. Zdania warunkowe zapisywane są w nawiasach kwadratowych. Wystąpienie takiego zdania definiuje nam rozgałęzienie się scenariuszy, tym samym wymaga dodania nowego scenariusza alternatywnego.

Warto zwrócić uwagę na to, że wszystkie zdania występujące przed zdaniem warunkowym są identyczne dla scenariusza pierwotnego i alternatywnego. Aby jednak nie dokonywać niepotrzebnych powtórzeń zdań, w scenariuszu alternatywnym można się odwołać do zdań w scenariuszu pierwotnym (np. „1-4 jak w scenariuszu głównym”). Dla zwiększenia czytelności, zdania scenariuszy alternatywnych powinny być oznaczane w sposób pozwalający na ich rozróżnienie od zdań scenariusza źródłowego. W powyższym przykładzie przyjęto konwencję oznaczania zdań kolejnych scenariuszy alternatywnych kolejną literą (np. 5a). Scenariusz alternatywny może zarówno prowadzić do spełnienia celu biznesowego (prowadzi do jego spełnienia inna drogą niż scenariusz główny) jak i do porażki (może np. obsługiwać sytuację wyjątkową lub błąd).

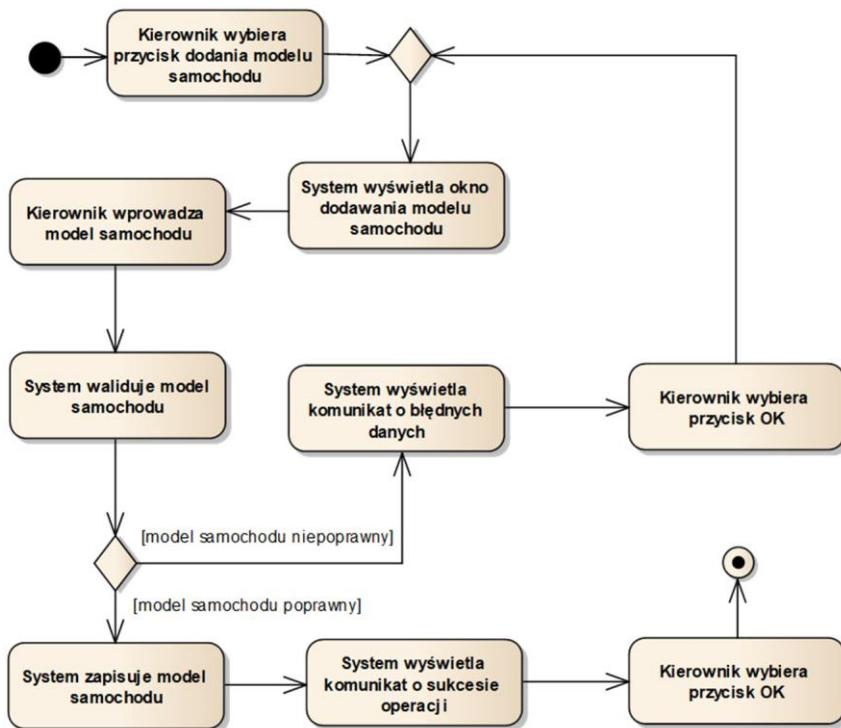
Zdania powrotu pozwalają na ponowne połączenie rozgałęzionych scenariuszy bądź zdefiniowanie powtarzanych interakcji (pętli). Przykładem takiego zdania jest ostatnie zdanie scenariusza alternatywnego na rysunku 8.9. Definiując powrót trzeba określić scenariusz i numer zdania, które nastąpi po wykonaniu wszystkich zdań danego scenariusza. Po dotarciu do końca scenariusza nastąpi przejście do wykonywania wskazanego zdania.

Zdania wywołania (ang. invoke) określają krok scenariusza, w którym uruchamiany jest inny przypadek użycia. Mogą one mieć charakter zarówno warunkowy, jak i bezwarunkowy. Zdania tego typu możemy na przykład oznaczyć stereotypem «invoke», opcjonalnym warunkiem (w nawiasach kwadratowych) i nazwą przypadku użycia. Po wykonaniu wywołanego przypadku użycia sterowanie powraca do bieżącego przypadku i wykonanie przypadku jest kontynuowane od miejsca wywołania. Przykład scenariusza przypadku użycia zawierającego takie zdania został przedstawiony na rysunku 8.10. Warto zwrócić uwagę na to, że scenariusze przypadku użycia powinny opisywać działania związane tylko bezpośrednio z tym przypadkiem użycia. Szczególnie, dotyczy to sytuacji kiedy dany przypadek wykorzystuje funkcjonalność innego przypadku użycia za pomocą zdań wywołania. W scenariuszu przypadku użycia zawierającego zdania «invoke» nie powinno się umieszczać zdań opisujących funkcjonalności „wywoływanej” przypadku użycia.



Rysunek 8.10: Przykładowy scenariusz przypadku użycia z zdaniami wywołań

Alternatywnym sposobem przedstawiania scenariuszy przypadków użycia jest prezentacja graficzna za pomocą diagramów czynności. Pozwala to na przedstawienie wielu scenariuszy na jednym diagramie, co w wielu przypadkach pomaga zrozumieć logikę działania danej aplikacji. Taka forma może być pomocna w trakcie analizy możliwych rozgałęzień scenariuszy oraz poszukiwaniu scenariuszy alternatywnych. Przykład takiej reprezentacji logiki przypadku użycia został przedstawiony na rysunku 8.11. Jak łatwo zauważyć, diagram ten ma identyczną zawartość informacyjną jak zestaw scenariuszy pokazany na rysunku 8.9. Wykorzystana została standardowa notacja diagramów czynności języka UML, omówiona w module II.

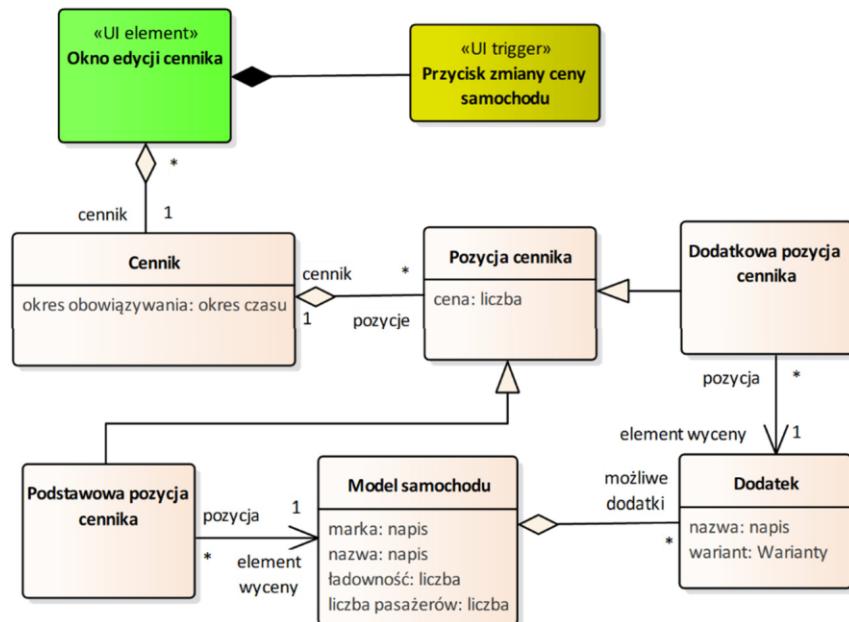


Rysunek 8.11: Przykład reprezentacji scenariusza przypadku użycia na diagramie czynności

Podczas specyfikowania wymagań oprogramowania, korzystamy ze **słownika** stworzonego na etapie wymagań użytkownika. Słownik ten uzupełniamy o pojęcia pojawiające się podczas tworzenia szczegółowych wymagań systemowych. Definicje pojęć już istniejących w słowniku są również uszczegóławiane na tym etapie. Najistotniejszymi nowymi pojęciami w słowniku są elementy interfejsu użytkownika, które będą konieczne do realizacji planowanej funkcjonalności systemu oraz ich powiązania z przetwarzanymi przez system danymi. Typowo odpowiadają one elementom występującym jako dopełnienia w zdaniach POD(D) scenariuszy przypadków użycia. W szczególności, dotyczy to zdań związanych z prezentacją elementów interfejsu użytkownika oraz interakcji użytkownika z systemem, poprzez takie elementy.

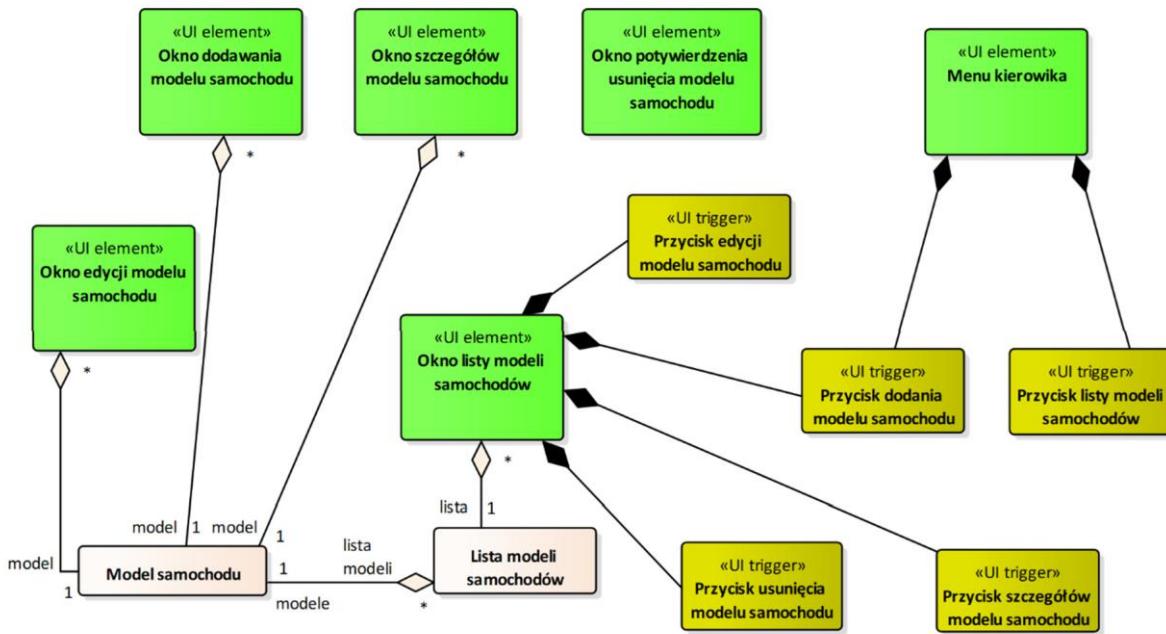
Przykłady fragmentów słownika rozbudowanego o elementy interfejsu użytkownika zostały pokazane na rysunkach 8.12 i 8.13. Pierwszy z nich obejmuje elementy związane z pojedynczym przypadkiem użycia („Edytuj model samochodu”), podczas gdy drugi dotyczy całą grupy (pozostałe przypadki użycia z rysunku 8.4). Na diagramach klas, elementy interfejsu użytkownika zostały oznaczone stereotypami. Elementy główne (okienka) zostały oznaczone stereotypem «UI element», a elementy aktywne (opcje, przyciski) – stereotypem «UI trigger» (wyzwalacz). Z okienkami zostały powiązane relacjami agregacji odpowiednie klasy słownika dziedziny. Pozwala to określić zakres danych, które będą wizualizowane przez dany element ekranowy. Przykładowo, z diagramu na rysunku 8.12 wynika, że „Okno

“edycji cennika” będzie umożliwiało modyfikację danych cennika, oraz danych poszczególnych jego pozycji, łącznie z informacją o modelu samochodu lub dodatku.



Rysunek 8.12: Przykład słownika na poziomie wymagań oprogramowania

Należy zwrócić uwagę na to, że diagramy klas przedstawiające elementy interfejsu użytkownika nie powinny służyć do pokazywania nawigacji między tymi elementami. Są to bowiem diagramy przedstawiające strukturę, a nie dynamikę działania systemu. Informacje o następstwach przyczynowo skutkowych, jak np. powiązania pomiędzy przyciskami a oknami których otwarcie one powodują nie są tutaj pokazane. Takie informacje są już zawarte w treści scenariuszy i powtarzanie ich na diagramach klas nie jest uzasadnione, a wręcz można uznać, że jest błędne. Przykładowo, na rysunku 8.13 widzimy element aktywny – „Przycisk listy modeli” zawarty w okienku („Menu kierownika”). Naciśnięcie tego elementu powoduje wyświetlenie „Okna listy modeli samochodów”. Na diagramie nie rysujemy jednak między tymi elementami żadnej relacji.



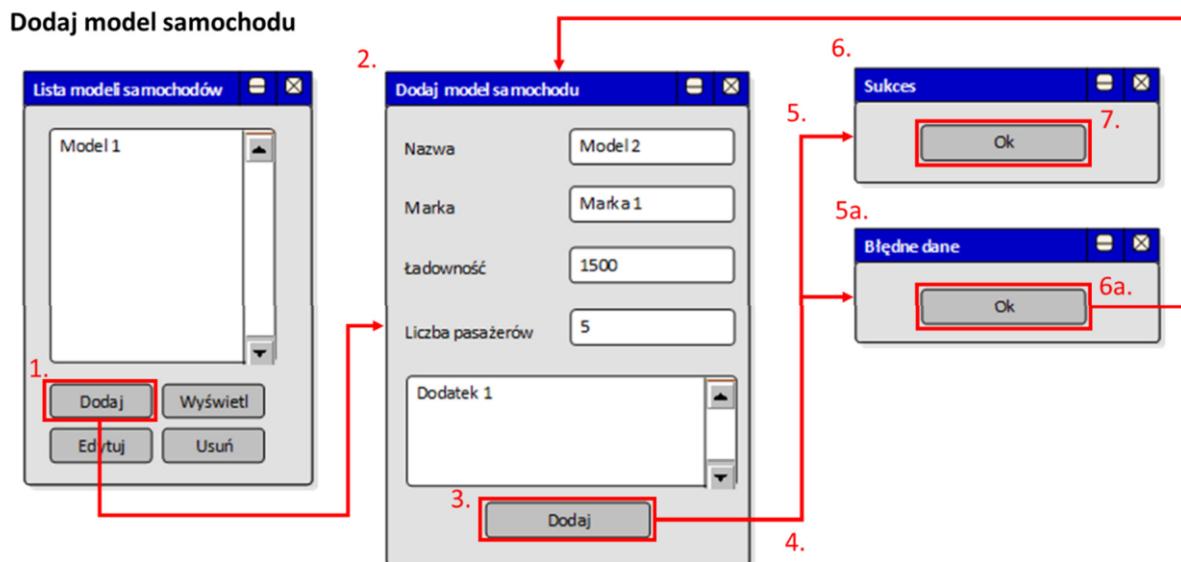
Rysunek 8.13: Przykład słownika na poziomie wymagań oprogramowania z elementami interfejsu użytkownika związanymi z wieloma przypadkami użycia

Utworzenie centralnego dla specyfikacji wymagań słownika dziedziny pozwala oddzielić opis struktury od opisu dynamiki. Takie wyraźne rozróżnienie zdecydowanie ułatwia zachowanie spójności wymagań. Każde pojęcie występujące w opisie wymagań funkcjonalnych i jakościowych posiada jednoznaczną definicję, wspólną dla wszystkich wymagań. Oznacza to, że wszystkie wystąpienia tego pojęcia wskazują na tę jedną definicję. Zauważmy ponadto, że pojęcia ze słownika również mogą być wykorzystywane do opisu innych pojęć. W ten sposób, tworzymy sieć pojęciową danej dziedziny, którą możemy wyrazić za pomocą modelu klas.

Opisy scenariuszy i dokładne modele (słowniki) dziedziny nie są dostarczają kompletnego informacji dla zaprojektowania i zaimplementowania systemu. Wiemy jaka jest logika aplikacji (scenariusze) oraz jakie dane podlegają przetwarzaniu i wizualizacji (model dziedziny), ale nie wiemy, w jaki sposób te dane będą konkretnie wizualizowane. Dlatego też, najczęściej konieczne jest zdefiniowanie wyglądu elementów interfejsu użytkownika. Definicja ta może być wykonana na różnym poziomie szczegółowości. W najprostszym przypadku może to być tzw. **szkielet interfejsu użytkownika** (ang. wireframe), który polega na naszkicowaniu prostymi kreskami układu elementów ekranowych (okienka z zawartymi w nich polami, przyciskami itp.). Jeśli potrzebny jest bardziej dokładna definicja, wykonujemy tzw. **makiety interfejsu użytkownika** (ang. mockup). Najbardziej dokładny jest **prototyp interfejsu użytkownika** (ang. UI prototype), który często wykonywany jest w docelowym narzędziu, w którym będzie implementowany cały system.

Stworzenie szkieletu, makiety lub prototypu zapobiega częstym nieporozumieniom między użytkownikami a wykonawcami systemu. Ułatwiają one zrozumienie zasad realizacji funkcjonalności oraz przedstawienie propozycji interfejsu użytkownika, za pośrednictwem którego będzie następowała interakcja pomiędzy użytkownikami i systemem. Dodatkowo, możemy pokazać pewne schematy nawigacji przez interfejs użytkownika przez zobrazowanie sekwencji pojawiających się widoków ekranowych. Tego typu modele wykorzystują elementy słownika dziedziny i słownika interfejsu użytkownika. Na to nakładane są opisy dynamiki działania systemu (scenariusze) z uwzględnieniem cech jakościowych (np. używalność).

Warto zauważyć, że wybrany stan interfejsu użytkownika stanowi swego rodzaju scenę. Poszczególne kroki scenariuszy przypadków użycia mogą powodować przejście do kolejnej sceny, czyli np. wyświetlenie kolejnego okienka. Takie sekwencje scen możemy nazwać **scenopisami** (ang. storyboard). Wiele narzędzi do projektowania interfejsu użytkownika pozwala na rysowanie scenopisów, a nawet na animację ich wykonania. Przykład scenopisu dla przypadku użycia z rysunku 8.9 przedstawia rysunek 8.14. Numery na diagramie odpowiadają numerom kroków w scenariuszach tego przypadku użycia.



Rysunek 8.14: Przykładowy scenopis scenariuszy przypadku użycia

Rolą scenopisu jest wstępna wizualizacja działania systemu w trakcie wykonywania scenariuszy przypadku użycia. Oprócz kroków scenariusza i projektu wyglądu elementu interfejsu użytkownika, można przedstawić także informacje o pojęciach dziedziny oraz o parametrach konfiguracyjnych. Połączenie opisu dynamiki systemu oprogramowania z jego elementami statycznymi jest ważnym produktem procesu wytwarzczego, który pomaga w zrozumieniu zachowania systemu. Scenopisy są podstawą dyskusji z przyszłymi użytkownikami systemu, których celem jest ustalenie pełnej funkcjonalności systemu, jego cech i ograniczeń, a także wykrycie nowych wymagań i sprecyzowanie już istniejących.

Aby zachować spójność, scenopis powinien być oznaczony nazwą przypadku użycia z opcjonalnie wyróżnionymi krokami scenariuszy. Zasadniczą część scenopisu stanowią „zdjęcia ekranu”, czyli jego sceny, oraz przejścia między nimi. Scena przedstawia stan systemu w kroku scenariusza, który definiuje ważne z punktu widzenia interfejsu użytkownika elementy (tutaj: ekran początkowy i kroki 2, 6 i 5a). Dla interfejsu graficznego są to okna, dla tekstowego interfejsu – stan konsoli, dla interfejsu dźwiękowego – opis dźwięku. Przejście między scenami odpowiada krokom scenariusza, który opisuje operacje wykonane przez aktora i powodujące zmianę sceny (tutaj: kroki 1, 3 i 6a). Przejście zwykle wyzwalane jest przez użycie pewnego elementu sceny (np. naciśnięcie przycisków „Dodaj”) i oznaczone strzałką, prowadzącą z wybranego elementu jednej sceny do kolejnej sceny.

Zadania

Zadanie 1

Mamy zadaną dziedzinę problemu (patrz rozdział 1, sekcja 1.5). Napisz stwierdzenie problemu w postaci jednego złożonego zdania, zgodnie z wybranym wzorcem. Proszę założyć hipotetyczny problem, który mógłby się pojawić dla zadanej dziedziny.

Zadanie 2

Mamy zadaną dziedzinę problemu (patrz rozdział 1, sekcja 1.5). Proszę narysować diagram przypadków użycia opisujący wybraną funkcjonalność systemu budowanego w ramach wybranej dziedziny problemu. Na diagramie proszę umieścić co najmniej 3 aktorów, 6 przypadków użycia, a także co najmniej 3 relacje między przypadkami użycia.

Zadanie 3

Mamy zadaną dziedzinę problemu (patrz rozdział 1, sekcja 1.5). Proszę narysować diagram klas przedstawiający słownik dziedzinowy dla systemu budowanego w ramach wybranej dziedziny problemu. Słownik powinien zawierać co najmniej 6 pojęć-klas, 1 relację generalizacji, 1 relację agregacji oraz 1 relację asocjacji. Dla odpowiednich relacji proszę określić role oraz krotności.

Zadanie 4

Dla zadanego przypadku użycia proszę napisać 2-3 scenariusze w notacji tekstowej podmiot-orzeczenie-dopełnienie. Zachowaj zgodność dopełnień w scenariuszach z modelem słownika dziedziny z zadania 3 – podkreśl dopełnienia występujące w słowniku dziedziny.

Zadanie 5

Narysuj scenopis dla zadanego przypadku użycia. Scenopis powinien być zgodny ze scenariuszami z zadania 4.

Zadanie 6

Proszę zapisać dwa wymagania jakościowe. Dla każdego z wymagań proszę określić rodzaj zgodnie z normą ISO/IEC 25010:2011. Proszę podać opis wymagań wraz z pełną metryką.

Słownik pojęć

Atrybut wymagania

Rodzaj meta-danymi wymagań, który może być używany do powiązania wymagań z danymi umożliwiającymi porządkowanie wymagań oraz zarządzanie projektem budowy oprogramowania.

Historia użytkownika

Opis oczekiwania klienta w stosunku do tworzonego systemu, przedstawiony za pomocą krótkiego zdania mieszącego się np. na kartce z notesu. Zdanie stanowiące historię użytkownika powinno określać podmiot historii (użytkownik, który czegoś chce od systemu), cel biznesowy realizowany przez system i oczekiwany przez użytkownika, oraz motywację klienta stojącą za konkretnym oczekiwaniem.

Interesariusz

Grupa osób, która ma określony, bezpośredni lub pośredni wpływ na proces wdrażania nowego lub aktualizowanego systemu oprogramowania.

Scenariusz

Precyjne określenie sekwencji interakcji użytkownika z systemem, prowadzącej do osiągnięcia określonego celu biznesowego. Scenariusze na poziomie wymagań oprogramowania określają wszystkie zidentyfikowane alternatywne ścieżki wykonania przypadku użycia oraz obsługę sytuacji wyjątkowych.

Scenopis

Sekwencja scen (stanów interfejsu użytkownika) odpowiadająca sekwencji interakcji użytkownika z systemem opisanej scenariuszami.

Co trzeba zapamiętać

Wizja systemu

Wizja systemu oprogramowania jest opisem potrzeb zamawiającego, dotyczącego systemu oprogramowania, przedstawiona na stosunkowo wysokim poziomie ogólności. Specyfikacja na poziomie wizji zbiera w jedną całość opis najważniejszych problemów biznesu oraz wynikających z nich cech produktu software'owego. Wizja systemu powinna dawać odpowiedź na podstawowe pytania: „Po co budujemy nowy system?”, „Jakie problemy naszego biznesu ten system rozwiąże?”, „Dla kogo jest ten system?”, „Jakie cechy ma mieć ten system, aby pokonać zidentyfikowane problemy?”. Na wizję systemu mogą się składać takie elementy jak stwierdzenie problemu, specyfikacja interesariuszy, lista cech systemu oraz słownik pojęć danej dziedziny.

Wymagania użytkownika

Wymagania użytkownika specyfikują wymagania na system w sposób wystarczający do określenia jego rozmiaru i nakładu pracy potrzebnej do jego zbudowania. Specyfikacja wymagań użytkownika powinna dostarczyć odpowiedzi na kluczowe w procesie zarządzania projektem pytania: „Jak obszerna będzie funkcjonalność systemu?”, „Jakie dane będzie przetwarzał system?”, „Kto będzie się posługiwał systemem?”, „Ile system będzie kosztował?”. Proces powstawania specyfikacji wymagań użytkownika możemy nazwać procesem „łowienia wymagań”, które polega na aktywnym odkrywaniu szczegółowych potrzeb zamawiającego przy wykorzystaniu różnych technik. Wymagania funkcjonalne na poziomie wymagań użytkownika formułuje się najczęściej przy wykorzystaniu modelu przypadków użycia lub w formie historii użytkownika. Formułując wymagania jakościowe należy posługiwać się odpowiednimi normami, aby nie pominąć istotnych obszarów jakościowych systemu. Bardzo istotne jest sformułowanie metryk pozwalających zweryfikować spełnienie wymagań jakościowych. Słownik dziedziny korzystnie jest przedstawić w formie graficznej jako model klas.

Wymagania oprogramowania

Wymagania oprogramowania stanowią uszczegółowienie wymagań użytkownika. Na tym poziomie piramidy wymagań, przypadki użycia (ew. historie użytkownika) zdefiniowane podczas formułowania wymagań użytkownika są opisywane w szczegółach, poprzez zdefiniowanie ich scenariuszy. Korzystne jest stosowanie jednolitej notacji dla scenariuszy (np. notacji POD), w tym np. wykorzystanie notacji graficznej (diagramy czynności). W miarę tworzenia scenariuszy uszczegóławiany i uzupełniany jest

słownik dziedziny oraz ustalany jest wygląd interfejsu użytkownika, a także tworzone scenopisy. Kompletna specyfikacja wymagań oprogramowania zawiera również rozbudowaną specyfikację wymagań jakościowych, gdzie nowe lub zaktualizowane wymagania wynikają z uszczegółowienia wymagań funkcjonalnych i słownika. Źródłem dla wymagań oprogramowania jest definicja zakresu systemu w postaci modelu przypadków użycia oraz powiązany z nimi słownik (model) dziedziny.

Rozwiązania zadań

Rozwiązanie zadania 1

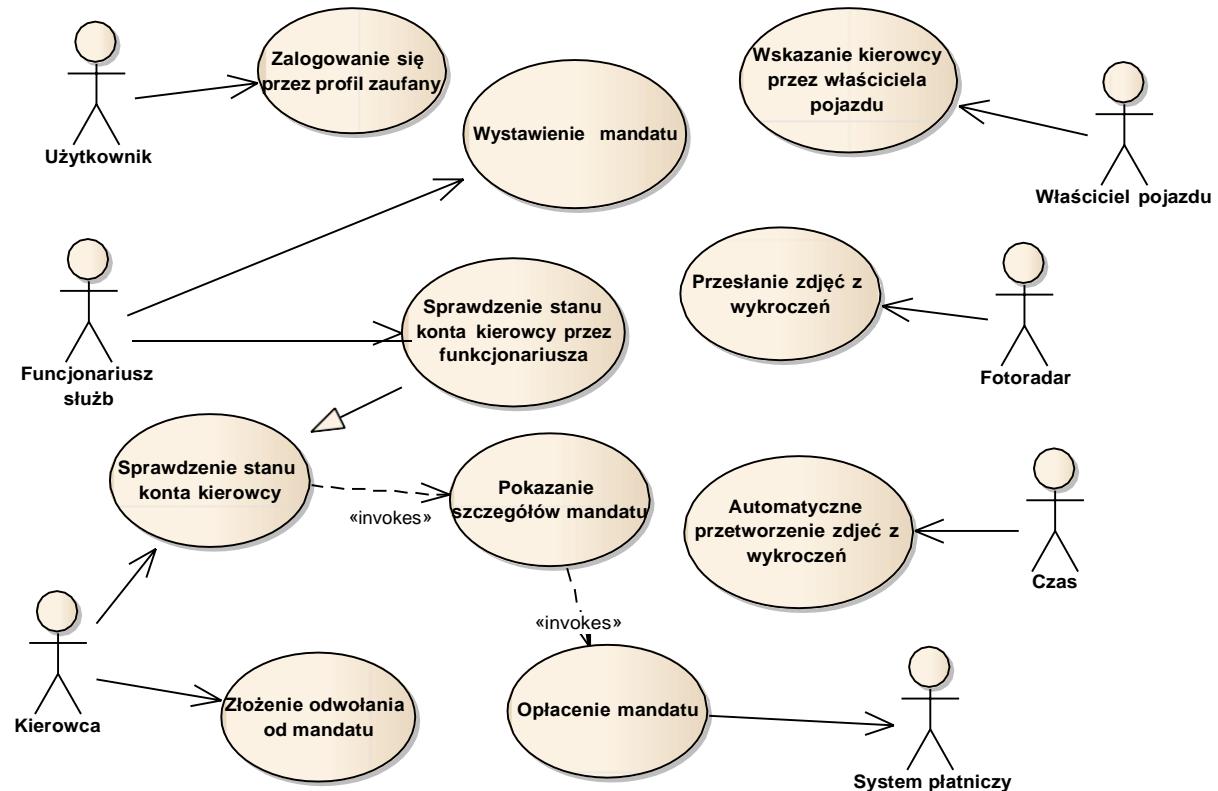
Dziedzina problemu – system dla Wydziału Ruchu Drogowego policji.

Problem związany ze skomplikowanymi procedurami egzekwowania przepisów ruchu drogowego w Polsce, dotyczy kierowców, właścicieli pojazdów i urzędników WRD, wynikiem czego jest niepotrzebnie długi proces egzekwowania mandatów.

Problem można rozwiązać budując system SeZaM, zarządzający obsługą wystawiania i egzekwowania mandatów, dzięki czemu zmniejszymy koszty urzędnicze i oszczędzimy czas wszystkich osób uczestniczących w procesie.

Rozwiązanie zadania 2

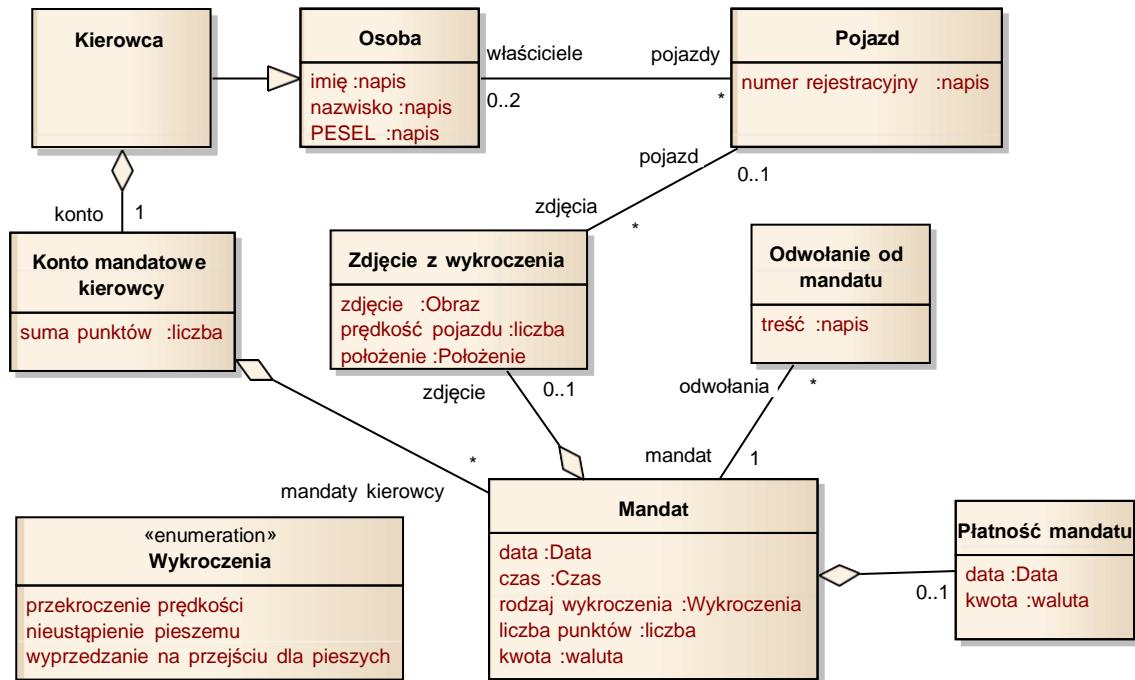
Diagram przedstawiający fragment modelu przypadków użycia dla systemu SeZaM (patrz rozwiązanie zadania 1). Diagram zawiera więcej elementów niż wymaganych w treści zadania.



Aktor „Czas” oznacza uruchamianie przypadków użycia w określonych momentach czasu (tu – np. codziennie o wyznaczonej godzinie). Aktor „Fotoradar” oznacza zewnętrzne w stosunku do systemu urządzenia, komunikujące się z nim poprzez odpowiedni interfejs programistyczny. Diagram zawiera także relację generalizacji – przypadek użycia „Sprawdzenie stanu konta kierowcy przez funkcjonariusza” jest modyfikacją i rozszerzeniem przypadku użycia „Sprawdzenie stanu konta kierowcy”.

Rozwiązań zadania 3

Diagram przedstawiający fragment modelu słownika dziedziny dla systemu SeZaM (patrz rozwiązań zadania 1). Diagram zawiera więcej elementów niż wymaganych w treści zadania.



Rozwiązań zadania 4

Scenariusze dla przypadku użycia „Sprawdzenie stanu konta kierowcy”:

Scenariusz główny

1. Kierowca wybiera opcję "sprawdź konto kierowcy"
 2. System pobiera dane Konta mandatowego kierowcy [kierowca ma aktywne mandaty]
 3. System wyświetla "okno stanu konta mandatowego kierowcy"
 4. Kierowca wybiera opcję "Zamknij"
- Sukces

Scenariusz alternatywny – brak mandatów

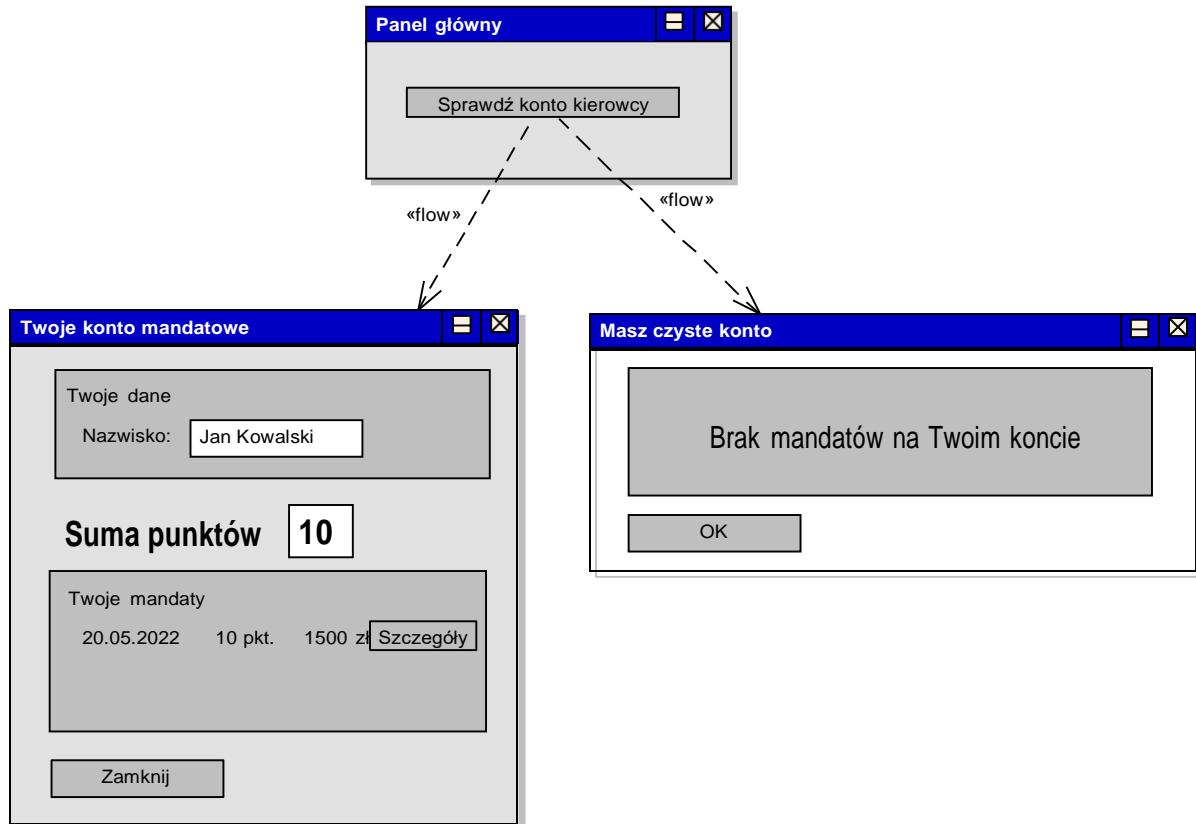
- 1.-2. jak w scenariuszu głównym
 - [kierowca nie ma aktywnych mandatów]
 - 3'. System wyświetla "okno braku mandatów"
 - 4'. Kierowca wybiera opcję "OK"
- Sukces

Scenariusz alternatywny – wywołanie „Pokazanie szczegółów mandatu”

- 1.-3. Jak w scenariuszu głównym
 - 4''. Invoke: "Pokazanie szczegółów mandatu"
- Idź do pkt. 3

Rozwiązywanie zadania 5

Scenariusz dla przypadku użycia „Sprawdzenie stanu konta kierowcy”:



Rozwiązywanie zadania 6

Wymaganie rodzaju „Efektywność wydajnościowa”:

JAK002: System będzie przechowywać do 2.000.000 rekordów użytkowników.

Sposób pomiaru:

Symulacja dodania 2.000.000 użytkowników. Sprawdzenie możliwości zalogowania się przez losowo wybranych 10 000 użytkowników. Sprawdzenie wydajności systemu dla 25 wybranych p.u.

Oczekiwana wartość:

1. Wszyscy wylosowani użytkownicy mogą się zalogować.
2. Wydajność systemu (czas odpowiedzi) jest większy o max. 5% (porównujemy dla 5000 użytkowników i 2.000.000)

Wymaganie rodzaju „Efektywność wydajnościowa”:

JAK003 System powinien dokonać zapamiętania zdjęcia z fotoradaru w czasie nie dłuższym niż 0,5 sek.

Sposób pomiaru:

Dokonujemy próbnego przesłania przez wybranych 50 fotoradarów, każdy po 200 zdjęć. Dokonujemy pomiaru czasu od momentu zainicjowania przez fotoradar transmisji zdjęcia do momentu wysłania przez system do fotoradaru potwierdzenia zarejestrowania zdjęcia.

Oczekiwana wartość:

1. Dla co najmniej 50% zdjęć czas wynosi poniżej 0,5 sek.
2. Dla pozostałych zdjęć czas wynosi do 0,7 sek.

Wymaganie rodzaju „Kompatybilność”

JAK004 System musi dostarczać API do sprawdzania konta mandatowego kierowcy.

Sposób pomiaru:

Przeprowadzenie testów API poprzez wywołanie 1000 razy funkcji odczytu stanu konta mandatowego.

Oczekiwana wartość:

Dla wszystkich wywołań stan konta mandatowego będzie zgodny ze stanem odczytanym przez UI.

9. Wprowadzenie do architektury oprogramowania

9.1. Rola projektowania architektonicznego

Współczesne systemy informatyczne charakteryzują się coraz większym poziomem złożoności. Średniej wielkości system oprogramowania składa się z setek tysięcy lub nawet milionów wierszy kodu. To przekłada się na dziesiątki tysięcy procedur i tysiące modułów (np. klas w programowaniu obiektowym). Gerald M. Weinberg już w 1988 roku napisał: „Programowanie komputerów jest zdecydowanie najbardziej złożonym zadaniem intelektualnym kiedykolwiek wykonywanym przez człowieka. Kiedykolwiek.” (Understanding the professional programmer, Dorset House, 1988). Można z tym stwierdzeniem polemizować, ale trudno jest zaprzeczyć, że budowa oprogramowania jest zadaniem niezmiernie złożonym intelektualnie.

Opanowanie złożoności na poziomie samego kodu jest bardzo trudne lub wręcz niemożliwe. Moduły (klasy), procedury, instrukcje są powiązane bardzo złożoną siecią zależności. Aby system działał efektywnie oraz pozwalał na łatwą jego pielęgnację, powinniśmy stosować dobre praktyki projektowania oprogramowania. Podstawą jest tutaj projekt architektoniczny pokazujący zasadniczą strukturę systemu oraz opisujący sposób działania podstawowych jednostek funkcjonalnych. Plany architektoniczne systemu powinny zapewniać członkom zespołu deweloperskiego dobrą platformę porozumienia – powinny wyrażać najważniejsze decyzje dotyczące sposobu budowy systemu. Niestety, bardzo często oprogramowanie budowane jest bez architektury. Architektura uważana jest za zbędny element burokratyzujący swobodę tworzenia kodu przez programistów. Często uważa się, że kod jest wystarczającą dokumentacją. Brak architektury oprogramowania można jednak porównać do budowy budynku bez planów architektonicznych. W dalszej części tego modułu pokażemy w jaki sposób tworzyć modele architektoniczne, aby stanowiły dobre wsparcie dla zespołów budujących systemy oprogramowania.

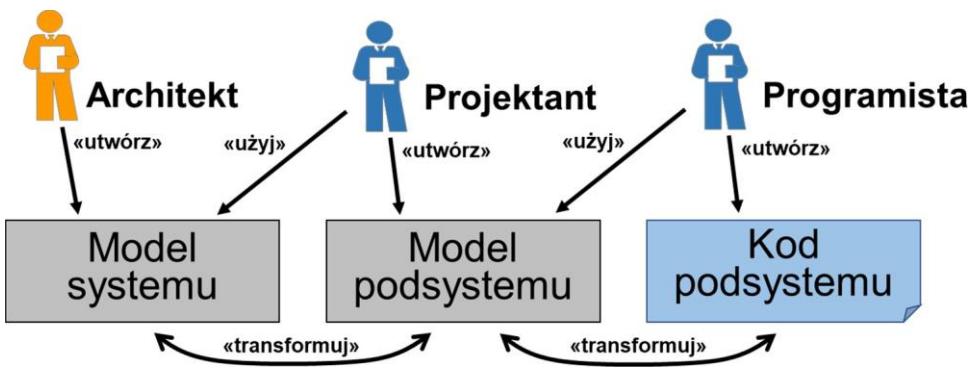
Należy jednak zadać zasadnicze pytanie – czym jest i jak wyraża się architektura systemu oprogramowania? Zasadniczym problemem jest brak powszechnie przyjętego standardu w tym zakresie. Istnieje wiele różnych definicji architektury oprogramowania. Software Engineering Institute Uniwersytetu Carnegie-Mellon podaje ponad 30 definicji dostępnych w literaturze. Spróbujmy jednak określić podstawowe cechy planów architektonicznych.

Architektura oprogramowania stanowi wyrażony zestaw decyzji podjętych przez architekta. Decyzje te są podejmowane na podstawie wymagań klienta oraz wiedzy na temat dostępnych technologii wytwarzania oprogramowania. Plany architektoniczne pokazują strukturę i dynamicę (działanie) budowanego (lub istniejącego) systemu. Biorąc one pod uwagę ograniczenia ekonomiczne i technologiczne, a także możliwość łatwego wprowadzania zmian i rozszerzeń wynikających ze zmian wymagań i zmian technologii. Są one wyrażane w języku graficznym (wizualnym), zrozumiałym dla projektantów i wykonawców systemu (programistów).

W skrócie możemy powiedzieć, że architektura oprogramowania jest ogólnym, wizualnym modelem systemu oprogramowania zapewniającym spełnienie zadanych wymagań. Zwróćmy uwagę na to, że podstawą tworzenia architektury jest stosowanie języka graficznego. Czasami polega to na tworzeniu różnego rodzaju nieformalnych diagramów pisanych w notacji wymyślonej ad-hoc („pudełka i linie”). Zaletą takiego podejścia jest szybkość powstawania modeli oraz brak konieczności nauki konkretnej

notacji. Podstawową wadą jest prowizoryczność oraz możliwość nieporozumień spowodowanych brakiem jednolitej interpretacji diagramów. Rozwiązaniem jest stosowanie ustandaryzowanych języków modelowania. Istnieje grupa takich języków dedykowanych do opisu architektur systemów oprogramowania. Przykładami są języki AADL (Architecture Analysis & Design Language) oraz ArchiMate. Mają one jednak dosyć ograniczone zastosowanie z uwagi na wąski zakres oraz konieczność nauki dosyć specjalizowanego języka. Inne podejście – znacznie szerzej stosowane – polega na wykorzystaniu języka uniwersalnego, jakim jest UML, omawiany w module II. Zaletą stosowania języka UML jest szerokie jego rozpowszechnienie oraz uniwersalność jego zastosowań. W dalszej części niniejszego modułu wykorzystamy podstawowe modele języka UML do przedstawienia sposobu definiowania architektur oprogramowania.

Architekturę oprogramowania tworzymy na stosunkowo wysokim poziomie ogólności. Podobnie jak w przypadku architektury budynków, architekci oprogramowania nie muszą projektować wszystkich szczegółów. Pozostawiają to zadanie deweloperom pełniącym role projektantów. Współpraca między rolami w ramach dyscypliny projektowania jest zilustrowana na rysunku 9.1. Architekt tworzy ogólny **model systemu**, dzieląc jego funkcjonalność na poszczególne komponenty oraz definiując interfejsy i przepływ komunikatów pomiędzy komponentami. Na tym poziomie, komponenty są czarnymi skrzynkami komunikującymi się między sobą poprzez dobrze zdefiniowane i stosunkowo wąskie interfejsy. Architekt projektuje również komunikację między komponentami tworząc odpowiednie diagramy opisujące dynamikę systemu, np. diagramy sekwencji. Projektowanie systemu na takim poziomie szczegółowości powoduje, że architekt musi zapanować nad co najwyżej kilkudziesięcioma komponentami zamiast setkami lub tysiącami klas (jednostek kodu) na raz. Oprócz projektowania elementów architektury logicznej, architekt definiuje również architekturę fizyczną, czyli strukturę składającą się z węzłów wykonawczych. Ponadto, rolą architekta jest zdefiniowanie zasad projektowania poszczególnych podsystemów oraz technologii ich implementacji. Architekt może na przykład zaprojektować część modeli projektowych komponentów, jako wskazówkę dla projektantów.



Rysunek 9.1: Projektowanie na różnych poziomach

Równolegle z architektem, projektanci tworzą szczegółowe **modele podsystemów**. Często jest to bezpośrednio powiązane z tworzeniem **kodu podsystemów**. Warto jednak podkreślić zasadniczą różnicę między projektowaniem i kodowaniem (pisaniem kodu programu). Projektowanie polega na podejmowaniu decyzji odnośnie sposobu implementacji komponentów poprzez podział na jednostki kodu (np. klasy i atrybuty) oraz ogólne określenie ich funkcjonalności (operacje). Kodowanie polega na realizacji projektu w formie konstrukcji programistycznych oraz na zapisaniu kompletnej treści procedur (metod). Ważne jest to, że projektanci korzystają ze specyfikacji komponentów stworzonej przez architekta i zachowują z nią zgodność. Ich zadaniem jest zaprojektowanie „wnętrza” poszczególnych komponentów. Posiadając zdefiniowane interfejsy komponentu i schemat wymiany komunikatów z innymi komponentami, projektant tworzy model klas implementujących założoną funkcjonalność.

komponentu oraz definiuje przepływ komunikatów wewnątrz komponentu. Projektant w trakcie projektowania pojedynczego komponentu, może skupić się na kilku lub kilkunastu klasach implementujących dany komponent. Reszta systemu jest dla niego widziana jedynie poprzez publiczne interfejsy komponentu oraz interfejsy komponentów z których korzysta. Nie musi on znać projektu szczególnego komponentów współpracujących z aktualnie projektowanym.

Na podstawie projektu podsystemu, programista implementuje poszczególne jednostki kodu (klasy) wewnątrz komponentów. Warto zauważyć, że rola programisty często jest pełniona przez osoby pełniące jednocześnie rolę projektanta. W mniejszych projektach również rola architekta jest współdzionego.

9.2. Architektury komponentowe i usługowe

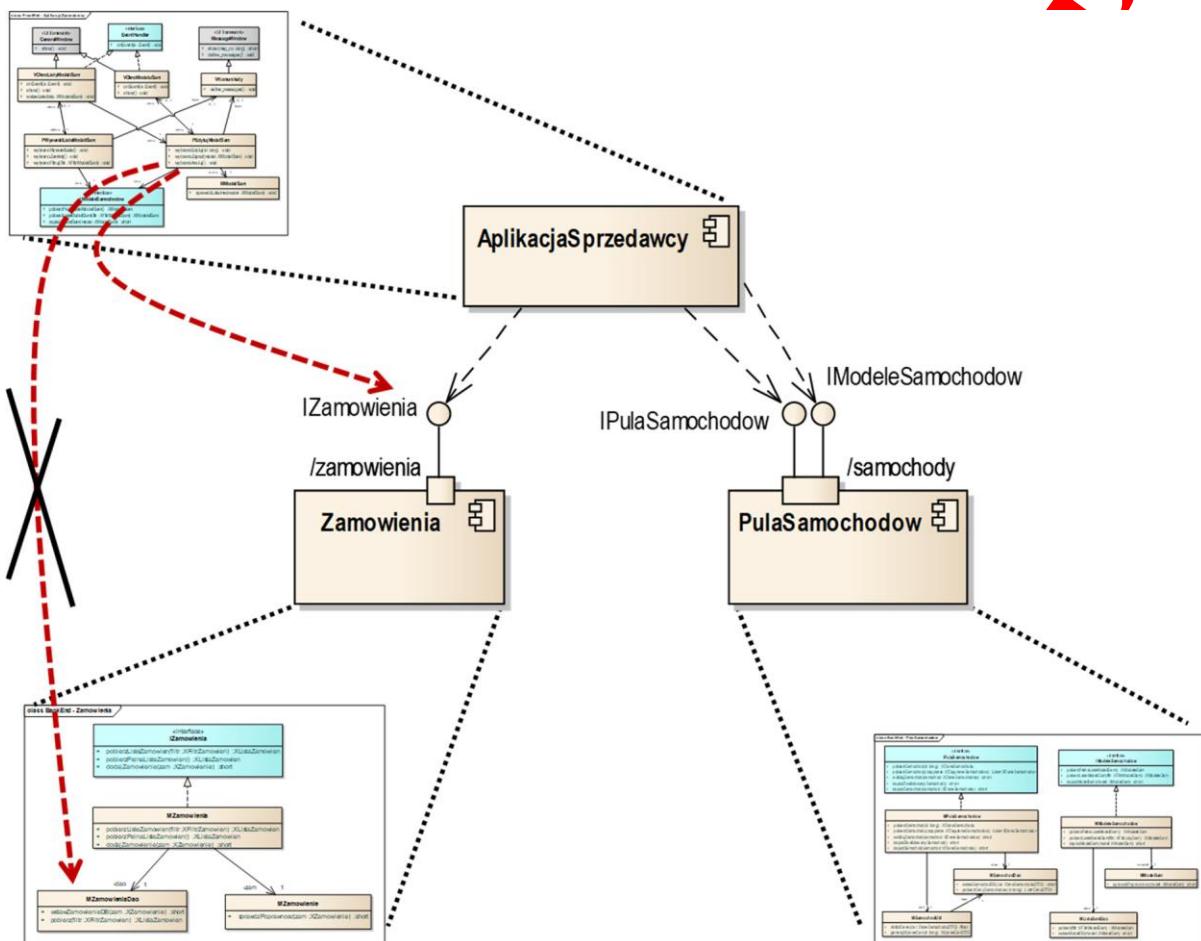
Jedną z najbardziej fundamentalnych decyzji, jaką musi podjąć architekt jest podział systemu na logiczne jednostki funkcjonalne, czyli komponenty. Jednostki te stanowią podstawę dalszych prac nad architekturą. Dzięki takiemu podziałowi, nasz system uzyska bardzo istotny poziom abstrakcji. Zamiast jednego „monolitu” składającego się z tysięcy elementów dostaniemy system składający się na najwyższy poziomie specyfikacji zaledwie kilkunastu lub kilkudziesięciu składników.

Podział na komponenty jest bardzo ważną decyzją architektoniczną. Decyzja ta powinna być podjęta na podstawie dobrze określonych przesłanek. Pierwszą przesłanką jest konieczność zapewnienia, aby komponent realizował dobrze zasadę abstrakcji. Oznacza to, że komponent powinien grupować w sobie elementy realizujące pewien zustytywny podsystem. **Zwartość** komponentów jest bardzo istotną cechą określającą jakość architektury. Określenie „stopnia zwartości” komponentów jest oczywiście rzeczą bardzo trudną. Tak naprawdę, konstrukcja zwartych komponentów jest sztuką wymagającą również sporo doświadczenia. Nasza intuicja wskazuje, że dobry komponent powinien odpowiadać jakiemuś zbiorowi ścisłe związanej ze sobą pojęć (klas) ze środowiska lub elementów wykonawczych (też klas!), realizujących pewne ścisłe związane ze sobą przypadki użycia systemu. Ze zwartością komponentów wiąże się druga przesłanka podziału na komponenty – realizacja zasady zamykania informacji. Oznacza ona, że podsystem realizowany przez komponent powinien być ukryty dla „świata zewnętrznego”. Jednocześnie, komunikacja z pozostałymi komponentami powinna się odbywać przez jak najwęższy styk. W ten sposób, **więzy** między komponentami są stosunkowo słabe. Większość komunikacji odbywa się wewnątrz komponentów. Komunikacja między komponentami odbywa się tylko wtedy, kiedy to jest niezbędne.

Podsumowując, możemy stwierdzić, że **dobra architektura** definiuje podział systemu na komponenty o **wysokiej zwartości** posiadających jednocześnie **słabe więzy** z innymi komponentami. Komponenty komunikują się ze sobą tylko w pewnych kluczowych momentach działania poszczególnych przypadków użycia systemu. Wewnątrz komponentów zawarta jest cała logika przetwarzania danych, a także logika interakcji z użytkownikami i systemami zewnętrznymi. Logika zawarta w danym komponencie jest ukryta dla komponentów pozostałych. Komponenty nawzajem oczekują od siebie jedynie efektów swojej pracy i co więcej – nie mogą od siebie wymagać nic więcej ponad te efekty.

Ilustrację powyższych zasad przedstawia rysunek 9.2. Widzimy tutaj trzy współpracujące ze sobą komponenty. Każdy z tych komponentów ma określona, silnie zwartą strukturę składającą się z kilku lub kilkunastu klas. Jeden z komponentów prosi o „pomoc” dwa pozostałe komponenty. Może to zrobić jedynie „drogą oficjalną” zwracając się za pośrednictwem portu lub interfejsu (patrz: przerywana strzałka). Nie może w żaden sposób „pójść na skróty” (patrz: przekreślona strzałka), czyli żadna klasa wewnątrz komponentu nie może wywoływać bezpośrednio operacji klas wewnątrz innego

komponentu. Tak naprawdę, klasy nie powinny nawet posiadać wiedzy o klasach zawartych w innych komponentach. Komponent, po otrzymaniu „oficjalnej prośby” od innego komponentu, zleca wykonanie zadania swoim klasom. Klasy te następnie wykonują jakąś, często bardzo skomplikowaną, czynność związaną z przetwarzaniem danych. Na koniec, komponent zwraca komponentowi, który go o to prosi, sterowanie wraz z rezultatem przetwarzania oraz ewentualnie zmienia swój stan.



Rysunek 9.2: Przykład współpracy komponentów

Podział systemu na wyraźnie wydzielone komponenty posiada wiele zalet.

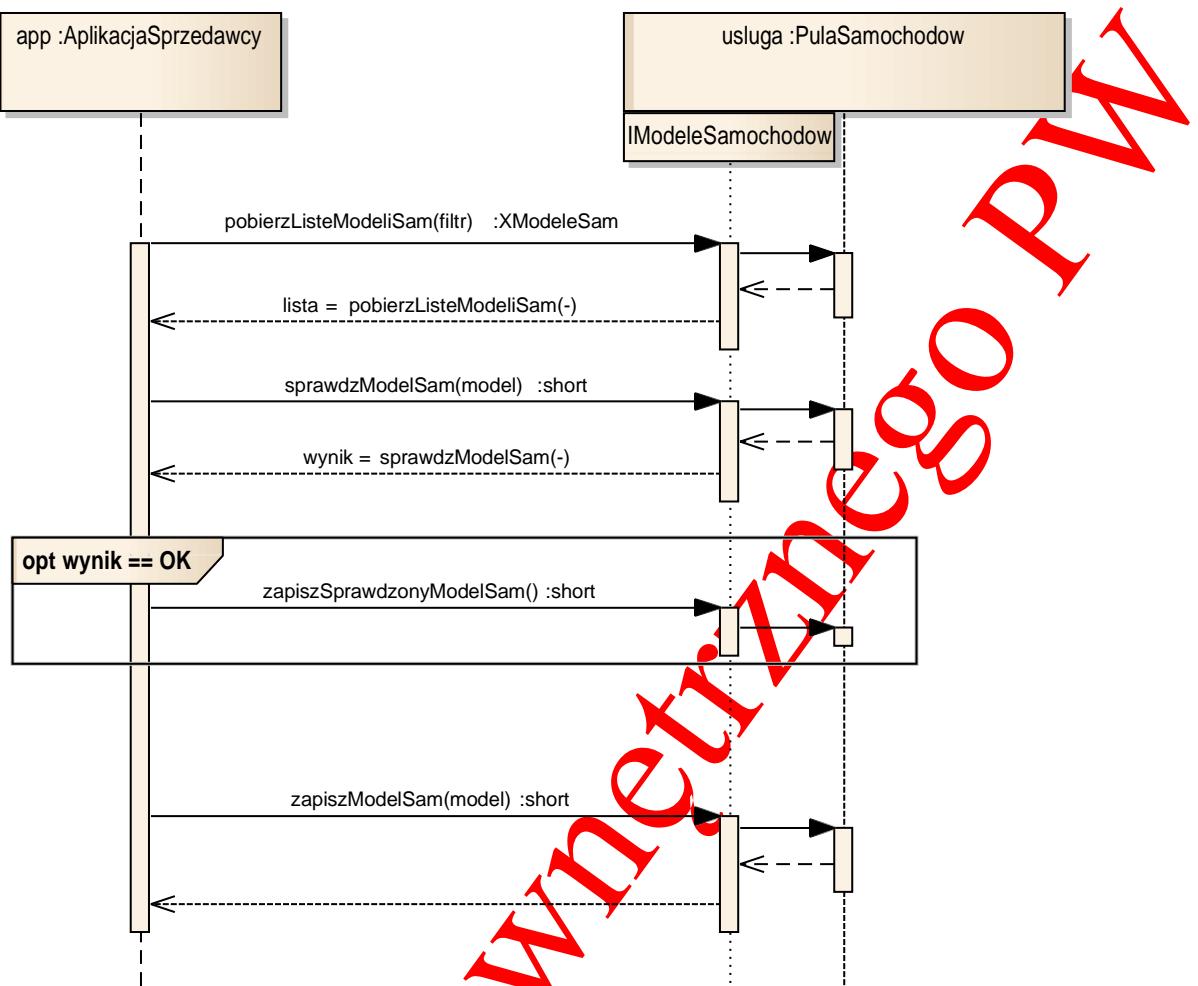
- Poprawia zrozumienie konstrukcji systemu. Realizacja zasad abstrakcji i zamykania informacji umożliwia koncentrację na istotnych aspektach systemu. Osoby zainteresowane szczegółami mogą zajrzać do szczegółowych modeli poszczególnych komponentów.
- Ułatwia pracę grupową, gdyż umożliwia podział pracy. Grupy realizujące swoje komponenty muszą oczywiście dokładnie znać ich strukturę, ale poza tym wystarczy im tylko znajomość specyfikacji powiązań z innymi komponentami.
- Zwiększa elastyczność systemów. Architektura podzielona na dobrze wydzielone komponenty może być łatwo rozszerzana o nowe składniki, a także łatwiej poddaje się zmianom. Ponadto, tego typu architektury wykazują dużą odporność na zmiany konfiguracji sprzętowej systemu – łatwo jest przydzielać komponenty do różnych maszyn fizycznych.
- Zapobiega tzw. kodowi spaghetti (bardzo zagmatwany kod). Znacznie ograniczamy niebezpieczeństwo niejasnych i trudno dostrzegalnych zależności między odległymi fragmentami kodu.
- Ułatwia testowanie ze względu na wyraźnie określone źródła błędów (komponenty).

- Ułatwia **ponowne wykorzystanie kodu**. Dobrze zaprojektowane (spójne i zamkające informację) i dobrze napisane komponenty są doskonałymi jednostkami możliwymi do ponownego użycia. Ponowne wykorzystanie komponentów można porównać do wielokrotnego użycia układów scalonych.

Dobrze zdefiniowane i reużywalne komponenty nazywamy **usługami**. Każda usługa dostarcza zestaw operacji dostępnych poprzez interfejsy. Każdy spójny zestaw operacji stanowi swego rodzaju „kontrakt”, na podstawie którego z usługi mogą korzystać inne komponenty. Taki kontrakt określa strukturę i sekwencje komunikatów odbieranych i wysyłanych przez usługę. W ramach wykonywania swoich operacji, usługa działa w sposób autonomiczny. Oznacza to, że usługa nie zakłada, że jest fragmentem jakiegoś konkretnego systemu. Zamiast tego, usługa jest zorientowana na wykonanie konkretnego zakresu funkcjonalności. Równocześnie, usługa powinna móc pracować w ramach systemów heterogenicznych, czyli złożonych z komponentów wykonanych w różnych technologiiach. Dzięki temu, usługi mogą być instalowane w różnych konfiguracjach i na różnych węzłach wykonawczych. Zdecydowanie zwiększa to elastyczność architektury i łatwość rozwoju systemu zbudowanego z usług. Usługi mogą być dostępne w sieci i wtedy takie usługi nazywamy **usługami webowymi** (ang. web service).

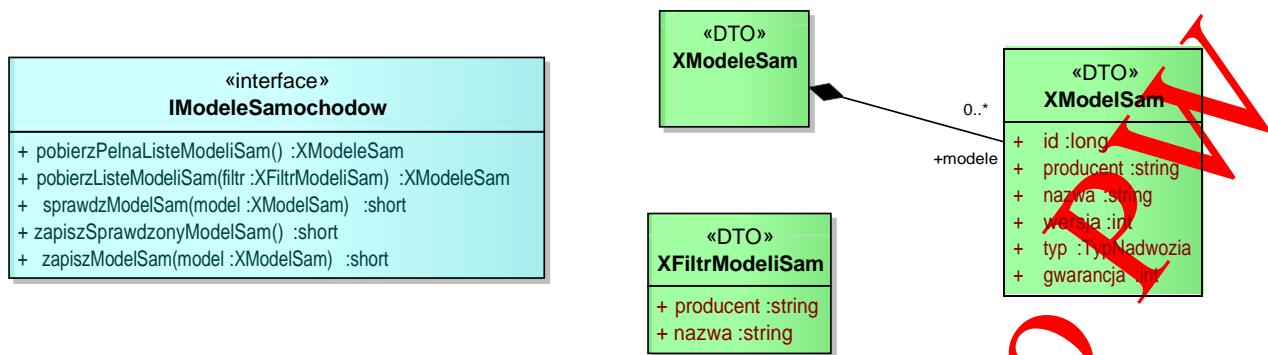
Przykładowa komunikacja w oparciu o usługi przedstawiona jest na rysunku 9.3. Widzimy tutaj sekwencję komunikatów stanowiących kontrakt definiowany przez usługę (komponent) typu „PulaSamochodow” za pośrednictwem interfejsu „IModeleSamochodow”. W ramach tego **kontraktu** możliwe jest dokonanie modyfikacji danych konkretnego modelu samochodu. Aby tego dokonać, konieczne jest najpierw pobranie listy modeli samochodów na podstawie określonego filtra (w szczególności – lista może zawierać jedynie jeden konkretny model). Zakładamy, że model samochodu posiada identyfikator, który komponent klienta usługi może wykorzystać do dalszej komunikacji. Identyfikator ten jest parametrem kolejnego komunikatu, który umożliwia sprawdzenie poprawności danych. Następny komunikat zleca ostateczne zapisanie danych w sposób trwały (np. w bazie danych). Zwróćmy uwagę na to, że możliwe byłoby alternatywne sformułowanie kontraktu. W takim rozwiązaniu konieczny jest tylko jeden komunikat zlecający zapisanie danych modelu samochodu pod warunkiem ich poprawności (patrz ostatni komunikat synchroniczny na rysunku 9.3).

Do użytku Web Services



Rysunek 9.3: Przykładowa sekwencja definiująca kontrakt w ramach usługi

Zwróćmy uwagę na to, że kontrakt powinien być zgodny ze strukturą usługi, to znaczy – opercjami odpowiedniego interfejsu oraz klasami definiującymi **obiekty transferu danych** (ang Data Transfer Object, DTO). Obiekt transferu danych oznacza spójny pakiet danych przesyłany między komponentami w celu wykonania zadanego zadania. Warto tutaj zaznaczyć, że określenie „obiekt” jest nieco mylące, gdyż taki element projektowy (również – programistyczny) nie definiuje konkretnej instancji w ramach działającego systemu, ale de-facto – klasę obiektów. DTO odpowiada pewnemu fragmentowi modelu danych. Może to być odpowiednik pojedynczej klasy (lub jej fragmentu) lub kilku ściśle ze sobą powiązanych klas (np. samochód + silnik). W naszym przykładzie na rysunku 9.4, klasy definiujące obiekty transferu danych wyróżniliśmy poprzez nadanie im przedrostka „X” (w języku angielskim skrót od X-fer = transfer). Alternatywnie, można np. zastosować przyrostek „DTO” oraz oznaczyć klasę odpowiednim stereotypem (np. «DTO»).



Rysunek 9.4: Przykładowy diagram definiujący strukturę usługi

Podejście architektoniczne, które bazuje na usługach nazywamy **architekturą zorientowaną na usługi** (ang. Service Oriented Architecture, SOA). Można również wyróżnić wariant architektury SOA – **architekturę mikro-usługową** (ang. microservice architecture). Wariant ten jest oparty na usługach, które dostarczają niewielkich, bardzo silnie skupionych zestawów operacji. Niezależnie od wariantu, podstawą architektur typu SOA są usługi gotowe. Spełniają one wszystkie założenia architektur komponentowych – minimalizują zależności między komponentami i ukrywają implementację zwartych wewnętrznie komponentów. Dodatkowo, w dużym stopniu oparte są na reużywalności, czyli ponownym wykorzystaniu gotowych komponentów – usług. Zadanie architekta polega na odpowiedniej „orkiestracji” takich usług, czyli połączeniu ich w spójną całość, umożliwiającą realizację zadanych wymagań. W zdecydowanej większości przypadków konieczne jest stworzenie nowych komponentów, które spajają cały system. W razie braku gotowych komponentów usługowych może być też konieczne zbudowanie nowych usług.

Systemy usługowe mogą być projektowane jako heterogeniczne. Oznacza to, że system jest budowany z usług tworzonych w różnych technologiach – na przykład, pisanych w różnych językach czy korzystających z różnych systemów zarządzania bazami danych. Dodatkowo, usługi mogą być zainstalowane na różnych węzłach wykonawczych, korzystających z różnych systemów operacyjnych i komunikujących się za pośrednictwem sieci. Aby takie usługi mogły się ze sobą efektywnie komunikować w sieci, potrzebny jest wspólny protokół wymiany danych. Interfejsy stosujące takie wspólne protokoły nazywamy **interfejsami programowania aplikacji** (ang. Application Programming Interface, API).

Obecnie najczęściej stosowanym stylem dla tworzenia API jest podejście **REST** (ang. REpresentational State Transfer), czyli „zmiana stanu poprzez reprezentację”. Jest ono oparte na idei nawigacji przez zasoby systemu poprzez wybieranie łączy webowych, co skutkuje zmianą stanu i przesłaniu reprezentacji tego stanu w celu jego udostępnienia użytkownikowi. REST korzysta ze standardowego protokołu HTTP, używanego m.in. w komunikacji WWW (ang. World Wide Web). Podstawowe zasady tworzenia systemów z API REST obejmują m.in. rozdzielenie obowiązków, bezstanowość, warstwowość, jednolitość interfejsów. Interfejsy REST są udostępniane w jednolity sposób pod odpowiednim adresem zasobu (tzw. URI, ang. Uniform Resource Identifier), np. „<https://moj.adres.pl/samochody>”. Możemy tutaj wyróżnić adres bazowy oraz tzw. **punkt końcowy** (ang. endpoint). W projekcie komponentowym (np. pokazanym na rysunku 9.2), adres bazowy określa położenie całego komponentu (usługi). Punkt końcowy natomiast określa kanał dostępu do usługi reprezentowany na modelu jako port (np. o nazwie „`/samochody`”).

Interfejsy REST korzystają ze standardowych typów operacji HTTP (GET, PUT, POST, DELETE). Operacje zaprojektowane w ramach danego interfejsu (patrz np. interfejs na rysunku 9.4) stanowią składowe dostępne pod adresem zagnieżdżonym w stosunku do adresu punktu końcowego (np. „`/samochody/sprawdz`”). Wymiana danych następuje albo poprzez wartość (zapisaną w adresie), albo poprzez

pliki (najczęściej w formacie JSON, czasami XML lub RSS). Wadą takiego sposobu komunikacji jest przesyłanie danych w sposób tekstowy, co zdecydowanie zwiększa objętość przesyłanych danych.

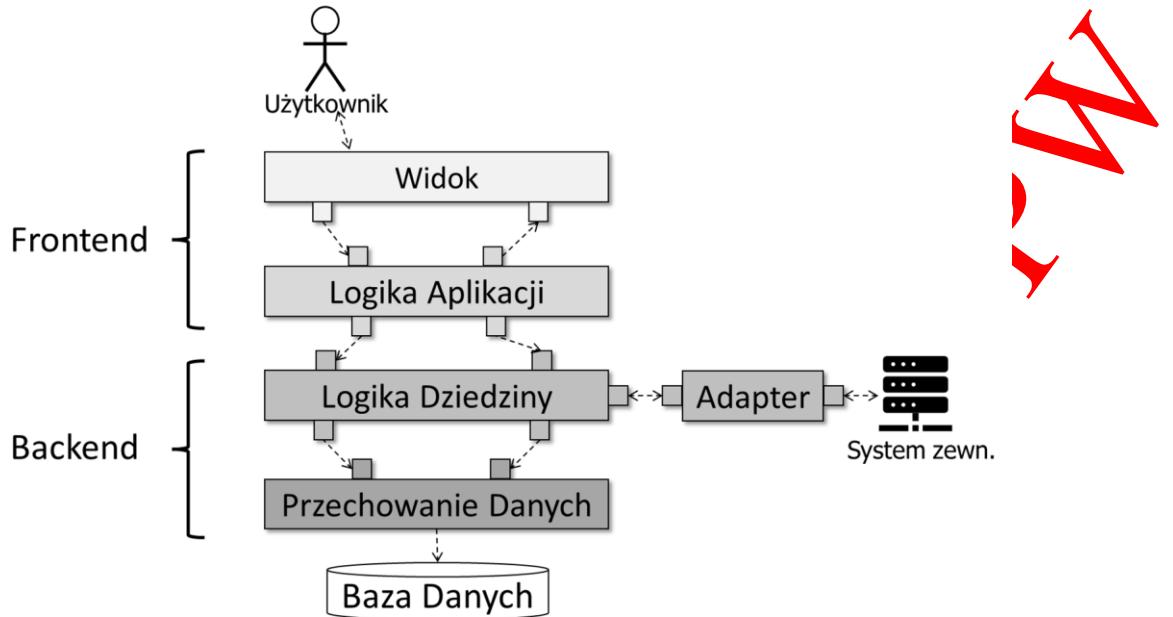
Współcześnie pojawił się szereg nowych technologii, w których dane przesyłane są w sposób binarny – bardziej zwarty i oszczędzający pasmo transmisyjne. Przykładem takiej technologii jest **gRPC**, oparta na zasadzie **zdalnych wywołań procedur** (ang. Remote Procedure Call, RPC). Ideą jest tutaj jak największe podobieństwo wywołania zdalnego do prostego wywołania procedury w programie napisanym w języku strukturalnym lub obiektowym. Wywołania procedur w API definiowane są w jednolitym języku opisu interfejsu (ang. Interface Definition Language, IDL). Taka definicja jest następnie tłumaczona na odpowiedni kod wywołania oraz kod implementacji w wybranym przez programistę języku programowania (np. C#, Go, Java, Python, ...).

9.3. Typowe style architektoniczne

Jak już wspomnieliśmy, podział systemu na komponenty lub usługi jest najważniejszą decyzją podejmowaną przez architekta. Kluczowe w takim podziale jest odpowiednie określenie odpowiedzialności poszczególnych komponentów. Część komponentów powinna odpowiadać za komunikację z użytkownikami, część – za realizację logiki przetwarzania danych, a jeszcze inne – za przechowywanie danych. Te różne poziomy odpowiedzialności komponentów najkorzystniej jest zorganizować w warstwy. Stąd też, najpowszechniej stosowanym podziałem komponentów jest tzw. **architektura warstwowa**. Poszczególne warstwy określają odległość komponentów od środowiska zewnętrznego (użytkowników, systemów zewnętrznych). Istotną cechą systemów w architekturze warstwowej jest uporządkowanie komunikacji między warstwami. Komponenty mogą komunikować się w jedynie ramach danej warstwy lub z warstwą wyżej lub niżej. Nie jest dopuszczalna komunikacja przez wiele warstw.

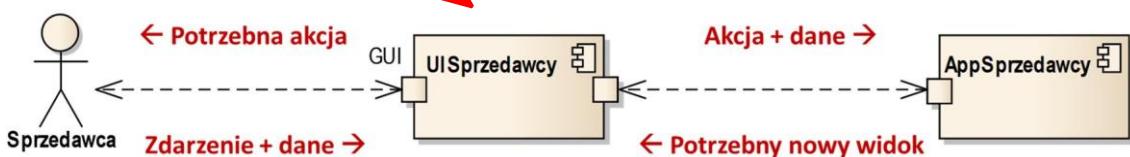
Najprostszym stylem architektonicznym jest podział na dwie warstwy, czasami nazywany **architekturą klient-serwer**. Warstwa górna (klient) odpowiada za interakcję z użytkownikiem, a warstwa dolna (serwer) – za przetwarzanie i przechowywanie danych. Takie podejście można porównać do ogólnej struktury zakładu usługowego. Dla klienta widoczna jest przede wszystkim **witryna** (ang. frontend), a właściwe wykonanie usługi odbywa się na **zapleczu** (ang. backend). Tego typu metafora jest często stosowana przy określaniu warstw w systemach opartych na technologiach webowych. Komponenty zawierające interfejs użytkownika oraz logikę nawigacji między ekranami nazywa się „frontendem”. Komponenty wykonujące zadania przetwarzania danych oraz komponenty przechowujące dane nazywa się „backendem”.

Architektura dwuwarstwowa jest często zbyt ogólna, dlatego też często stosuje się **architekturę wielowarstwową**. Najczęściej wyróżnia się cztery zasadnicze warstwy, które ilustruje rysunek 9.5. Dwie górne warstwy najczęściej stanowią frontend systemu, a dwie dolne – jego backend. Warstwa górna – Widok – odpowiada za komunikację z użytkownikami (ludźmi). Logika całego systemu podzielona jest na dwie warstwy. Warstwa Logiki aplikacji steruje działaniem systemu poprzez realizację funkcjonalności przypadków użycia. Warstwa Logiki dziedziny odpowiada za właściwe przetwarzanie danych oraz zmiany stanu systemu. Poziom logiki dziedzinowej jest również najdogodniejszy do zorganizowania komunikacji z systemami zewnętrznymi (maszynami). Dobrą praktyką jest wydzielenie specjalnych komponentów adaptujących, które pośredniczą w tej komunikacji, dostosowując ją do istniejących interfejsów programistycznych. Najniższą warstwą jest warstwa Przechowywania danych, która umożliwia przechowywanie stanu systemu w sposób trwały. Najczęściej korzysta ona z odpowiedniego systemu zarządzania bazą danych. Poniżej przedstawimy rolę poszczególnych warstw oraz typowy sposób ich wzajemnej komunikacji.



Rysunek 9.5: Schemat architektury warstwowej

Warstwa widoku obsługuje elementy interfejsu użytkownika (formularze, menu, okna multimedialne, itd.) oraz umożliwia wymianę danych z użytkownikiem – wprowadzanie danych przez użytkownika oraz prezentację (wyświetlanie) danych użytkownikowi. Warstwa widoku komunikuje się z jednej strony z użytkownikiem, a z drugiej strony – z warstwą logiki aplikacji. Schemat komunikacji dla warstwy widoku ilustruje przykład na rysunku 9.6. Komponent warstwy widoku (tu: „UISprzedawcy”) reaguje na dwa rodzaje komunikatów. Z jednej strony, otrzymuje on polecenia od komponentu warstwy logiki aplikacji (tu: „AppSprzedawcy”). Dotyczą one potrzeby zaprezentowania (wyświetlenia) nowego widoku (np. okna, formularza) w ramach interfejsu użytkownika. Z drugiej strony, od użytkownika (tu: „Sprzedawca”) docierają komunikaty związane z różnego rodzaju zdarzeniami (np. przyciśnięcie przycisku, wybranie opcji). Takie komunikaty mogą również być związane z wprowadzeniem przez użytkownika odpowiednich danych.



Rysunek 9.6: Schemat komunikacji warstwy widoku

Zdarzenie oraz dane otrzymane od użytkownika są natychmiast przesyłane przez komponent warstwy widoku do komponentu warstwy logiki aplikacji. Odpowiada to realizacji kolejnej akcji (kroku) w scenariuszu interakcji aktora z systemem (np. w scenariuszu przypadku użycia). Z drugiej strony, każdy komunikat otrzymany od warstwy logiki aplikacji przekładany jest na odpowiednie polecenia wyświetlenia nowych elementów interfejsu użytkownika. W rezultacie, do aktora przesyłany jest „komunikat” oznaczający potrzebę podjęcia przez niego odpowiednich akcji. Zwrócić uwagę na to, że ten ostatni komunikat jest komunikatem poniekąd wirtualnym. Użytkownik, z oczywistych względów nie implementuje procedury obsługi takiego komunikatu, a jedynie reaguje na pojawiające się na ekranie nowe elementy interfejsu użytkownika.

W ramach swojej implementacji, komponenty warstwy widoku zawierają odpowiedni kod (np. klasy) odpowiedzialny za rysowanie (ang. render) elementów ekranowych oraz obsługę zdarzeń

pochodzących od użytkownika. Bardzo istotne jest to, że w ramach swojej logiki, kod warstwy widoku odpowiada jedynie za prezentowanie oraz przyjmowanie danych. Częstym błędem jest umieszczanie kodu logiki aplikacji bądź nawet kodu logiki dziedzinowej wewnątrz kodu odpowiedzialnego za interfejs użytkownika. W szczególności, kod elementów ekranowych nie powinien decydować o nawigacji między poszczególnymi widokami. Więcej szczegółów na ten temat oraz typowe rozwiązania projektowe spotykane w warstwie widoku zostaną przedstawione w kolejnym rozdziale niniejszego modułu.

Warstwa logiki aplikacji bezpośrednio odpowiada za realizację scenariuszy interakcji użytkowników z systemem. W tej warstwie zawarta jest implementacja logiki opisanej np. scenariuszami przypadków użycia. Warstwa logiki aplikacji komunikuje się z jednej strony z warstwą widoku, a z drugiej strony – z warstwą logiki dziedzinowej. Schemat komunikacji dla warstwy widoku ilustruje przykład na rysunku 9.7. Z warstwy widoku (tu: „UISprzedawcy”) docierają polecenia wykonania kolejnych akcji (kroków) scenariuszy wraz z danymi wprowadzonymi przez użytkownika. W rezultacie, komponent warstwy logiki aplikacji (tu: „AppSprzedawcy”) podejmuje odpowiednie działania zgodne z odpowiednim krokiem w scenariuszu. Działania mogą być dwojakiego rodzaju. Po pierwsze, komponent może poprosić warstwę logiki dziedzinowej (tu: „Zamówienia”) o wykonanie jakiegoś zadania przetwarzania, aktualizacji lub odczytu stanu danych. Zazwyczaj zadanie jest wykonywane w sposób synchroniczny, czyli na końcu jest przesyłany komunikat zwrotny, zawierający wynik wykonania zadania (np. status lub odczytane dane). Drugi rodzaj działania polega na przesłaniu komunikatu do warstwy widoku z poleceniem wyświetlenia nowego widoku oraz ew. przesłanie danych do wyświetlenia.



Rysunek 9.7: Schemat komunikacji warstwy logiki aplikacji

Zwróćmy uwagę na to, że komunikaty między warstwą logiki aplikacji a warstwą logiki dziedzinowej przepływają w obydwie strony, lecz odpowiednia zależność na rysunku 9.7 jest skierowana w jedną stronę. Wynika to z tego, że komunikacja między tymi warstwami jest zazwyczaj synchroniczna w jednym kierunku. Oznacza to, że tylko warstwa logiki aplikacji przesyła komunikaty z zadaniami do wykonania i to ona oczekuje na ich zakończenie, czyli na zwrotny komunikat wykonania zadania. Warstwa logiki aplikacji „posiada wiedzę” o warstwie logiki dziedzinowej i jest zależna od wykonywanych przez logikę dziedzinową zadań. Z drugiej strony, warstwa logiki dziedzinowej jest niezależna od warstwy wyżej. W szczególności, nie ma ona wiedzy o tym, kto zleca wykonywanie zadań, za które jest odpowiedzialna.

Z definicji działania warstwy logiki aplikacji wynika, że jest ona głównym czynnikiem sterującym całego systemu. Jej kod zawiera klasy decydujące o sekwencji akcji wykonywanych przez inne warstwy zgodnie z zadanimi scenariuszami. O ile kod warstwy widoku „przechwytuje” zdarzenia od użytkownika, to jedynie kod warstwy logiki aplikacji powinien wiedzieć, jak zareagować na te zdarzenia. Reakcja ta jest zależna od przesłanych przez warstwę widoku danych (decyzje użytkownika) oraz od uzyskanych z warstwy logiki dziedzinowej wyników wykonania zadań (stan systemu). Komponenty warstwy logiki aplikacji są swego rodzaju „kierownikami”, instruującymi wszystkie pozostałe komponenty, co mają robić.

Warstwa logiki dziedzinowej (często nazywana warstwą logiki biznesowej) odpowiada za wykonywanie wszystkich zadań związanych z przetwarzaniem oraz zmianą stanu danych w systemie. Warstwa logiki dziedzinowej komunikuje się z jednej strony z warstwą logiki aplikacji, a z drugiej strony – z warstwą logiki przechowywania danych. Schemat komunikacji dla warstwy logiki dziedzinowej ilustruje przykład na rysunku 9.8. Komponent tej warstwy (tu: „Zamówienia”) reaguje na komunikat zlecający

wykonanie określonego zadania. Wykonanie zadania zależy od treści komunikatu oraz zawartych w nim danych. Może to być określone przetwarzanie danych, zmiana stanu systemu lub odczyt stanu systemu. Po wykonaniu zadania, komponent zwraca jego wynik komponentowi, który zadanie zlecił. Wynik może zawierać proste przekazanie sterowania, odpowiedni kod odpowiedzi lub obiekt transferu danych. W przypadku konieczności zmiany stanu lub odczytu stanu systemu, komponent warstwy logiki dziedzinowej przesyła komunikat do warstwy przechowywania danych (tu: „GlownaBazaDanych”). Komunikat taki zawiera odpowiednie zapytanie o dane lub zlecenie aktualizacji danych. W rezultacie, zwrotnie przesyłany jest komunikat z wynikiem zapytania lub rezultatem aktualizacji danych. Wynik ten może być wykorzystany np. do dalszego przetwarzania danych.



Rysunek 9.8: Schemat komunikacji warstwy logiki dziedzinowej

Należy podkreślić, że w dobrze zaprojektowanym systemie, komponenty warstwy logiki dziedzinowej realizują wyłącznie zadania związane z logiką danej dziedziny problemu. Logika ta wynika bezpośrednio ze słownika danej dziedziny oraz odpowiednich reguł dziedzinowych (biznesowych). W tej warstwie tworzymy kod różnego rodzaju algorytmów przetwarzania danych. Obrazowo możemy zauważyć, że o ile logika aplikacji jest „kierownikiem” kierującym wykonywaniem zadań, to logika dziedzinowa stanowi „robotnika” wykonującego różnego rodzaju zadania związane z „obróbką” danych. Należy się jednak wystrzegać sytuacji, w której komponent w warstwie logiki dziedzinowej steruje nawigacją między widokami interfejsu użytkownika. Podobnie, w ramach logiki dziedzinowej nie realizujemy technicznych operacji związanych z dostępem do utrwalonych danych (np. w bazie danych).

Warstwa przechowywania danych jest odpowiedzialna za trwały zapis oraz odczyt utrwalonych danych na odpowiednich nośnikach danych (przy wykorzystaniu odpowiednich systemów zarządzania utrwalaniem danych (np. systemy zarządzania bazami danych)). Warstwa ta komunikuje się z warstwą logiki dziedzinowej, co ilustruje omawiany już rysunek 9.8. Kod w ramach tej warstwy zależny jest od konkretnej technologii przechowywania danych. Może to być relacyjna baza danych, baza danych NoSQL, repozytorium grafowe, system plików. Zapytania otrzymywane z warstwy logiki dziedzinowej dotyczą wykonania operacji na danych typu CRUD (Create / Read / Update / Delete). Są to typowe operacje związane z aktualizacją oraz odczytem danych. Komponenty w warstwie przechowywania danych mają za zadanie przetłumaczyć zapytania z formatu przyjętego w warstwach logiki (np. obiekty transferu danych) na odpowiednie zapytania w formacie przyjętym przez daną technologię przechowywania danych. Na przykład, w technologiach relacyjnych, zapytania są formułowane w języku SQL (Sequence Query Language). O projektowaniu najczęściej stosowanych baz danych relacyjnych powiemy w kolejnym rozdziale niniejszego modułu.

Warstwa przechowywania danych stanowi pewnego rodzaju adapter, dostosowujący logikę systemu do konkretnej technologii przechowywania danych. Podobny adapter jest często konieczny w przypadku integracji budowanego systemu z systemami zewnętrznymi. Odpowiedni schemat komunikacji przedstawia rysunek 9.9. Najodpowiedniejszą warstwą dla dokonania integracji jest warstwa logiki dziedzinowej (tu: „Zamowienia”). W celu zintegrowania z innym systemem projektujemy dodatkowy komponent, który dokona adaptacji formatu danych oraz operacji. Komponent warstwy logiki dziedzinowej wysyła do komponentu adaptera (tu: „AdapterBanku”) odpowiedni komunikat zlecający wykonanie konkretnego zadania. Komunikat ten jest następnie przekazywany przez adapter do systemu zewnętrznego. Przed przekazaniem następuje jednak odpowiednia translacja z formatu przyjętego w

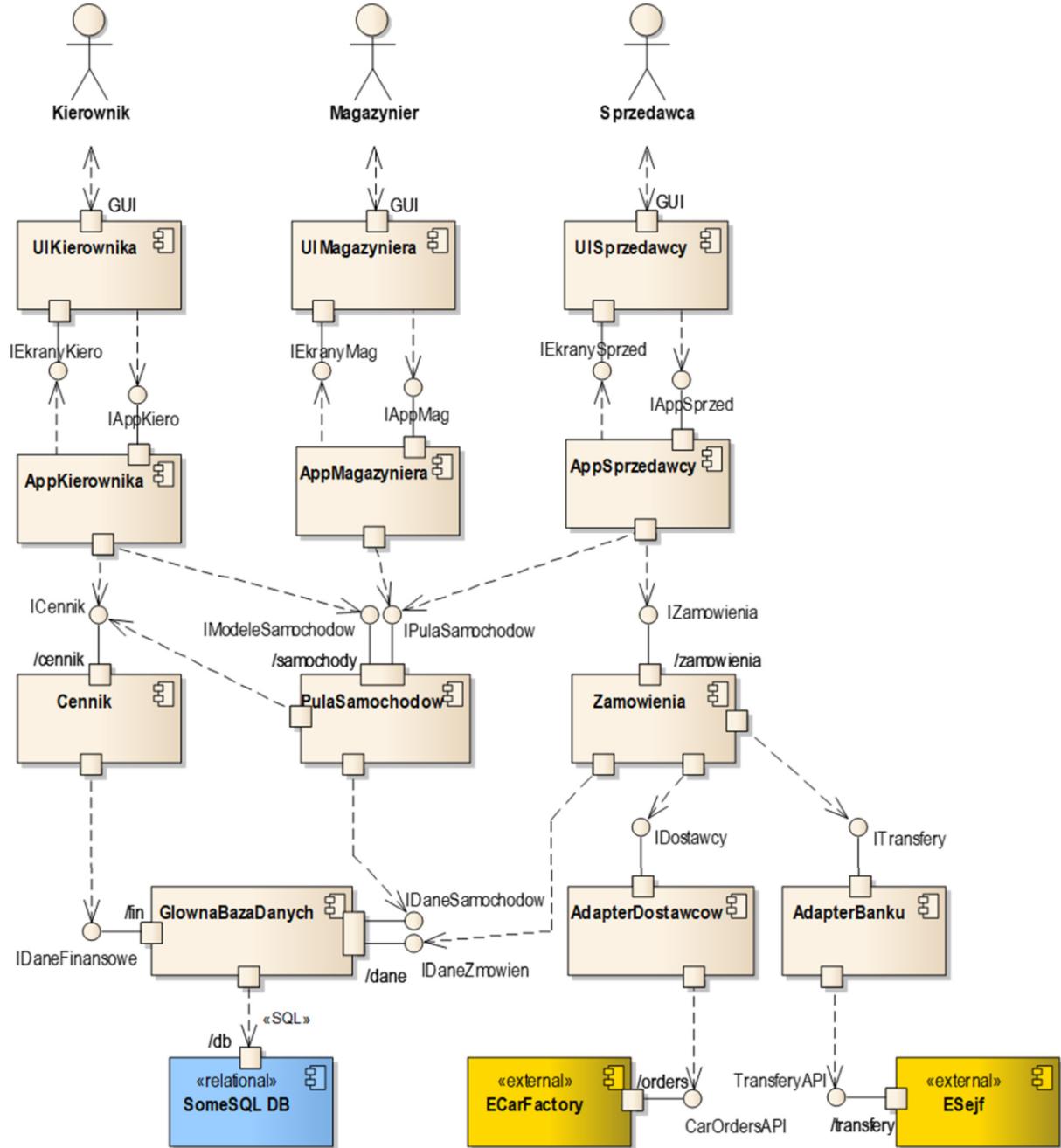
budowanym systemie na format usług (np. API) dostarczanych przez system zewnętrzny. Podobna sytuacja dotyczy komunikatu zwrotnego, zawierającego wynik wykonania zadania. Zauważmy również, że komunikacja może się również odbywać w przeciwnym kierunku. To system zewnętrzny może przesłać komunikat do naszego systemu ze zleceniem wykonania określonego zadania. Oczywiście, warunkiem jest, aby nasz system dostarczał odpowiedniej usługi (zewnętrzne API). W sytuacji, kiedy konieczne jest zintegrowanie większej liczby systemów można zaprojektować osobną warstwę integracyjną. Taka warstwa zapewnia odpowiednią translację różnych formatów danych spotykanych w integrowanych systemach.



Rysunek 9.9: Schemat komunikacji komponentów integracji

Na koniec omawiania architektury warstwowej przedstawimy bardziej rozbudowany przykład. Na rysunku 9.10 widzimy istotny fragment struktury systemu zarządzaną siecią salonów sprzedaży samochodów. Realizuje on (w wybranym zakresie) wymagania przedstawione w poprzednim rozdziale. Jak widzimy, frontend systemu podzielony jest na trzy części zawierające funkcjonalność dla trzech aktorów – użytkowników systemu. Każda część składa się z komponentu widoku (np. „UIKierownika”) oraz komponentu logiki aplikacji (np. „AppKierownika”). Między komponentami frontendu komunikacja następuje poprzez interfejsy dostarczane przez obydwie warstwy. Przykładowo, komponent „UIKierownika” korzysta z odpowiedniego interfejsu komponentu „AppKierownika” i odwrotnie. W ten sposób, możliwe jest asynchroniczne przekazywanie zdarzeń/akcji od warstwy widoku oraz polecień od warstwy logiki aplikacji.

Do użytku wezwano!



Rysunek 9.10: Przykład struktury warstwowej systemu

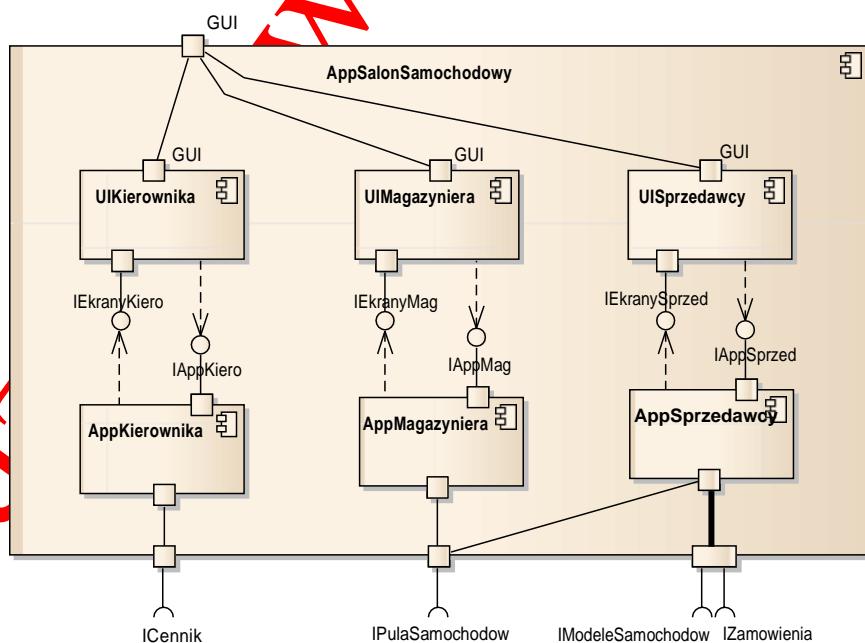
Warstwa logiki dziedzinowej składa się z trzech komponentów („Cennik”, „PulaSamochodow”, „Zamowienia”), które stanowią usługi dla komponentów frontendu. Komponenty logiki dziedzinowej posiadają zestaw interfejsów (np. „ICennik”), za pośrednictwem których realizowane są zadania przetwarzania danych. Interfejsy te dostępne są pod adresami określonymi w nazwach portów (np. „/cennik”). Zwróćmy uwagę na to, że komponenty warstwy logiki dziedzinowej mogą również korzystać ze swoich usług. W naszym przykładzie, komponent „PulaSamochodow” korzysta z interfejsu „ICennik” dostarczanego przez komponent „Cennik”. Takie rozwiązywanie jest stosowane w przypadkach, gdy część operacji logiki dziedzinowej może być wykorzystana w różnych sytuacjach, np. jako fragmenty algorytmów zawartych w innych komponentach.

Warstwa dostępu do danych zawiera tylko jeden komponent („GlownaBazaDanych”). Dostarcza on podstawowych operacji zapisu i odczytu danych z relacyjnej bazy danych (komponent „SomeSQL DB”).

Operacje te podzielone są w sposób logiczny na trzy interfejsy odpowiadające za wymianę danych trzech rodzajów (dane finansowe, dane samochodów oraz dane zamówień). Interfejsy te operują na formatach danych (obiektach transferu danych) wykorzystywanych w pozostałych warstwach systemu (patrz rysunek 9.4). Zwróćmy uwagę na to, że interfejsy są dostępne poprzez dwa porty o różnych adresach lokalnych („/fin” oraz „/dane”). Wewnątrz komponentu następuje konwersja formatu danych oraz uruchamiane są niezbędne zapytania w języku SQL. Zapytania te są obsługiwane przez komponent relacyjnej bazy danych poprzez port o podanym adresie lokalnym („db”).

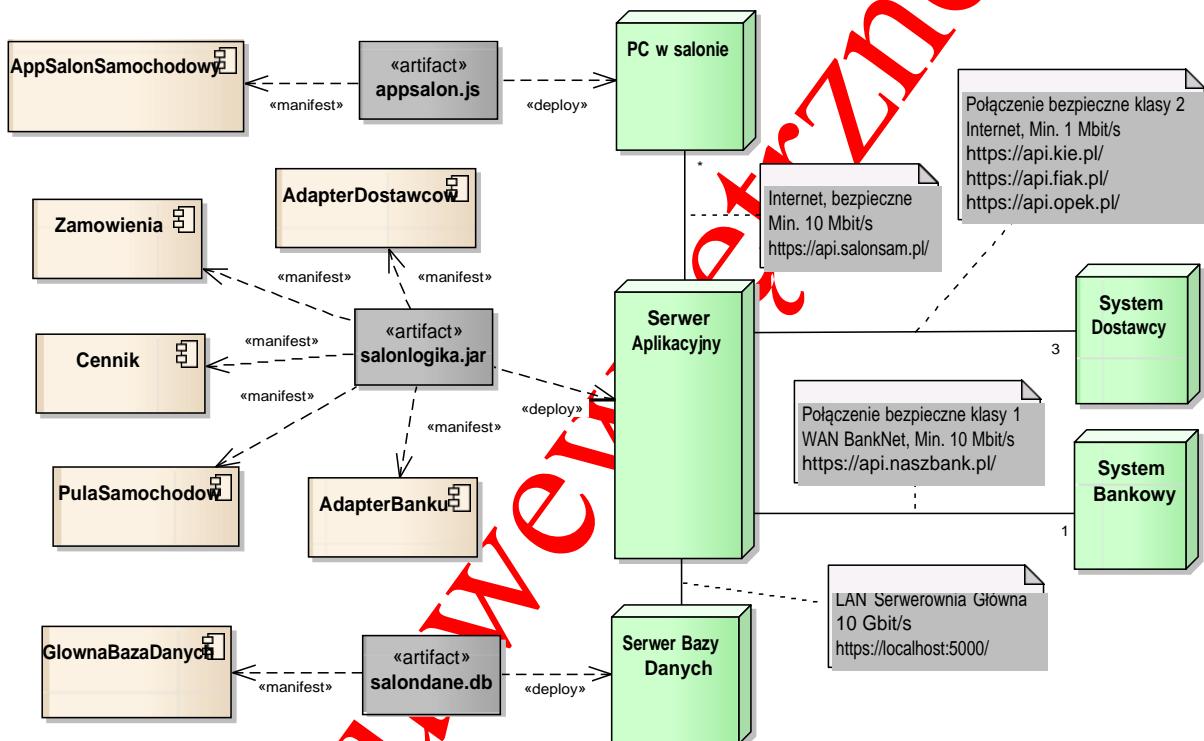
Projekt architektoniczny systemu określa również sposób integracji z systemami zewnętrznymi dwóch rodzajów. W obydwu przypadkach zaprojektowano odpowiedni komponent adaptera (np. „AdapterBanku”). Adapter ten, podobnie jak komponent dostępu do danych, dostarcza interfejsów operujących na formatach danych używanych w innych komponentach systemu. W jego wnętrzu jest dokonywana konwersja tych danych na format używany przez odpowiednie systemy zewnętrzne. Systemy te są na diagramie wyróżnione jako osobne komponenty ze stereotypem «external». Dostarczają one odpowiednich interfejsów programistycznych (np. „TransferyAPI”) dostępnych pod podanymi adresami lokalnymi (np. „/transfery”). Konwersja jest niezbędna, gdyż API systemów zewnętrznym używają własnych formatów danych.

Model architektoniczny przedstawiony na rysunku 9.10 jest dość rozbudowany – zawiera kilkanaście komponentów wraz z portami i interfejsami. Aby ułatwić zrozumienie tej struktury można wprowadzić dodatkowy poziom abstrakcji poprzez zamknięcie kilku komponentów w jednym komponencie złożonym. Taki zabieg ilustruje rysunek 9.11. Widzimy tutaj wszystkie komponenty warstw frontendu zamknięte w jednym, większym komponencie „AppSalonSamochodowy”. Komponent ten aggreuuje wszystkie interfejsy użytkownika (GUI) oraz komunikuje się z komponentami warstwy logiki dziedziny wej poprzez odpowiednie interfejsy, wymagane przez komponenty warstwy logiki aplikacji. Zwróćmy uwagę na to, że struktura architektury logicznej nie uległa zmianie. Dodaliśmy jedynie dodatkowe „pułapki” agregujące kilka mniejszych, lecz nie zmieniające działania interfejsów i portów. To dodatkowe „pułapki” może sygnalizować, że wszystkie składowe komponenty będą zawsze instalowane w jednym miejscu (jako jedna całość).



Rysunek 9.11: Przykład komponentu złożonego

Grupowanie i instalacja komponentów na odpowiednich jednostkach wykonawczych prowadzi nas do zagadnienia projektowania architektury fizycznej. Nasz przykładowy system posiada dosyć złożoną architekturę logiczną, która wymaga zaprojektowania niezbędnych węzłów wykonawczych oraz łączy komunikacyjnych między tymi węzłami. Odpowiedni przykład pokazany jest na rysunku 9.12. Jest to przykład **architektury fizycznej rozproszonej**. Architektura taka zakłada istnienie kilku różnych typów węzłów wykonawczych usługowych (**serwerów**). Poszczególne komponenty logiczne są zainstalowane w różnych węzłach, w ten sposób „rozpraszając” funkcjonalność systemu. W naszym przykładzie widzimy dwa rodzaje serwerów – „Serwer Aplikacyjny” oraz „Serwer Bazy Danych”. Jest to częsty przykład wydzielania osobnej maszyny odpowiedzialnej za przechowywanie danych. Tego typu maszyny są wyposażone w duże przestrzenie dyskowe oraz są optymalizowane pod kątem instalacji baz danych. Serwer aplikacyjny nie wymaga natomiast dużo miejsca na przechowywanie danych, za to powinien posiadać dużą moc obliczeniową, niezbędną do przetwarzania danych. Dodatkowo, moc ta powinna być odpowiednio skalowalna, aby móc obsługiwać zmieniające się obciążenie serwera w różnych okresach.



Rysunek 9.12: Przykład architektury fizycznej

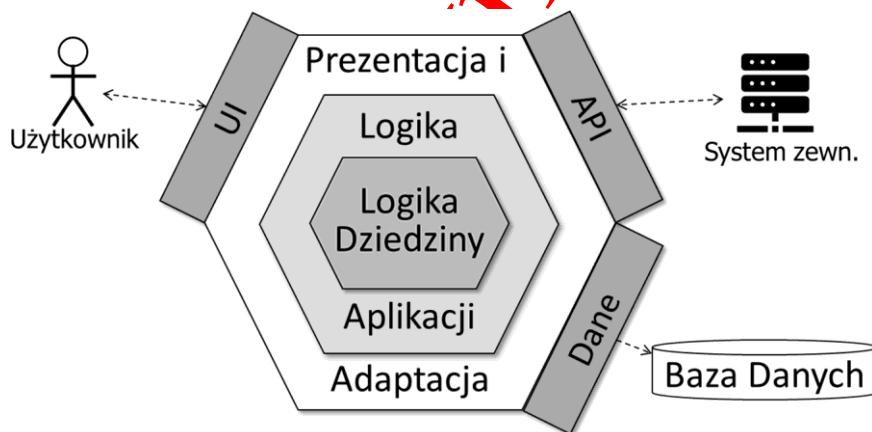
Serwer aplikacyjny współpracuje z wieloma (patrz krotność „*” asocjacji) komputerami typu PC zainstalowanymi z salonach sprzedaży samochodów („PC w salonie”). Komputery te są tzw. **klientami**, czyli węzłami korzystającymi z usług zainstalowanych w innych węzłach (serwerach). Serwer aplikacyjny jest również połączony z trzema (patrz krotność „3”) systemami dostawców oraz jednym (krotność „1”) systemem bankowym. Są to systemy zewnętrzne, zainstalowane „w chmurze”, czyli dostępne przez odpowiednie publiczne adresy sieciowe (URL). Jak widzimy, połączenia między węzłami są odpowiednio zaprojektowane pod kątem ich przepustowości oraz bezpieczeństwa. Na przykład, połączenie z systemem bankowym powinno być zrealizowane jako połączenie o odpowiedniej klasie bezpieczeństwa i szybkości co najmniej 10 Mbit/sekcji.

Projekt na rysunku 9.12 zawiera również decyzje dotyczące sposobu instalacji komponentów. Jak widzimy, cały kod wykonawczy systemu będzie umieszczony w trzech głównych plikach (artefaktach). W pierwszym pliku będzie zawarty (patrz relacja «manifest») cały frontend. Pliki tego typu będą

zainstalowane (relacja «deploy») na maszynach typu PC w poszczególnych salonach sprzedaży. Plik z komponentami logiki dziedzinowej i adapterami będzie zainstalowany na serwerze aplikacyjnym. Trzeci plik będzie zawierał cały system bazy danych wraz z komponentem dostępowym i będzie zainstalowany na serwerze bazy danych.

Zwróćmy uwagę na to, że projekt architektury fizycznej zawiera również informacje o adresach poszczególnych serwerów. Adresy te pozwalają na ustalenie kanałów dostępu do poszczególnych komponentów usługowych. Przykładowo, komponent „Cennik” posiada port definiujący lokalny adres „cennik” (patrz rysunek 9.10). Jest on zawarty w pliku zainstalowanym na komponencie serwera aplikacyjnego. API tego komponentu są dostępne po adresie sieciowym „<https://api.salonsam.pl/>”. Oznacza to, że operacje interfejsu „ICennik” będą dostępne pod adresem „<https://api.salonsam.pl/cennik>”. Jest to pełny adres punktu końcowego (endpoint), który należy wykorzystać w kodzie komponentów logiki aplikacji, korzystających z tego interfejsu.

Architektura warstwowa jest prawdopodobnie najczęściej spotykanym rozwiązaniem projektowym w typowych systemach interaktywnych (komunikujących się z użytkownikami – ludźmi). Istnieją jednak również inne style architektoniczne, w tym takie, które są przeznaczone np. do projektowania systemów wbudowanych (np. system sterowania samochodem), systemów automatyki (np. system sterowania produkcją) lub systemów obliczeń inżynierskich (np. obliczenia rozproszone przy wykorzystaniu superkomputerów). Omówienie wszystkich spotykanych stylów architektonicznych jest poza zakresem niniejszego podręcznika. Tutaj ograniczymy się do omówienia stylu architektury heksagonalnej (ew. cebulowej), zilustrowanego na rysunku 9.13. Jak widzimy, występuje tutaj podobny jak w architekturze warstwowej podział odpowiedzialności.



Rysunek 9.13: Schemat architektury heksagonalnej

Możemy wyróżnić cztery zasadnicze „warstwy cebuli”. Warstwa zewnętrzna dostarcza odpowiednich interfejsów do elementów zewnętrznych. Przy tym, obiektami zewnętrznymi mogą być ludzie, systemy oraz repozytoria danych. W zależności od rodzaju elementu zewnętrznego, stosowana jest odpowiednia technologia (interfejs użytkownika, interfejs programistyczny, połączenie bazodanowe). Kolejna warstwa umożliwia realizację prezentacji i adaptacji danych. W tej warstwie następuje odpowiednie przekształcenie formatu danych, aby był dostępny dla konkretnego obiektu zewnętrznego. Jeśli konieczna jest wymiana danych z ludźmi – stosowany jest odpowiedni projekt kodu interfejsu użytkownika. Jeśli wymiana danych dotyczy systemu zewnętrznego – stosowane są odpowiednie adaptery danych. Dwie warstwy wewnętrzne są najbardziej stabilne i niezależne od stosowanych technologii. Dostarczają one „czystej” logiki systemu, w podziale na logikę aplikacji i logikę dziedzinową. Zasada działania tych warstw jest podobna jak w przypadku architektury warstwowej. Zasadniczą różnicą jest to,

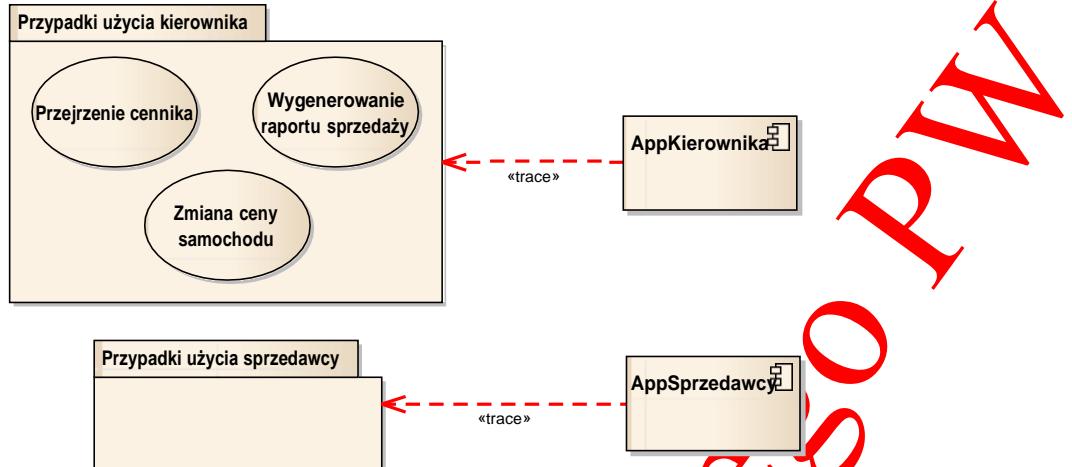
że logika dziedzinowa nie ma bezpośredniego powiązania z warstwą przechowywania danych i adaptacji. Zawsze dzieje się to za pośrednictwem warstwy logiki aplikacji.

9.4. Projektowanie architektury na podstawie wymagań

Z definicji architektury oprogramowania wynika, że projekt architektoniczny powinien przede wszystkim umożliwiać realizację wymagań, jednocześnie uwzględniając kwestie techniczne. Do tej pory omówiliśmy zagadnienia dotyczące aspektów technicznych projektowania architektonicznego (styl architektoniczny, technologie API itp.). Poniżej zajmiemy się zasadami zapewnienia zgodności projektowanej architektury z różnymi rodzajami wymagań, omówionymi w poprzednich rozdziałach niniejszego modułu. Proponowane zasady ujmujemy w postaci kilku podstawowych reguł. Reguły te zakładają określony styl architektoniczny dla docelowych modeli projektowych, a także określony sposób zdefiniowania wymagań funkcjonalnych oraz słownikowych. Nawiązują one do zasad podanych w tym oraz poprzednim rozdziale niniejszego modułu. Dla ułatwienia opisu, reguły oznaczamy identyfikatorami („R1” – „R10”).

Każda zaprezentowana przez nas reguła opisuje pewien krok w przekształceniu modeli wymagań w modele architektoniczne. Po jednej stronie są przedstawione elementy modelu docelowego (np. komponenty czy interfejsy), a po drugiej stronie – elementy modelu źródłowego (np. przypadki użycia czy klasy słownikowe). Bardzo istotną kwestią jest tutaj wyraźne zaznaczenie śladu łączącego projektowane elementy architektoniczne z wymaganiami, które stanowią dla nich źródło. Na diagramach przedstawiających poszczególne reguły zostało to zaznaczone relacjami zależności o stereotypie «trace» (ślad). Relacja ta wskazuje zawsze na element wymagań, na podstawie którego powstał element projektowy. Pozwala ona na jednoznaczną identyfikację elementów projektu wymagających zmian w razie zmiany wymagań.

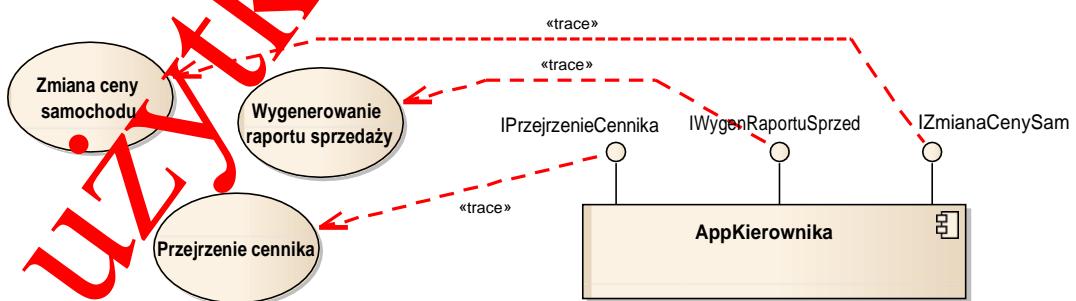
Rysunek 9.14 przedstawia regułę R1. Jest to prosta reguła regulująca tworzenie komponentów warstwy logiki aplikacji. Zgodnie z tą regułą, źródłem przekształcenia są pakiety przypadków użycia. Warunkiem jest zatem odpowiedni podział modelu przypadków użycia. W naszym przykładzie, podział ten przebiega według głównych aktorów. Na przykład, wszystkie przypadki użycia, dla których aktorem jest „Kierownik” zostały umieszczone w pakiecie „Przypadki użycia kierownika”. Na diagramie widzimy wybrane przypadki użycia z tego pakietu umieszczone wewnątrz ikony pakietu. Podobnie wygląda sytuacja dla przypadków użycia, dla których głównym aktorem jest „Sprzedawca”. Dla uproszczenia opisu w tym wypadku pokazana została jedynie ikona pakietu – przypadki użycia są na tym diagramie ukryte.



Rysunek 9.14: Projektowanie komponentów logiki aplikacji (R1)

Zgodnie z regułą R1, dla każdego pakietu przypadków użycia należy utworzyć komponent warstwy logiki aplikacji o nazwie zgodnej z jednolitą konwencją nazewniczą. W naszym przykładzie komponenty takie mają przedrostek „App”, po którym następuje nazwa aktora w odpowiednim przypadku (dotyczy języka polskiego, np. „Kierownika”). Utworzony w ten sposób komponent odpowiada za realizację logiki przypadków użycia, które są zawarte w pakiecie źródłowym. Zwróćmy uwagę na to, że dzięki wyraźnym śladom (relacje «trace») bardzo łatwa jest identyfikacja komponentów wymagających aktualizacji w razie zmiany modelu przypadków użycia.

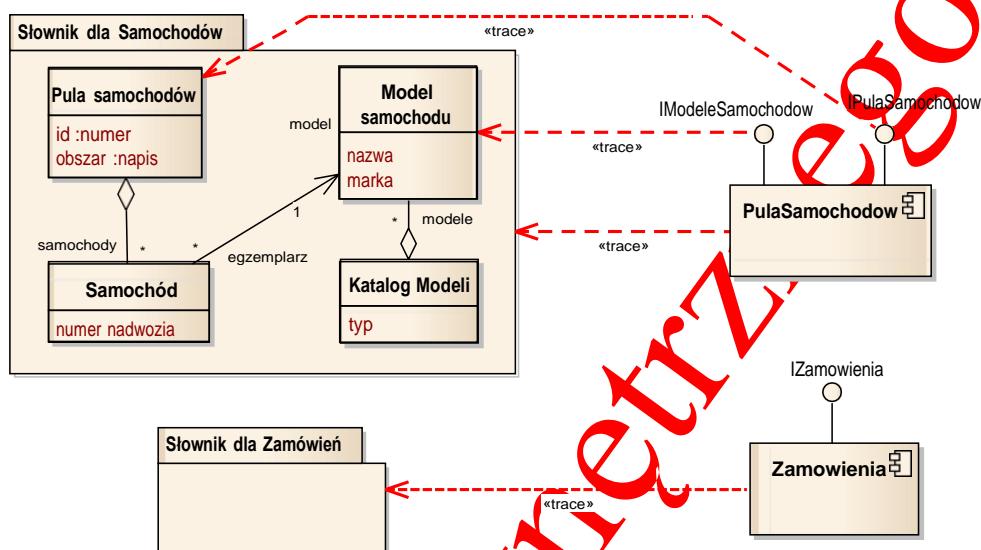
Reguła R2 została zilustrowana na rysunku 9.15. Zgodnie z tą regułą, dla każdego przypadku użycia w modelu wymagań tworzony jest odpowiedni interfejs. Interfejs ten jest przypisany do realizacji temu komponentowi, który wynika z odpowiedniego pakietu wymagań zgodnego z regułą R1. Nazwa interfejsu jest zgodna z jednolitą konwencją nazewniczą. W naszym przykładzie, nazwy te pochodzą bezpośrednio od nazw przypadków użycia z dodanym przedrostkiem „I”. Zauważmy, że reguła jest czysto mnemotechniczna i może być wręcz zastosowana automatycznie. Czasami jednak wskazana jest interwencja architekta. Dotyczy to sytuacji, kiedy model źródłowy składa się z wielu prostych przypadków użycia. W takiej sytuacji, na bazie reguły R2 powstałoby bardzo dużo interfejsów zawierających najwyżej 2-3 operacje. W takiej sytuacji korzystne byłoby połączenie kilku powiązanych ze sobą interfejsów w jeden o większej liczbie (np. około 10) operacji. W skrajnym przypadku, dany komponent logiki aplikacji mógłby dostarczać zaledwie jeden interfejs, jak to pokazano na rysunku 9.10 (patrz np. interfejs „IAppKiero”).



Rysunek 9.15: Projektowanie interfejsów logiki aplikacji (R2)

Reguły projektowania komponentów warstwy dziedzinowej zostały pokazane na rysunku 9.16. Podobnie jak powyżej, mamy tutaj zdefiniowane dwie reguły. **Reguła R3** jest analogiczna do reguły R1, lecz źródłem są pakiety modelu klas słownika. Mówiąc ona, że dla każdego takiego pakietu powinien

zostać utworzony komponent warstwy logiki dziedzinowej. Przyjmuje się, że nazwy tych komponentów pochodzą bezpośrednio z nazw pakietów. Oczywiście, tutaj również konieczne jest odpowiednie podzielenie słownika na pakiety. Reguły takiej pakietyzacji powinny umożliwiać utworzenie zwartych i luźno ze sobą powiązanych komponentów. Na przykład, na rysunku 9.16 widzimy pakiet zawierający słownik pojęć, którego centralnym elementem są samochody i ich zestawy („pule”). W innym pakiecie (jego struktura nie jest pokazana na rysunku) znajdują się pojęcia skupione wokół zamówień. Powiązanie między samochodami a zamówieniami jest stosunkowo wąskie, stąd decyzja dotycząca podziału miedzy dwa różne pakiety wzdłuż takiej granicy.



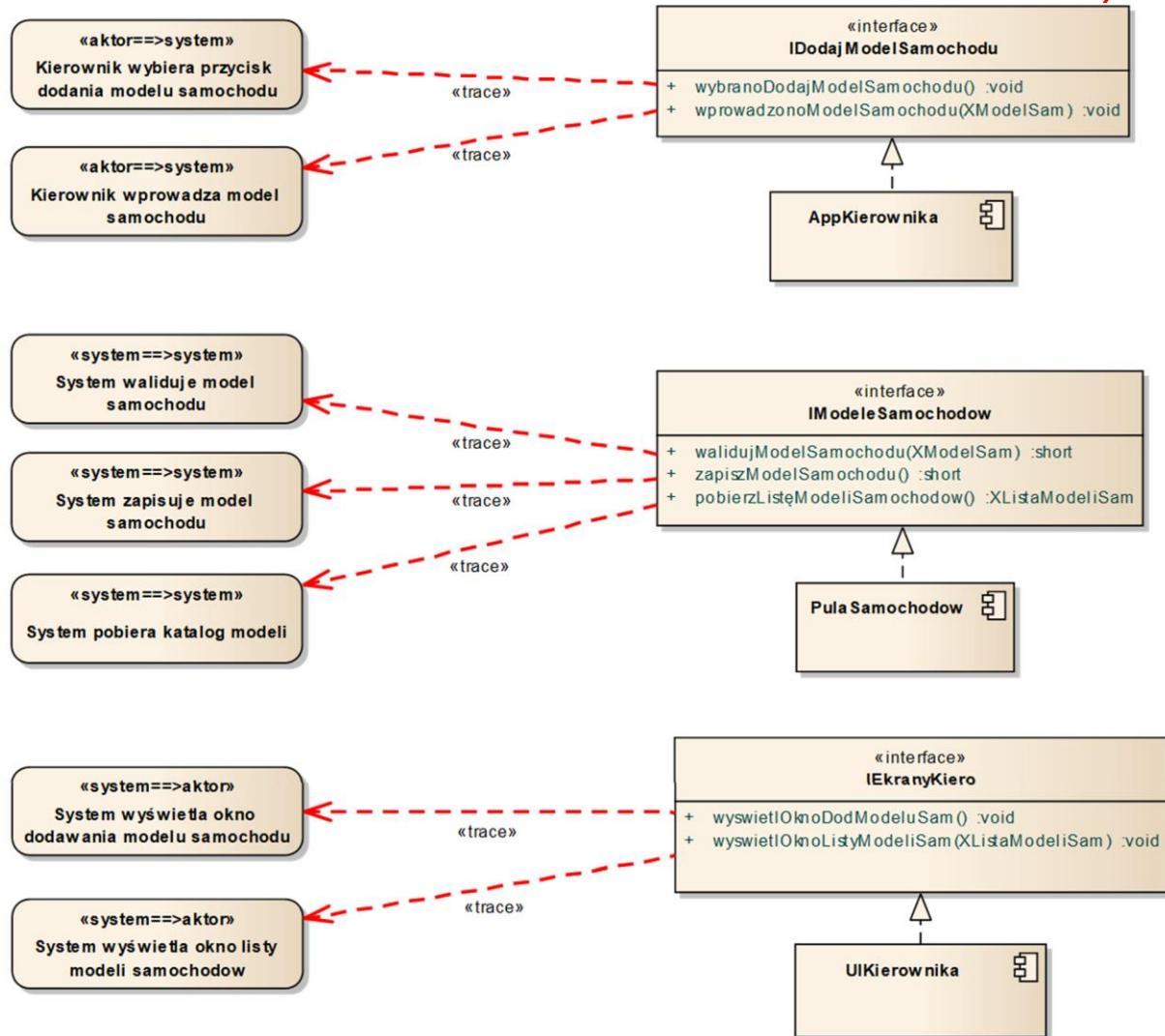
Rysunek 9.16: Projektowanie komponentów i interfejsów logiki dziedzinowej (R3 i R4)

Reguła R4, również zilustrowana na rysunku 9.16 jest analogiczna do reguły R5. Definiuje ona tworzenie interfejsów dostarczanych przez komponenty warstwy logiki dziedzinowej. Elementami źródłowymi są klasy słownikowe. Reguła nie jest regułą mechaniczną i wymaga odpowiedniej selekcji. W najprostszym przypadku, interfejs tworzony jest dla jednej, „centralnej” klasy słownikowej w pakiecie źródłowym. Przykładem jest tutaj interfejs „IZamówienia” powstający na podstawie centralnej klasy „Zamówienie” zawartej w pakiecie słownika dla zamówień. W przypadku bardziej rozbudowanych pakietów słownikowych, korzystne może być utworzenie większej liczby interfejsów. W takiej sytuacji, konieczne jest określenie kilku pojęć centralnych, wokół których będą określone operacje przetwarzania danych. W naszym przykładzie, takim klasami są „Model samochodu” i „Pula samochodów”. Na ich bazie utworzone zostały zatem dwa interfejsy („IModelsSamochodow” oraz „IPulaSamochodow”). Oczywiście, interfejsy są umieszczane w odpowiednich komponentach, utworzonych zgodnie z regułą R3.

Pominiemy tutaj szczegółowe omówienie reguł dotyczących tworzenia warstwy widoku. W najczęstszych przypadkach, komponenty tej warstwy są tworzone wspólnie z komponentami warstwy logiki aplikacji (jeden do jednego). Komponentom tym przypisane są jako ich źródło, odpowiednie pakiety klas słownika interfejsu użytkownika. W przypadku warstwy przechowywania danych częstą praktyką jest utworzenie jednego komponentu zawierającego centralne repozytorium danych. Jego modelem źródłowym jest cały model słownika przekształcony do szczegółowego modelu danych. O tym, w jaki sposób zaprojektować taki komponent mówimy w następnym rozdziale niniejszego modułu.

Kolejnych kilka reguł dotyczy projektowania szczegółów operacji interfejsów. Podstawa tych reguł jest analiza scenariuszy przypadków użycia. W trakcie tej analizy konieczne jest wyróżnienie kilku rodzajów zdań. W zależności od rodzaju zdania, tworzona jest odpowiednia operacja interfejsu, co ilustruje rysunek 9.17. Pierwsza reguła z tej grupy (**reguła R5**) dotyczy zdań typu „aktor do systemu”. Są to zdania,

w których podmiotem jest jakiś aktor, a dopełnienie wskazuje na jakiś element wywołujący zdarzenie (np. przycisk, opcja) lub element słownika dziedziny (np. model samochodu). Na podstawie takich zdan tworzone są operacje interfejsów logiki aplikacji. Nazwy takich operacji tworzymy na podstawie orzeczenia oraz dopełnienia zdania źródłowego. Na przykład (patrz rysunek 9.17), na podstawie zdania „Kierownik wprowadza model samochodu” tworzona jest operacja „wprowadzonoModelSamochodu”. Operacja jest tworzona w interfejsie („IDodajModelSamochodu”), który powstał na podstawie przypadku użycia („Dodaj model samochodu”), w którego scenariuszach zawarte jest zdanie źródłowe.

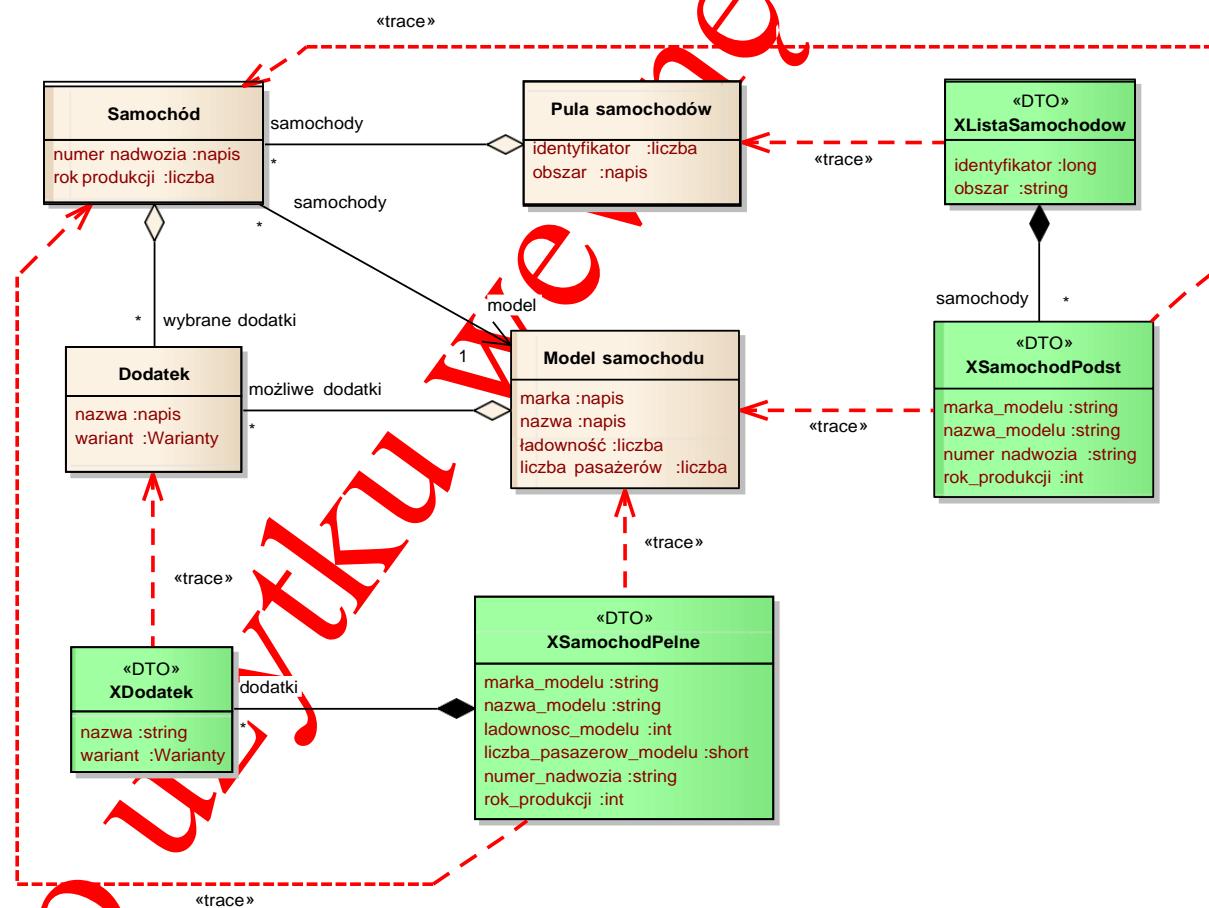


Rysunek 9.17: Projektowanie operacji interfejsów (R5-R7)

Reguła R6 dotyczy zdan typu „system do systemu”. Są to zdania, w których podmiotem jest system, a dopełnieniem jest jakiś element modelu słownika dziedziny. Zdania takie dotyczą jakiejś operacji przetwarzania, aktualizacji lub odczytu danych. Zgodnie z regułą, każde takie zdanie jest przekształcane w operację interfejsu logiki dziedzinowej. Nazwa operacji jest tworzona na podstawie orzeczenia i dopełnienia zdania źródłowego. O umiejscowieniu operacji decyduje dopełnienie zdania. Dopełnienie wskazuje na pewien element modelu słownikowego, na podstawie którego powstał konkretny interfejs. Operacja utworzona według reguły R6 powinna zostać umieszczona w tym interfejsie. Odpowiedni przykład jest przedstawiony rysunku 9.17. Widzimy tu trzy zdania „system do systemu” gdzie dopełnieniem są „model samochodu” oraz „katalog modeli”. Na ich podstawie tworzymy odpowiednie trzy operacje i wstawiamy do interfejsu „IMODELESAMOCHODOW”. Jest to zgodne z regułą R4 i rysunkiem 9.16.

Reguła R7 dotyczy zdań typu „system do aktora”. Są to zdania, w których podmiotem jest system, a dopełnieniem jest jakiś element interfejsu użytkownika (okno, menu, formularz itp.). Zdania takie dotyczą operacji prezentacji odpowiednich widoków użytkownikom systemu. Reguła ta jest podobna do reguł R5 i R6. Dla każdego zdania zgodnego z tą regułą tworzona jest operacja w interfejsie realizowanym przez komponent warstwy widoku. Zasady nazewnicze są podobne jak w poprzednich regułach.

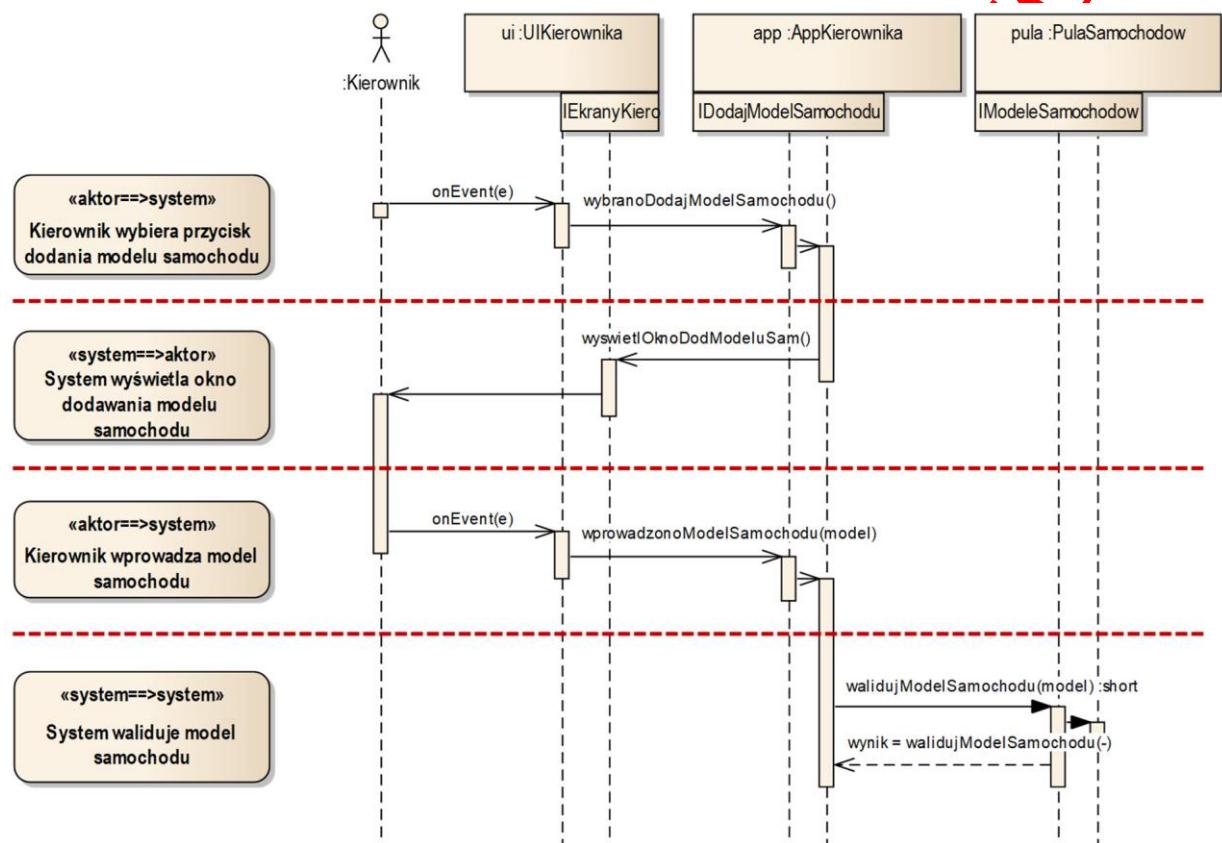
Operacje tworzone regułami R5-R7 mogą posiadać parametry wejściowe oraz mogą zwracać wyniki odpowiednich typów – obiektów transferu danych (DTO). Klasy DTO tworzymy zgodnie z **regułą R8** przedstawioną na rysunku 9.18. Każda taka klasa powinna być oparta na klasie lub klasach modelu słownika. Nazwa klasy DTO pochodzi od klasy bazowej. Jednocześnie, może być potrzebne stworzenie kliku różnych klas DTO na bazie jednej klasy słownikowej. W naszym przykładzie widzimy klasę słownikową „Samochód”, na podstawie której utworzone zostały dwie klasy DTO – „XSamochodPodst” i „XSamochodPelne”. W obydwu przypadkach brane są również pod uwagę atrybuty klasy „Model samochodu”. Klasa „XSamochodPodst” zawiera jedynie dane podstawowe samochodu, potrzebne do wyświetlenia w przypadku pobierania całej listy samochodów jako składnik klasy „XListaSamochodow”. Klasa „XSamochodPelne” zawiera pełne dane samochodu, łącznie z pełnymi danymi modelu danego samochodu. Potrzeba tworzenia różnych klas DTO dla tego samego pojęcia słownikowego wynika z różnych potrzeb dotyczących treści przesyłanych między komponentami danych. Jeśli np. frontend wymaga wyświetlania jedynie danych podstawowych samochodu, możemy w ten sposób ograniczyć transfer danych.



Rysunek 9.18: Projektowanie obiektów transferu danych (R8)

Z reguł R5-R7 wynikają również reguły dotyczące dynamiki systemu. Reguły te możemy wyrazić przy pomocy diagramu sekwencji przedstawionego na rysunku 9.19. Na rysunku wykorzystano strukturę

systemu przedstawioną na rysunku 9.10 oraz definicje interfejsów podane na rysunku 9.17. **Reguła R9** dotyczy zdań typu „aktor do systemu” (na rysunku są pokazane dwa takie zdania). Zgodnie z tą regułą, takie zdanie możemy zamienić na dwa komunikaty. Pierwszy komunikat jest komunikatem asynchronicznym od odpowiedniego aktora (tu: „Kierownik”) do komponentu warstwy widoku (tu: „UIKierownika”). Drugi komunikat, również asynchroniczny jest przesyłany z komponentu warstwy widoku do odpowiedniego komponentu warstwy logiki aplikacji (tu: „AppKierownika”) za pośrednictwem interfejsu (tu: „IDodajModelSamochodu”). W ten sposób projektujemy obsługę zdarzeń pochodzących od użytkownika. Zdarzenie jest „przechwycone” przez komponent warstwy widoku i natychmiast przekazane do odpowiedniego komponentu logiki aplikacji, który w odpowiedni sposób reaguje na zdarzenie.



Rysunek 9.19. Projektowanie dynamiki systemu (R9-R11)

Reguła R10 dotyczy zdań typu „system do aktora”. Zgodnie z tą regułą obsługiwane są akcje podejmowane przez system, dotyczące wyświetlania elementów na ekranie. Każde takie zdanie tłumaczone jest na komunikat asynchroniczny wysyłany przez komponent warstwy logiki aplikacji (tu: „AppKierownika”) do komponentu warstwy widoku, za pośrednictwem odpowiedniego interfejsu (tu: „IEkranKiero”). Dodatkowo, przesyłany jest „komunikat” do aktora, który otrzymuje sterownie i może podjąć odpowiednie dalsze działania.

Ostatnią regułą jest **reguła R11**, dotycząca zdań typu „system do systemu”. Każde takie zdanie jest zamieniane na komunikat synchroniczny od komponentu logiki aplikacji do komponentu logiki dziedziny, za pośrednictwem odpowiedniego interfejsu (tu: „IModeleSamochodow”). Z tym komunikatem powiązany jest odpowiedni komunikat zwrotny, który umożliwia np. przesłanie rezultatu przetwarzania danych.

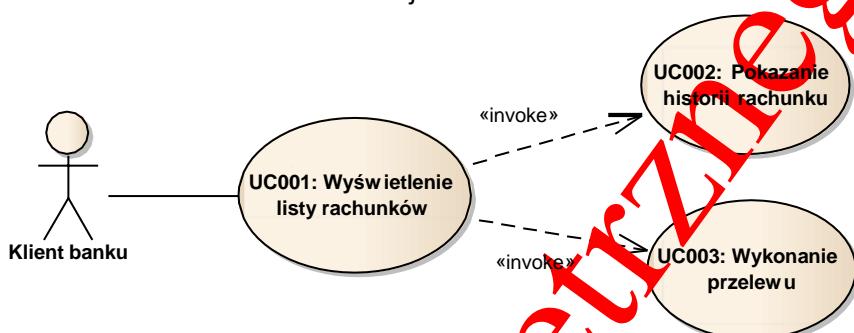
Zwróćmy uwagę na to, że reguły R10 i R11 definiują działanie operacji obsługi zdarzeń przez komponenty warstwy logiki aplikacji, utworzonych na podstawie reguły R5. Zauważmy również, że diagram

na rysunku 9.19 pokazuje połączenie wyniku działania reguł R9-R11 w jedną całość. W rezultacie powstał diagram opisujący realizację pewnego fragmentu scenariusza przypadku użycia w zaprojektowanej strukturze komponentowej systemu.

Zadania

Zadanie 1

Zadany jest model wymagań (przypadeków użycia) systemu przedstawiony na rysunku. Proszę zaprojektować architekturę logiczną systemu: narysować diagram komponentów (komponenty, interfejsy, porty, relacje) w architekturze czterowarstwowej.



Zadanie 2

Dla architektury logicznej z rozwiązań zadania 1 proszę zaprojektować architekturę fizyczną. Proszę narysować diagram montażu, zawierający co najmniej 3 węzły oraz odpowiednie artefakty w relacjach „deploy” oraz „manifest”.

Zadanie 3

Proszę zaprojektować interfejsy dla architektury logicznej z rozwiązań zadania 1. Proszę narysować diagram zawierający definicję dwóch interfejsów (pełne operacje z sygnaturami) wybranych z modelu komponentów. Jeden interfejs powinien być dostarczany przez komponent warstwy logiki aplikacji, a drugi przez komponent warstwy logiki dziedzinowej. Dla każdego interfejsu proszę zaproponować po dwie operacje – proszę założyć odpowiednią funkcjonalność (scenariusze) przypadków użycia z zadania 1.

Zadanie 4

Proszę narysować diagram sekwencji dla realizacji przypadku użycia „Pokazanie historii rachunku” (patrz zadanie 1). Diagram powinien być zgodny z modelem komponentów z rozwiązań zadania 2 oraz definicja interfejsów z rozwiązań zadania 3. Proszę założyć odpowiedni scenariusz główny tego przypadku użycia, a diagram narysować tylko dla realizacji tego scenariusza.

Słownik pojęć

Architektura mikro-usługowa

Wariant architektury zorientowanej na usługi, w którym projektuje się usługi dostarczające niewielkich, bardzo silnie skupionych zestawów operacji.

Architektura oprogramowania

Wyrażony zestaw decyzji podjętych przez architekta. Decyzje te są podejmowane na podstawie wymagań klienta oraz wiedzy na temat dostępnych technologii wytwarzania oprogramowania. Plany architektoniczne pokazują strukturę i dynamikę (działanie) budowanego (lub istniejącego) systemu. Biorą one pod uwagę ograniczenia ekonomiczne i technologiczne, a także możliwość łatwego wprowadzania zmian i rozszerzeń wynikających ze zmian wymagań i zmian technologii. Są one wyrażane w języku graficznym (wizualnym), zrozumiałym dla projektantów i wykonawców systemu (programistów).

Architektura warstwowa

Styl architektoniczny oparty na podziale komponentów na warstwy, określające odległość komponentów od środowiska zewnętrznego. Istotną cechą systemów w architekturze warstwowej jest uporządkowanie komunikacji między warstwami. Komponenty mogą komunikować się w jedynie ramach danej warstwy lub z warstwą wyżej lub niżej.

Architektura zorientowana na usługi

Podejście do architektury oprogramowania, które bazuje na usługach.

Interfejs programowania aplikacji (API)

Typ interfejsu w aplikacjach sieciowych, który jest oparty na ujednoliconym protokole wymiany danych poprzez sieć.

Logika aplikacji

Warstwa systemu oprogramowania, która bezpośrednio odpowiada za realizację scenariuszy interakcji użytkowników z systemem.

Logika dziedzinowa (biznesowa)

Warstwa systemu oprogramowania, która odpowiada za wykonywanie wszystkich zadań związanych z przetwarzaniem oraz zmianą stanu danych w systemie.

Model systemu

Model, który opisuje podział funkcjonalności systemu na poszczególne komponenty oraz definiuje interfejsy i przepływy komunikatów pomiędzy komponentami. Model systemu może być uszczegółowiony poprzez określenie modeli podsystemów (komponentów).

Usługa

Dobrze zdefiniowany i reużywalny komponent oprogramowania, który dostarcza zestaw operacji dostępnych poprzez jego interfejsy. Zestaw operacji usługi stanowi swego rodzaju „kontrakt”, na podstawie którego z usługi mogą korzystać inne komponenty.

REST

Podejście do tworzenia interfejsów programowania aplikacji (API), polegające na zmianie stanu po przez reprezentację (ang. REpresentational State Transfer). Podejście to oparte jest na idei nawigacji przez zasoby systemu poprzez wybieranie łączy webowych, co skutkuje zmianą stanu i przesłaniu re prezentacji tego stanu w celu jego udostępnienia użytkownikowi.

Usługa webowa

Usługa dostępna za pośrednictwem sieci, dostarczająca interfejsy programistyczne oparte na technologiiach webowych.

Zdalne wywołania procedur

Podejście do tworzenia interfejsów programowania aplikacji (API), którego główną ideą jest jak największe podobieństwo wywołania zdalnego do prostego wywołania procedury w programie napisanym w języku strukturalnym lub obiektowym.

Co trzeba zapamiętać

Rola architektury oprogramowania

Podstawowym składnikiem dobrych praktyk inżynierii oprogramowania jest stworzenie projektu architektonicznego pokazującego zasadniczą strukturę systemu oraz opisującego sposób działania podstawowych jednostek funkcjonalnych. Plany architektoniczne systemu powinny zapewniać członkom zespołu deweloperskiego dobrą platformę porozumienia – powinny wyrażać najważniejsze decyzje dotyczące sposobu budowy systemu. Architekturę oprogramowania tworzymy na stosunkowo wysokim poziomie ogólności. Architekt tworzy ogólny model systemu, dzieląc jego funkcjonalność na poszczególne komponenty oraz definiując interfejsy i przepływy komunikatów pomiędzy komponentami.

Architektury komponentowe i usługowe

Kluczową decyzją architektoniczną jest podział systemu na komponenty. **Dobra architektura** definiuje podział systemu na komponenty o **wysokiej zwartości** posiadających jednocześnie **słabe więzy** z innymi komponentami. Komponenty komunikują się ze sobą tylko w pewnych kluczowych momentach działania poszczególnych przypadków użycia systemu. Wewnątrz komponentów zawarta jest cała logika przetwarzania danych, a także logika interakcji z użytkownikami i systemami zewnętrznymi. Logika zawarta w danym komponencie jest ukryta dla komponentów pozostałych. Komponenty nawzajem oczekują od siebie jedynie efektów swojej pracy i co więcej – nie mogą od siebie wymagać nic więcej ponad te efekty. Dobrze zdefiniowane i reużywalne komponenty nazywamy usługami. Usługi mogą być dostępne w sieci i wtedy takie usługi nazywamy usługami webowymi.

Style architektoniczne

Dobrze zaprojektowana architektura oprogramowania przestrzega określonego stylu architektonicznego, który definiuje sposób podziału komponentów. Najpowszechniej stosowanym podziałem komponentów jest architektura warstwowa. Najprostszym stylem architektonicznym jest podział na dwie warstwy, czasami nazywany architekturą klient-serwer. Warstwa górna stanowi witrynę (frontend) i odpowiada za interakcję z użytkownikiem, a warstwa dolna stanowi zaplecze (backend) i odpowiada za przetwarzanie i przechowywanie danych. Często stosuje się także architekturę wielowarstwową, przy czym najczęściej wyróżnia się cztery zasadnicze warstwy: widok, logikę aplikacji, logikę

dziedzinową i przechowywanie danych. Komponenty poszczególnych warstw należy zainstalować na odpowiednich jednostkach wykonawczych, co jest projektowane w ramach architektury fizycznej. W złożonych systemach projektuje się architekturę fizyczną rozproszoną, która zakłada istnienie kilku różnych typów węzłów wykonawczych usługowych (serwerów).

Architektura na podstawie wymagań

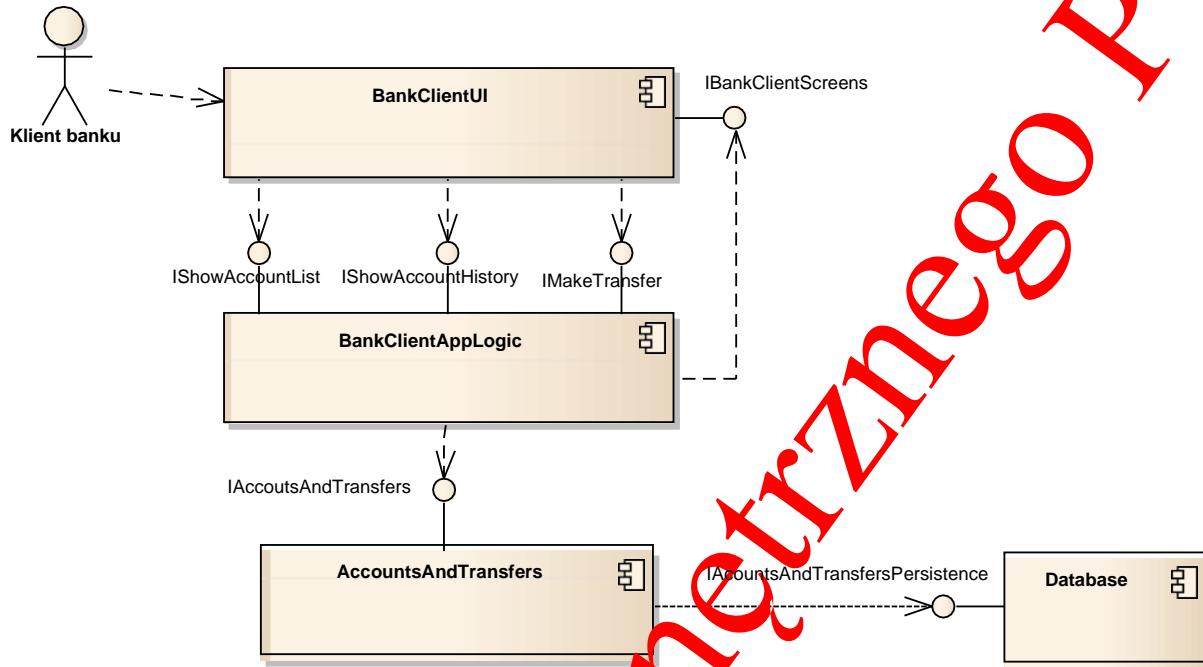
Projekt architektoniczny powinien przede wszystkim umożliwiać realizację wymagań, jednocześnie uwzględniając kwestie techniczne. Warto jest korzystać z reguł przekształcania wymagań w projekt architektoniczny. Reguły takie obejmują m.in. projektowanie komponentów na podstawie pakietów wymagań, tworzenie interfejsów na podstawie jednostek wymagań funkcjonalnych oraz pojęć słowni- kowych, a także definiowane operacje interfejsów na podstawie scenariuszy. Można również stosować reguły projektowania obiektów transferu danych na podstawie słownika dziedziny, a także – sekwencji komunikatów wykorzystujących te obiekty.

Do użytku wewnętrzno-

Rozwiązania zadań

Rozwiązanie zadania 1

Model komponentów realizujący zadane trzy przypadki użycia systemu.

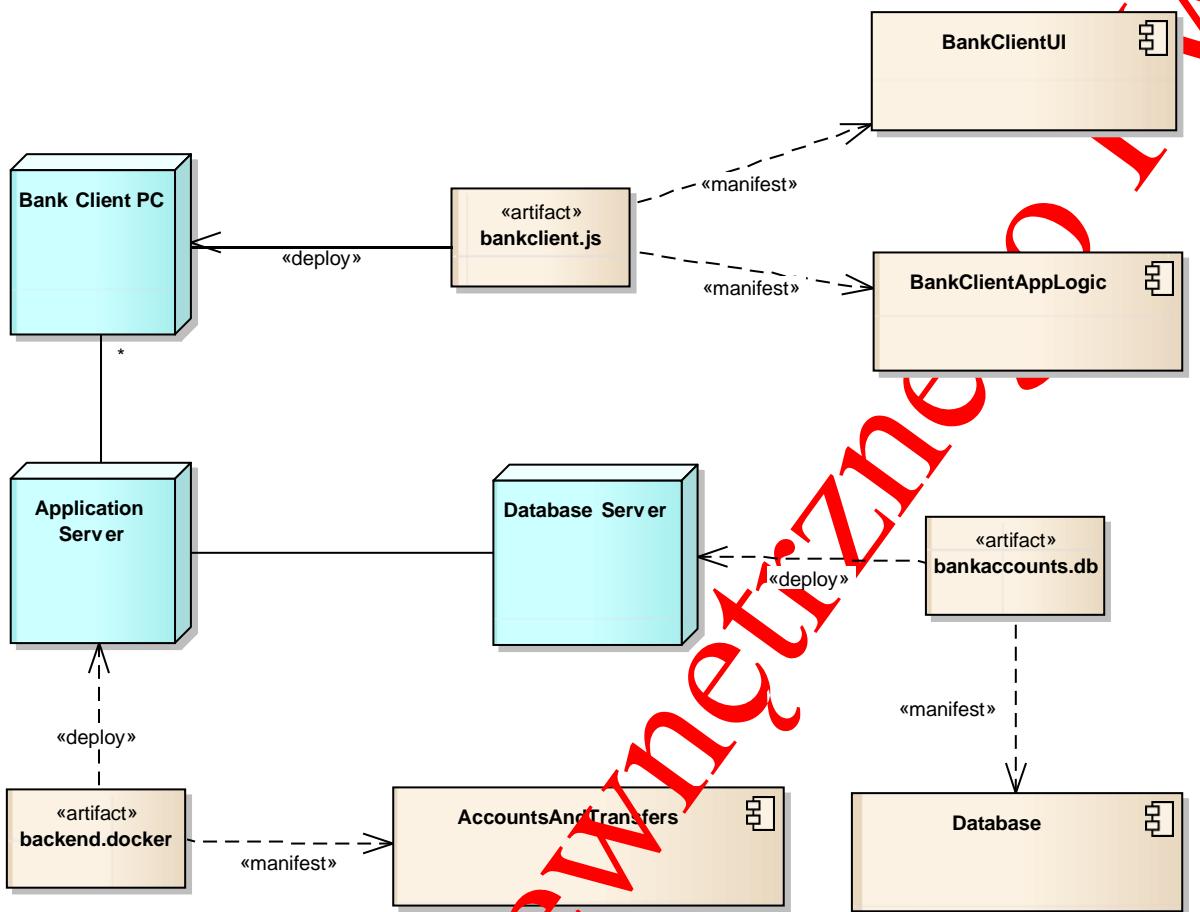


Model utworzony jest w języku angielskim, przy czym nazwy interfejsów odpowiadają nazwom przypadków użycia i pojęć w tłumaczeniu z języka polskiego.

Do użytku wewnętrznego PW

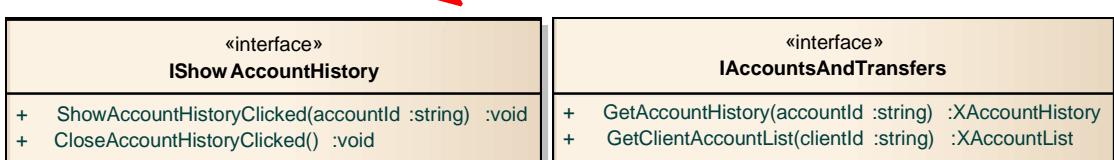
Rozwiązanie zadania 2

Model montażu dla architektury logicznej z rozwiązań zadania 1.



Rozwiązanie zadania 3

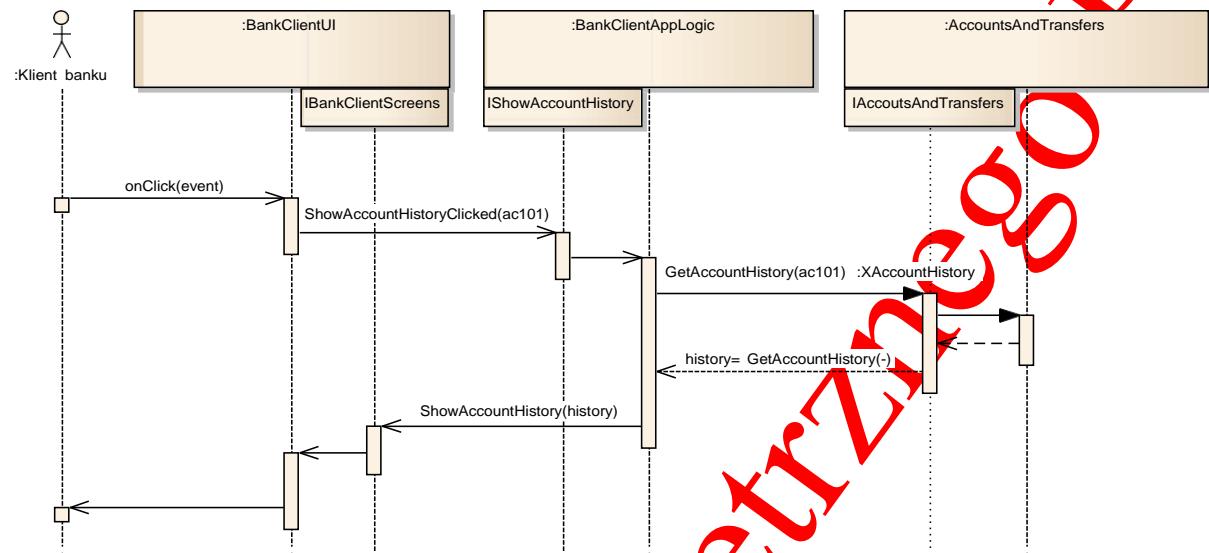
Diagram klas prezentujący szczegóły dwóch interfejsów z rozwiązania zadania 1.



Rozwiązanie zadania 4

Diagram sekwencji dla realizacji scenariusza głównego przypadku użycia „Pokazanie historii rachunku”.

Założono następujący scenariusz: 1. Klient banku wybiera opcję „Pokaż historię rachunku”. 2. System pobiera historię rachunku. 3. System wyświetla okno historii rachunku.



10. Podstawy projektowania podsystemów

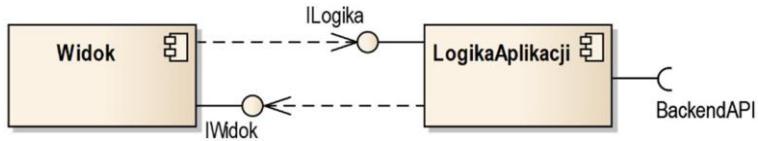
10.1. Projektowanie warstw prezentacji i logiki aplikacji

Komponenty warstw prezentacji i logiki aplikacji tworzą zazwyczaj spójną całość. Zgodnie z tym, co powiedzieliśmy w poprzednim rozdziale, warstwy te składają się na **frontend** systemu. Głównym zadaniem podczas projektowania frontendu jest zaprojektowanie elementów odpowiedzialnych za obsługę interfejsu użytkownika oraz logikę nawigacji między tymi elementami. Bardzo istotne jest również zaprojektowanie współpracy z warstwą logiki dziedzinowej (backendem).

Dobrze zaprojektowany frontend systemu powinien spełniać kilka kryteriów decydujących o łatwości jego implementacji, utrzymania i rozwoju. Pierwsze kryterium dotyczy testowalności. Podczas implementacji komponentu frontendu powinniśmy mieć możliwość łatwego zastąpienia wszystkich zależności od innych komponentów i zewnętrznych bibliotek przez kod tymczasowy (tzw. mock). Szczególnie dotyczy to możliwości zastąpienia API do komponentów logiki dziedzinowej przez tymczasowy kod, który pozwala szybko przetestować samą logikę aplikacji.

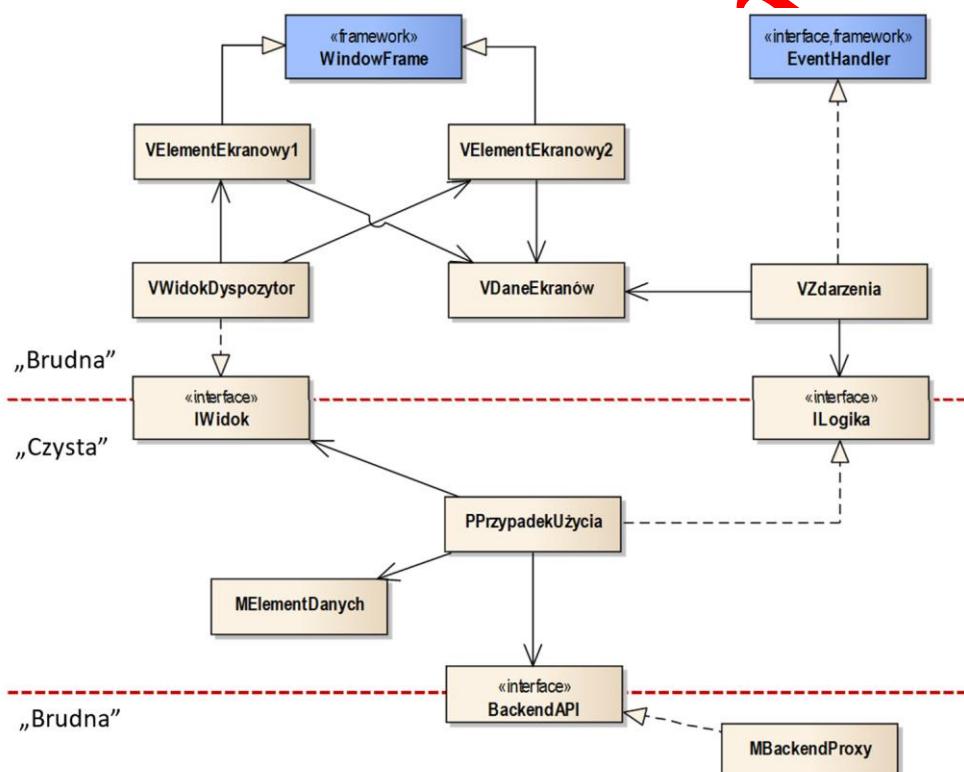
Bardzo istotne jest wyraźne wydzielenie tych elementów frontendu, które są silnie zależne od technologii interfejsu użytkownika oraz technologii wymiany danych. Technologie te ulegają częstym zmianom, czy wręcz są zastępowane innymi. Ulega zmianie format przesyłanych danych czy styl elementów interfejsu użytkownika. Dlatego też, projektując frontend bardzo wskazane jest dokonanie podziału na kod „brudny” i kod „czysty”. **Kod „brudny”** jest podatny na zmiany technologii i podlega istotnym zmianom w trakcie rozwoju całego systemu. W szczególności, kod ten odpowiada np. za wyświetlanie (ang. render) elementów ekranowych zgodnie z wybraną technologią interfejsu użytkownika, za obsługę zdarzeń pochodzących od systemu operacyjnego, oraz za dostosowanie przesyłanych danych do potrzeb technologii API. **Kod „czysty”** jest zależny jedynie od logiki aplikacji opisanej scenariuszami interakcji użytkowników z systemem (np. scenariuszami przypadków użycia). Zawiera on kod niezależny od technologii, stosujący tzw. „czyste” klasy zapisane w wybranym języku oprogramowania. Klasy takie nie są zależne od jakiegokolwiek konkretnego **szkieletu technologicznego** (ang. framework). W stosunku do takich klas w środowiskach języka Java/JavaScript i .NET/CLR (np. C#) często stosuje się skrót POJO/POCO (Plain Old Java(Script)/CLR Object). Ogólnie chodzi tutaj o klasy, które nie implementują żadnych interfejsów zdefiniowanych w ramach szkieletów technologicznych i których struktura nie zależy od wymagań takich szkieletów.

Komponenty warstwy frontend projektujemy na bazie projektu architektonicznego, który został omówiony w poprzednim rozdziale. Punktem wyjścia do projektowania jest zatem model komponentów, którego ogólna struktura jest przedstawiona na rysunku 10.1. Model składa się z komponentów warstwy widoku komunikujących się z warstwą logiki aplikacji za pomocą dwóch interfejsów (jeden udostępniany przez widok, drugi przez logikę aplikacji). Ponadto, komponent warstwy widoku korzysta z interfejsu (API) warstwy logiki dziedzinowej.



Rysunek 10.1: Ogólna struktura komponentów warstwy frontend

Projektując strukturę komponentu warstwy frontend można skorzystać ze wzorca pokazanego na rysunku 10.2. Wzorzec ten definiuje klasy implementujące odpowiednie interfejsy z rysunku 10.1, a także klasy realizujące szczegółową funkcjonalność frontendu. Zwróćmy uwagę na to, że we wzorcu wyraźnie wyróżnione są elementy „czyste” i „brudne”. Wszystkie klasy warstwy widoku traktowane są jako „brudne”, czyli zależne od odpowiedniego szkieletu technologicznego (frameworka). Elementami tego szkieletu są klasa i interfejs opatrzone stereotypem «framework» („WindowFrame” i „EventHandler”). Zakładamy tutaj hipotetyczny szkielet technologiczny, który należy w praktycznym zastosowaniu zmienić na szkielet rzeczywisty (najpopularniejsze szkielety omawiamy niżej).



Rysunek 10.2: Przykładowa struktura komponentu warstwy frontend

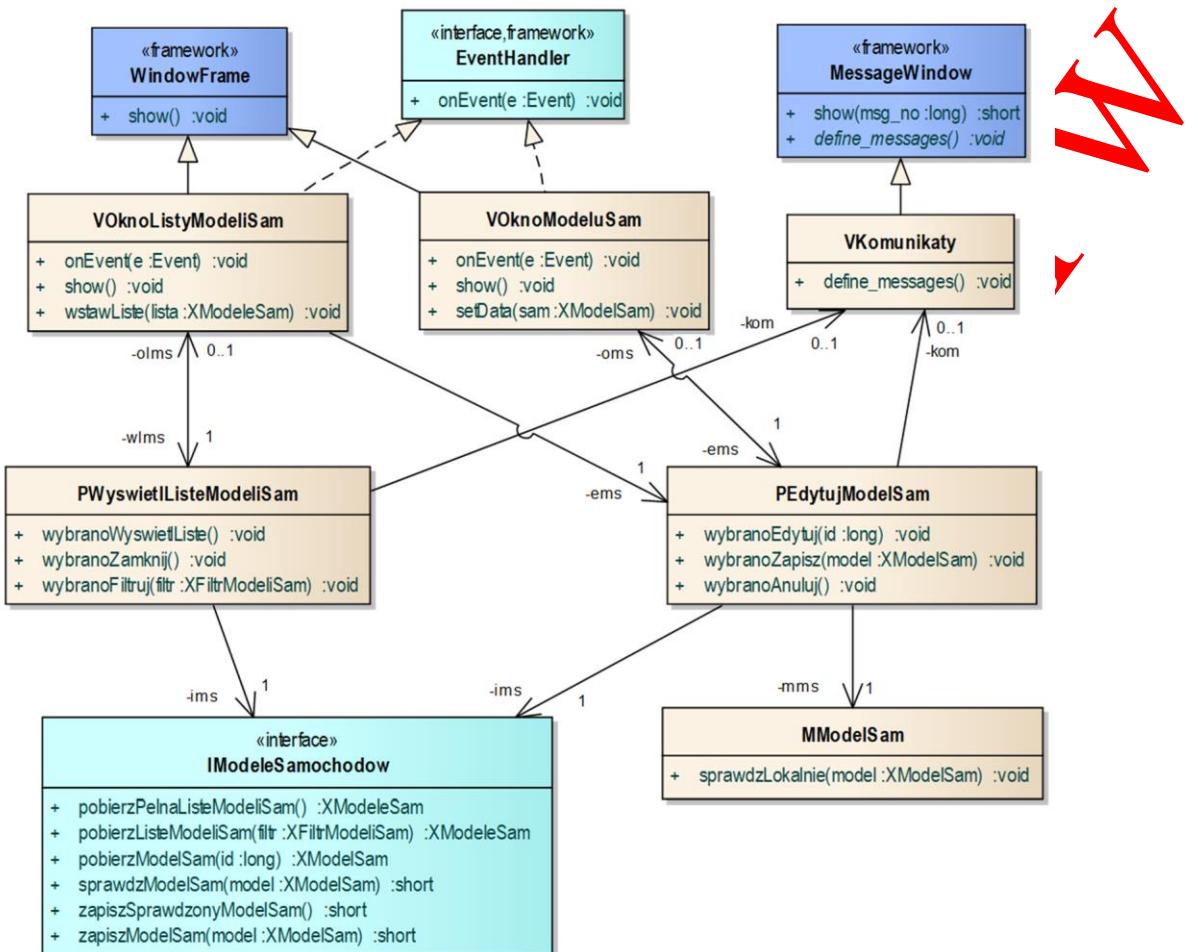
Za wyświetlanie poszczególnych elementów ekranowych odpowiadają klasy specjalizujące od odpowiedniej klasy „okienkowej” zawartej we framework'u (tu: „WindowFrame”). Za przechwytywanie i obsługę zdarzeń pochodzących od użytkownika odpowiada klasa realizująca standardowy interfejs (tu: „EventHandler”). Innym często stosowanym rozwiązaniem jest bezpośrednia realizacja interfejsu obsługi zdarzeń przez klasy wyświetlające elementy ekranowe. W naszej przykładowej strukturze oznaczałoby to, że klasy „VElementEkranowy” oprócz specjalizowania od klasy „WindowFrame” implementowałyby interfejs „EventHandler”.

Komunikacja warstwy widoku z warstwą logiki aplikacji odbywa się za pomocą interfejsów „IWidok” oraz „ILogika”. Pierwszy z tych interfejsów umożliwia sterowanie wyświetlaniem kolejnych widoków (ekranów). W naszym wzorcu, jest on realizowany przez klasę, która spełnia rolę „dyspozytora” („VWidokDyspozytor”). Klasa ta rozdziela polecenia wyświetlania elementów ekranowych do

poszczególnych klas typu „VElementEkranowy”. Drugi z interfejsów służy do przekazywania zdarzeń do realizacji. Każde zdarzenie przechwycone w warstwie widoku jest kierowane do odpowiedniego interfejsu warstwy logiki aplikacji w celu podjęcia odpowiednich działań. Zwróćmy uwagę na to, że warstwa widoku w żadnym wypadku nie realizuje logiki nawigacji między ekranami.

Warstwa logiki aplikacji jest warstwą „czystą” i zawiera klasy implementujące interfejsy typu „ILogika”. Korzystają one również z interfejsów warstwy widoku (tu: „IWidok”) oraz warstwy backend (logiki dziedzinowej, tu: „BackendAPI”). W warstwie tej możemy również umieścić klasy odpowiedzialne za lokalną walidację oraz przetwarzanie danych (tu: „MElementDanych”). Za właściwe przetwarzanie danych odpowiada warstwa backend, do której zazwyczaj odwołujemy się za pomocą API. W naszym wzorcu zastosowaliśmy interfejs („BackendAPI”), który separuje czysty kod od kodu „brudnego”, odpowiedzialnego za komunikację sieciową. Interfejs ten jest realizowany przez tzw. klasę **proxy** (zastępnik, tu: „MBackendProxy”). Jest to klasa, która dokonuje odpowiednich operacji „zastępczych”, tzn. tłumaczy lokalne wywołania procedur zdefiniowanych przez interfejs, na wywołania zdalne, realizowane np. w technologii REST API. Więcej o działaniu klas proxy mówimy w następnym rozdziale.

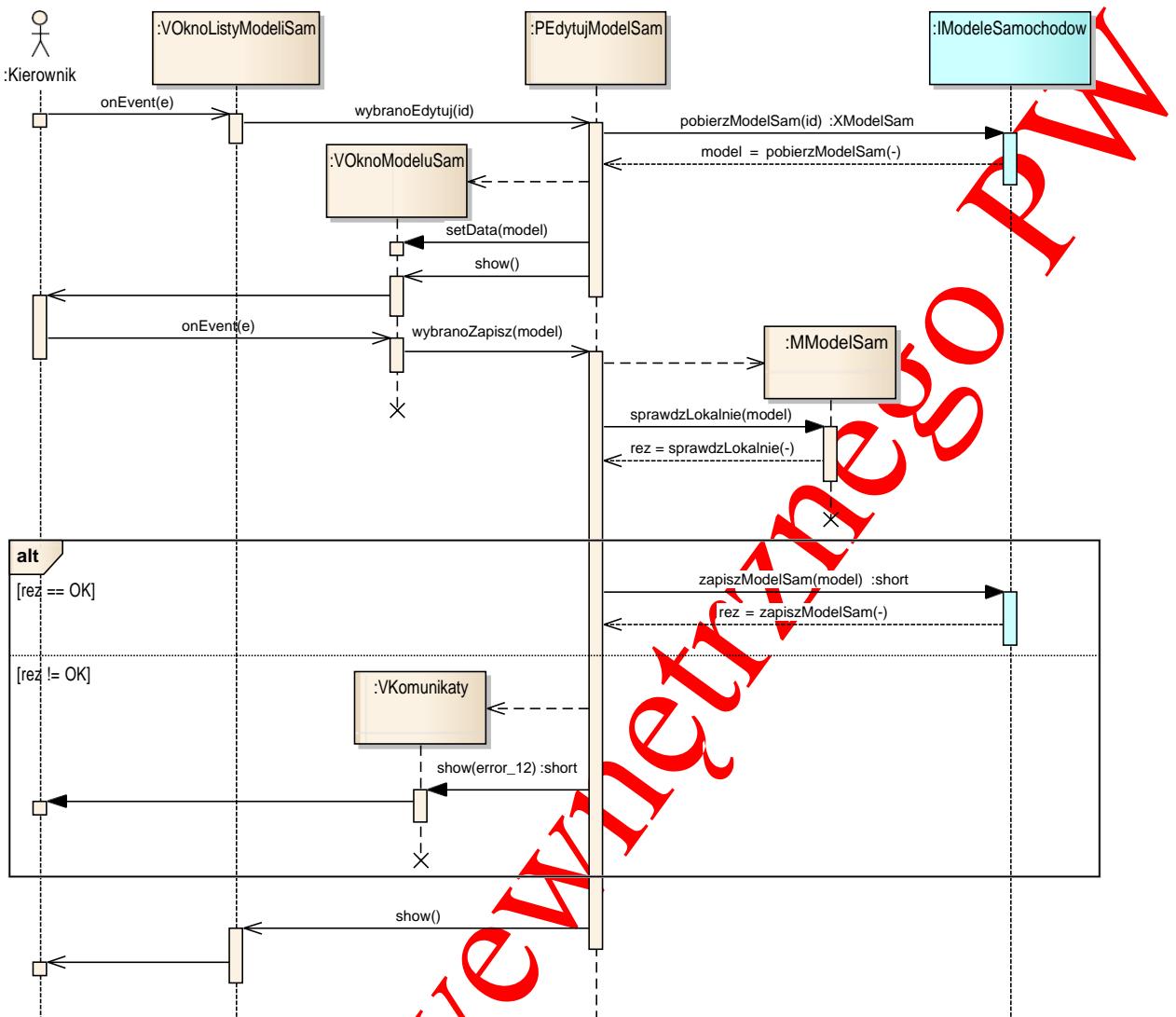
Na rysunku 10.3 widzimy fragment przykładowego projektu komponentu zgodnego z podanym wzorcem. Zwróćmy uwagę na to, że pominęliśmy tutaj interfejsy wewnętrzne oraz klasę „dyspozytora”. Jest to rozwiązanie, które można zastosować dla niewielkich komponentów o prostej logice w celu zmniejszenia rozmiaru kodu. Przykład zawiera szczegółowe specyfikacje operacji interfejsów oraz klas. Mamy tutaj zaprojektowane dwie klasy warstwy logiki aplikacji implementujące logikę dwóch przypadków użycia występujących w przykładach z poprzednich rozdziałów. Każda z tych klas zawiera operacje odpowiadające szczegółowym zdarzeniom wywoływanym interakcjami użytkownika (tutaj są to głównie zdarzenia przyciśnięcia przycisków). Zdarzenia pochodzące od systemu operacyjnego (czyli wywołane przez użytkownika) są obsługiwane bezpośrednio przez klasy „okienkowe” (operacje „onEvent”). Klasę tę jednocześnie specjalizują klasy „WindowFrame” i implementują interfejs „EventHandler”.



Rysunek 10.3: Przykładowy projekt struktury komponentu warstwy frontend

Działanie komponentu z rysunku 10.3 ilustruje sekwencję pokazany na rysunku 10.4. Opisuje on wymianę komunikatów między obiektami w warstwie frontend podczas realizacji przypadku użycia „Edytuj model samochodu”. Sekwencję komunikatów rozpoczyna komunikat przesłany przez system operacyjny (wywołany przez aktora) do odpowiedniego obiektu klasy obsługującej zdarzenia (tu: „VOknoListyModeliSam”). Komunikat ten jest od razu przekazany do odpowiedniego obiektu w warstwie logiki aplikacji (tu: obiekt klasy „PEdytujeModelSam”). Obiekt ten w pierwszej kolejności odwołuje się do warstwy logiki dziedziny z polecienniem pobrania danych modelu samochodu o konkretnym numerze identyfikacyjnym („id”). Następnie, wysyła on polecenie utworzenia nowego obiektu „okienkowego” („VOknoModeluSam”), przesyła do niego dane pobranego modeli oraz prosi o wyświetlenie tych danych (komunikat „show”).

Do użycia



Rysunek 10.4: Przykładowy projekt działania komponentu warstwy frontend

Po wyświetleniu okienka na ekranie, sterowanie jest przekazywane do aktora. W rezultacie, aktor wprowadza odpowiednie nowe dane i uruchamia kolejne zdarzenie. Jest ono ponownie przekazane (wraz z nowymi danymi) przez obiekt klasy „okienkowej” do obiektu warstwy logiki aplikacji („wybranoZapisz”). Tym razem, logika aplikacji decyduje o tym, że przekazane dane powinny być lokalnie sprawdzone. W tym celu najpierw tworzony jest obiekt klasy logiki dziedzinowej („MModelSam”), a następnie przekazywane do niego polecenie sprawdzenia danych. W zależności od rezultatu tego sprawdzenia („rez”) zlecane jest wykonanie zapisu danych lub wyświetlany jest komunikat błędu. Zapis danych odbywa się za pośrednictwem warstwy logiki dziedzinowej umieszczonej w innym komponencie, czyli – za pośrednictwem odpowiedniego interfejsu („IModeleSamochodow”).

Warto zwrócić uwagę na to, że obiekt klasy „VKomunikaty” zachowuje się inaczej niż inne obiekty okienkowe (por. „VOknoModeluSam”). Komunikacja z nim odbywa się przez komunikaty synchroniczne. Wynika to z tego, że wyświetlany przez ten obiekt element ekranowy jest okienkiem modalnym (np. okienko wyskakujące) o ograniczonej funkcjonalności. Wykorzystanie komunikacji synchronicznej upraszcza komunikację, gdyż nie ma potrzeby organizowania systemu obsługi zdarzeń dla tego okienka. Jego sposób zachowania jest standardowy – pokazanie komunikatu na ekranie i oczekiwanie na potwierdzenie przez użytkownika. Z kolei, obiekt zlecający wyświetlenie komunikatu

(„:PEdytujeModelSam”) musi czekać na to potwierdzenie, co jest dobrze obsługiwane przez komunikację synchroniczną.

Zauważmy, że klasy w naszych powyższych przykładach mają nadane przedrostki określające ich przynależność do konkretnych warstw. Klasy widoku posiadają przedrostek „V” od angielskiego „View”. Klasy logiki aplikacji sterują prezentacją, zatem nadajemy im przedrostek „P” od angielskiego „Presenter”. Z kolei klasy warstwy logiki dziedzinowej odpowiadają modelowi danych, zatem nadajemy im przedrostek „M”, czyli „Model”. Od tak określonych zakresów odpowiedzialności poszczególnych warstw pochodzi definicja wzorca o nazwie Model-View-Presenter (MVP). Jest to jeden z podstawowych wzorców architektoniczno-projektowych, które są stosowane w systemach interaktywnych. W zaprezentowanym powyżej podejściu wykorzystane zostały właśnie podstawowe zasady wzorca MVP. Jest on oparty na innym wzorcu – Model-View-Controller, w którym odpowiedzialność poszczególnych warstw i relacje między nimi są określone nieco inaczej.

Należy podkreślić, że podany tutaj wzorzec struktury komponentu warstwy frontend jest dużym uogólnieniem. Szczegółowe rozwiązania zależą w dużym stopniu od wyboru konkretnego, rzeczywistego szkieletu technologicznego. Najczęściej spotykane szkielety oparte są na języku **JavaScript** (oraz **TypeScript**). Większość z nich realizuje przetwarzanie po stronie klienta (ang. client side), tzn. kod wykonywany jest na maszynie użytkownika końcowego (np. w przeglądarce internetowej). Z najpopularniejszych technologii warto wymienić React, Vue, Angular i Vanilla. Technologia Node, również oparta na języku JavaScript realizuje przetwarzania po stronie serwera (ang. server side). Oznacza to, że cały kod jest wykonywany na maszynie serwerowej, a na maszynie klienta jest przesyłany jedynie gotowy widok (np. plik HTML). Istnieją również, znacznie rzadziej używane, technologie frontend oparte na innych językach programowania. Technologie GWT i Vaadin wykonują kod bazujący na języku **Java** po stronie serwera, a technologie JavaFX i Swing – po stronie klienta. Technologia Blazor oparta jest na środowisku **.NET** i języku **C#** i wykonuje kod po stronie klienta. Podobna jest zasada działania technologii **Dart** opartej na języku o tej samej nazwie. Powyższe technologie mogą pracować w oparciu o różne środowiska wykonawcze. Część z nich pozwala na tworzenie aplikacji przeglądarczych (aplikacje WWW), część – aplikacji na urządzenia mobilne, a część – komputery PC (aplikacje desktop). Szczegółowe omówienie tych technologii wykracza poza zakres niniejszych materiałów i odsyłamy do ich dokumentacji.

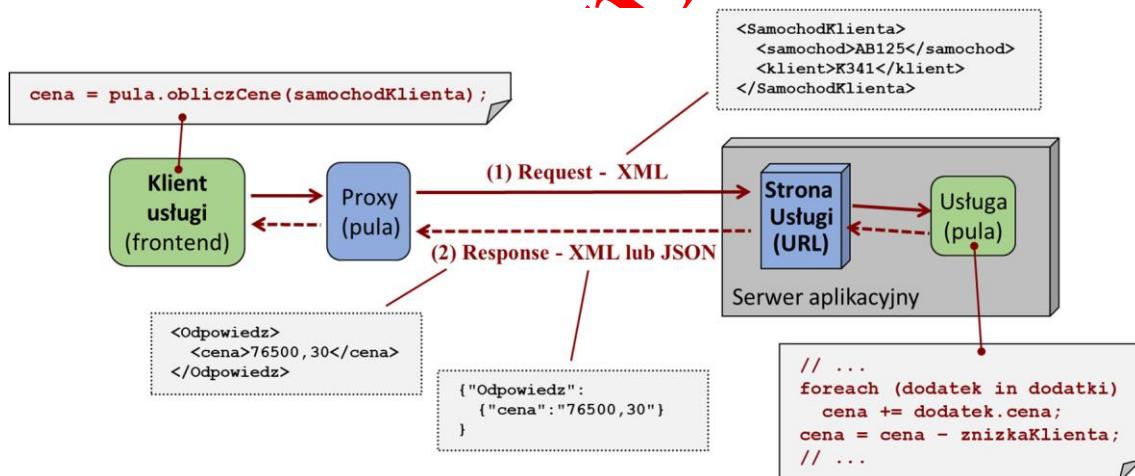
10.2. Projektowanie warstwy logiki dziedzinowej

Jak wiemy z poprzedniego rozdziału, komponenty warstwy logiki dziedzinowej odpowiadają za wykonywanie zadań zleczanych przede wszystkim przez warstwę logiki aplikacji. Zadania te dotyczą wszelkiego rodzaju przetwarzania, zapamiętywania oraz odczytu danych. Logika dziedzinowa może być umieszczona w tym samym węźle wykonawczym i być ściśle powiązana z elementami logiki aplikacji. Przykłady „lokalnych” klas logiki dziedzinowej widzimy na rysunkach 10.2 i 10.3 – są to klasy z przedrostkiem „M” („MElementDanych”, „MModelSam”). Realizują one stosunkowo proste, lokalne przetwarzanie danych. Najczęstszym zastosowaniem takich klas jest lokalna walidacja danych, nie wymagająca dużych zasobów obliczeniowych oraz dostępu do danych trwałych (np. do bazy danych). Zwróćmy uwagę na to, że często operacje logiki dziedzinowej projektuje się jako składnik elementów ekranowych (w warstwie widoku). Najczęstszym przykładem jest walidacja danych wprowadzanych na różnego rodzaju formularzach bezpośrednio w kodzie elementów ekranowych. Takie rozwiązanie należy uznać za problematyczne z tego powodu, że znacznie utrudnia modyfikację kodu w razie zmian wymagań. Kod dotyczący walidacji jednego elementu danych (np. danych samochodu) może być rozproszony między wiele klas obsługujących różne kontrolki ekranowe. Modyfikacja takiego kodu wymaga bardzo uważnego odszukania wszystkich tych klas i dokonania niezbędnych zmian. Co więcej,

taki kod jest najczęściej osadzony i „przemieszany” z kodem ściśle powiązanym z konkretną technologią interfejsu użytkownika. Każda zmiana technologii powoduje konieczność całkowitej przebudowy takich lokalnych operacji logiki dziedzinowej.

Zasadnicza funkcjonalność logiki dziedzinowej zazwyczaj projektowana jest jako zestaw niezależnych komponentów, często traktowanych jako usługi. Dla każdego takiego komponentu projektowany jest jeden lub więcej interfejsów zawierających odpowiednie zestawy operacji na obiektach dotyczących danej dziedziny problemu. Mamy wtedy do czynienia z wyraźnie wydzieloną warstwą backendu, na którą dodatkowo składa się warstwa przechowywania danych.

Jeśli warstwa frontendu wykonywana jest po stronie serwera, to komunikacja z warstwą logiki dziedzinowej odbywa się poprzez zwyczajne, lokalne wywoływanie procedur klas, realizujących odpowiednie interfejsy. Znacznie bardziej złożona jest sytuacja, jeśli warstwa frontendu wykonywana jest po stronie klienta, tzn. na maszynie użytkownika końcowego. W takiej sytuacji, aby wykonać operację logiki dziedzinowej należy skomunikować się z warstwą backendu poprzez **zdalne wywoływanie procedur**. Typowy mechanizm takiej komunikacji ilustruje przykład przedstawiony na rysunku 10.5. W przykładzie tym, kod frontendu chce wywołać procedurę obliczania ceny samochodu („obliczCene”). Nie może tego wykonać bezpośrednio, gdyż kod odpowiedniej usługi logiki dziedzinowej znajduje się na serwerze aplikacyjnym. Rozwiązaniem jest wykorzystanie kodu pośredniego (proxy), który zamieni proste wywołanie lokalne na wywołanie zdalne. Klasa proxy implementuje odpowiednią technologię wywołań zdalnych (RPC, REST, SOAP). W technologiach typu REST i SOAP dane przekazane jako parametry procedury lokalnej zamieniane są w odpowiedni plik tekstowy (tzw. serializacja danych), np. w formacie XML lub JSON. W technologiach typu RPC dane są najczęściej przesyłane w zwartej postaci binarnej.

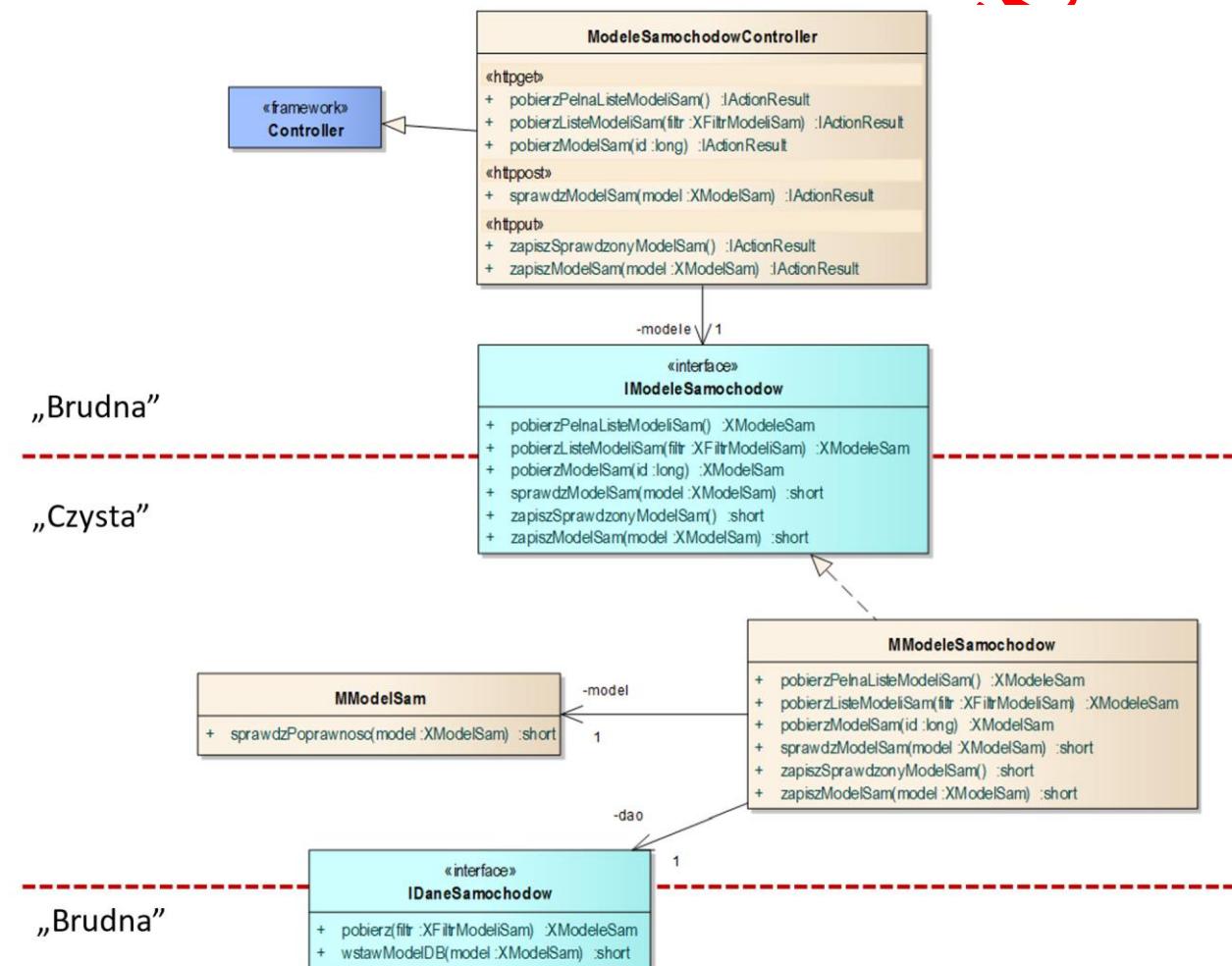


Rysunek 10.5: Zasada działania wywołań zdalnych (RPC, REST)

Obiekt klasy proxy przesyła pod odpowiedni adres (URL) serwera, **komunikat żądania** (ang. request) z umieszczonym w nim obiektem danych. Taki komunikat jest interpretowany przez serwer, w wyniku czego powinna zostać wywołana odpowiednia procedura obsługi żądania. Procedura ta odpowiada procedurze wywoływanej po stronie klienta i zawiera odpowiedni kod logiki aplikacji (tutaj: kod obliczania ceny samochodu). Po wykonaniu procedury, zwraca ona wynik i przekazuje sterowanie. W rezultacie, odpowiedni kod po stronie serwera powinien wysłać **komunikat odpowiedzi** (ang. response). Komunikat ten zawiera wynik przetwarzania danych wraz z ew. obiektem danych (tu: obiekt zawierający wyliczoną cenę). Po stronie klienta odpowiedź ta skutkuje powrotem z wywołania procedury i kontynuacją działania przez kod frontendu.

Realizacja powyższego mechanizmu przez komponenty logiki dziedzinowej zależy od zastosowanego szkieletu technologicznego (frameworku), umożliwiającego realizację wywołań zdalnych. Tutaj

omówimy jedno z typowych podejść do tego zagadnienia. Na rysunku 10.6 widzimy przykładową strukturę komponentu logiki dziedzinowej. Za obsługę wywołań zdalnych odpowiada „brudna” część tego komponentu, czyli część zależna od wybranego szkieletu technologicznego. W naszym przykładzie, za projektowano tzw. **klasę kontrolera API** (ang. API controller) o nazwie „ModeleSamochodowController”. Klasa ta specjalizuje standardową klasę „Controller”, która zapewnia obsługę odpowiednich mechanizmów technologicznych. Nasza klasa kontrolera zawiera obsługę wszystkich operacji realizowanego przez komponent interfejsu („IModeleSamochodów”, patrz również rysunek 10.3). Operacje te są zadeklarowane jako obsługujące odpowiednie zapytania HTTP (GET, POST, PUSH). Sygnatury tych operacji dostosowane są do wybranego szkieletu technologicznego dla obsługi API REST.



Rysunek 10.6: Przykładowa struktura komponentu backend

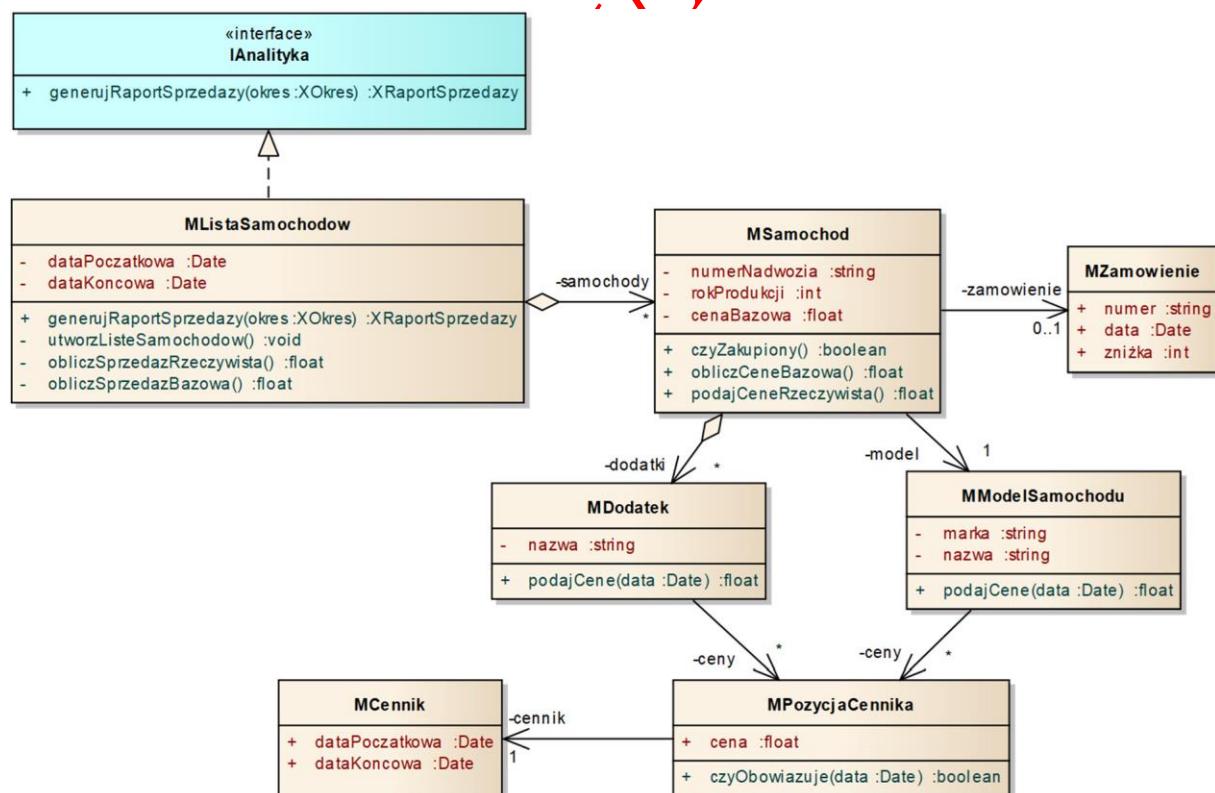
Zaprojektowany przez nas kontroler API odwołuje się do części „czystej” komponentu za pośrednictwem interfejsu. Interfejs ten jest realizowany przez odpowiednią klasę (tu: „MModeleSamochodow”), która może korzystać z pomocy innych klas, oraz odwoływać się do interfejsów innych komponentów. Najczęściej dotyczy to dostępu do komponentów warstwy przechowywania danych (tu: „IDaneSamochodow”).

Pokazana na rysunku 10.6 struktura komponentu jest stosunkowo prosta. Można powiedzieć, że realizuje ona wzorzec **skryptu transakcyjnego** (ang. transaction script). Wzorzec ten polega na organizacji całej logiki w pojedyncze transakcje realizowane w sposób proceduralny. Każda transakcja odpowiada jednej procedurze. Tego typu rozwiązanie jest najczęściej stosowane podczas realizacji prostych

operacji zapisywania oraz odczytywania danych oraz prostych obliczeń możliwych do obsłużenia w jednej procedurze.

Diagram na rysunku 10.6 nieco łamie zasadę skryptu transakcyjnego poprzez wydzielenie operacji sprawdzania poprawności danych do osobnej klasy. W ten sposób możemy wyraźnie wydzielić pewne operacje zgrupowane wokół określonych elementów modelu danych. W bardziej złożonych sytuacjach, kiedy logika dziedziny implementuje złożone algorytmy, warto zastosować rozdzielenie odpowiedzialności na wiele klas. Tego typu podejście realizuje wzorzec **modelu dziedziny** (ang. domain model). We wzorcu tym, logika jest rozproszona między klasy zgodne z modelem opisującym daną dziedzinę problemu. Źródłem dla modelu dziedziny może być szczegółowy słownik dziedziny opracowany podczas specyfikowania wymagań użytkownika oraz wymagań oprogramowania. Zadaniem projektanta jest odpowiednie zdefiniowanie operacji poszczególnych klas, tak, aby rozdzielić składowe zadania przetwarzania danych między klasy, które te dane reprezentują.

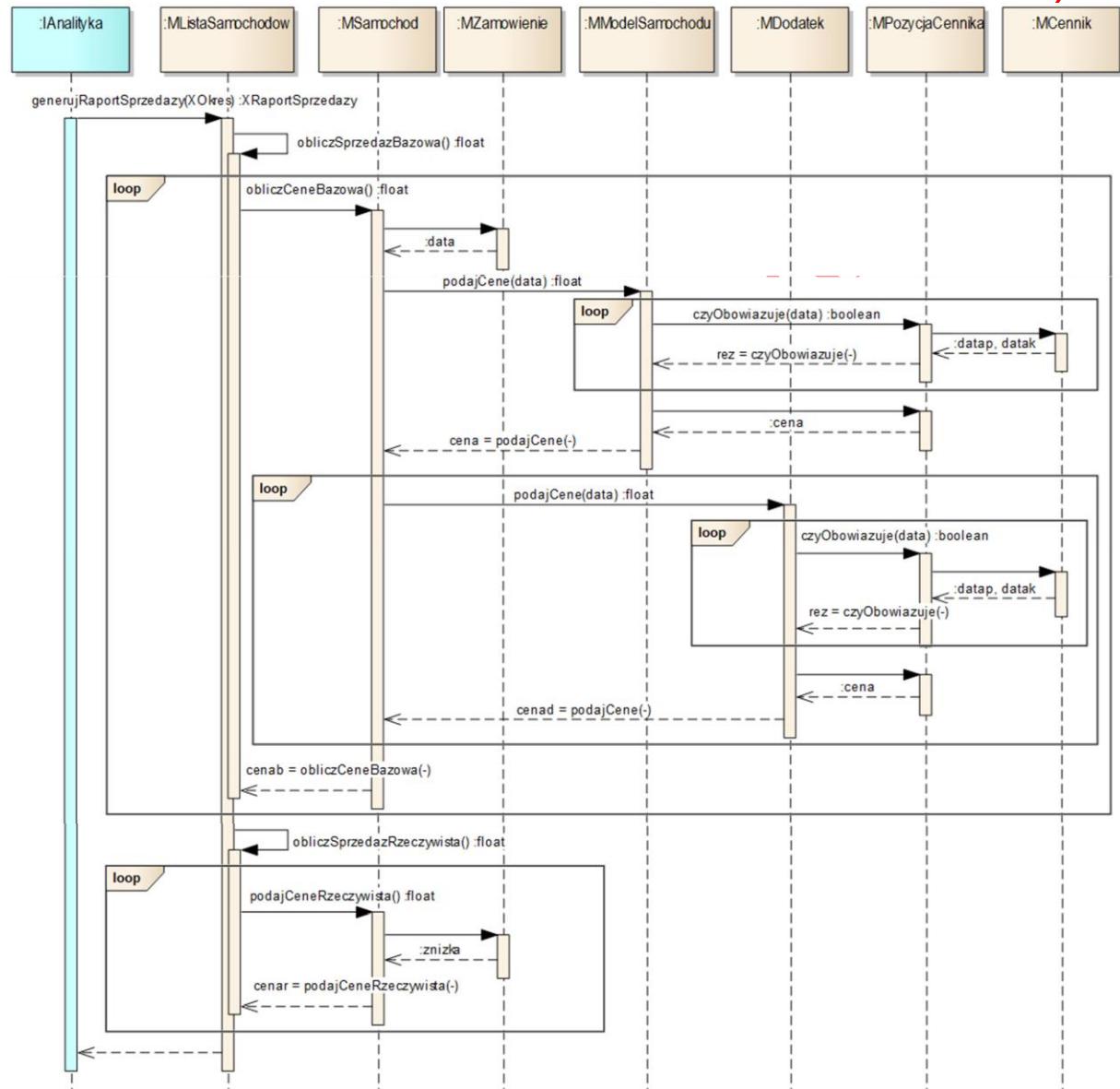
Przykład zastosowania wzorca model dziedziny widzimy na rysunku 10.7. Prezentuje on fragment projektu komponentu obsługującego raportowanie, a konkretnie – realizację operacji generowania raportu sprzedaży. Odpowiedni interfejs jest realizowany przez klasę, która wspomagana jest przez zestaw klas, które są zaprojektowane na podstawie modelu słownika opisanego w poprzednim rozdziale. Klasy te współpracują ze sobą w celu implementacji algorytmu generowania raportu. Zwróćmy uwagę, że dosyć złożona struktura komponentu przysłonięta jest dla innych komponentów przez wąski zestaw operacji interfejsu (tu: zaledwie jedna operacja interfejsu „IAalityka”). Jest to realizacja wzorca projektowego **fasada** (ang. facade). Wzorzec ten występuje w wielu opisywanych już przez nas przykładach – wszędzie tam, gdzie interfejsy ukrywają złożoną strukturę zawartą we wnętrzu realizowanych przez nie komponentów.



Rysunek 10.7: Komponent backend zaprojektowany zgodnie z modelem dziedziny

Działanie wzorca fasady połączonego z wzorcem modelu dziedziny ilustruje rysunek 10.8. Przedstawia on sposób realizacji operacji „generujRaportSprzedazy”. Odpowiedni algorytm realizowany jest przez

dwie operacje prywatne klasy „MListaSamochodow”. Każda z tych operacji implementuje odpowiednią pętlę, która dokonuje odpowiednich obliczeń cen bazowych oraz cen rzeczywistych samochodów będących w sprzedaży w danym okresie sprzedawczym. Jak widzimy, cały algorytm oparty jest na dosyć ścisłej współpracy obiektów kilku klas. Zwróćmy uwagę na to, że w sytuacji, kiedy realizowana jest pętla (ramka „loop”), to przesyłane komunikaty mogą dotyczyć wielu obiektów danej klasy. Na przykład, operacja „obliczCeneBazowa” jest uruchamiana wiele razy w pętli dla kolejnych obiektów klasy „MSamochod”. Szczegółową analizę działania operacji generowania raportu pozostawiamy czytelnikowi.



Rysunek 10.8: Działanie przykładowego komponentu backend

10.3. Projektowanie baz danych

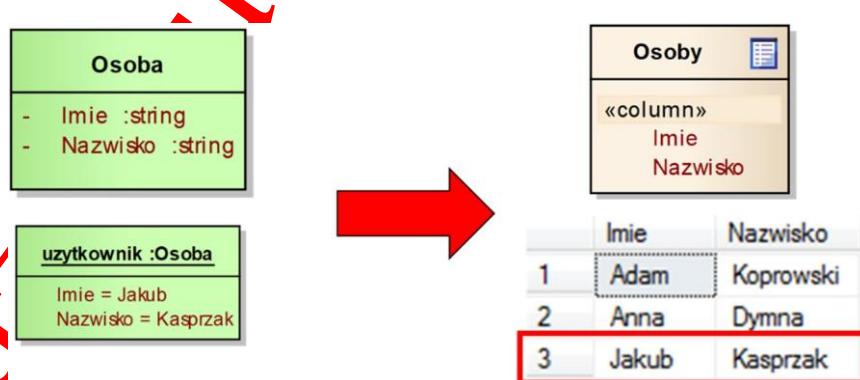
Komponenty warstwy przechowywania danych mogą być realizowane w różnych technologiach i mogą przechowywać dane różnego rodzaju w różnych formatach i strukturach. Omówienie wszystkich dostępnych technologii przechowywania danych jest poza zakresem niniejszych materiałów. Tutaj

skoncentrujemy się na dwóch najczęściej używanych paradymatach przechowywania danych – bazach danych relacyjnych oraz bazach danych nierelacyjnych.

Technologia **baz danych relacyjnych** jest obecnie powszechnie stosowana jako podstawowa metoda przechowywania danych ustrukturalizowanych. Technologia ta jest oparta na teorii relacyjnej zaproponowanej w 1970 roku. Podstawowym językiem umożliwiającym dokonywanie operacji na danych w bazach relacyjnych jest język **SQL** (Structured Query Language – strukturalny język zapytań). Język ten pozwala na pisanie tzw. zapytań do bazy danych, które pozwalają na wykonywanie złożonych operacji zapisu i odczytu danych. W języku SQL definiujemy również strukturę bazy danych, która oparta jest na czterech podstawowych konstrukcjach – tabelach, kolumnach, wierszach i powiązaniach. Opisując zasady projektowania baz danych relacyjnych zakładamy, że czytelnik jest zaznajomiony z podstawami tej technologii.

Podstawą projektu bazy danych jest definicja jej struktury. W tym celu możemy wykorzystać notację **ERD/ERM** (Entity Relationship Diagram/Model – diagram/model związków encji). Notacja ta nie jest ustandaryzowana i istnieją różne jej warianty. Jest ona podobna do notacji modelu klas języka UML. Tabele bazodanowe odpowiadają klasom, a kolumny w tabelach odpowiadają atrybutom. Powiązania między tabelami (często mylnie nazywane relacjami) odpowiadają asocjacjom między klasami. Podstawową jednostką przechowywania danych w bazie relacyjnej jest wiersz tabeli. Analogiem wiersza w podejściu obiektowym jest obiekt, zatem wiersze nie są zazwyczaj uwzględniane podczas projektowania bazy danych. Warto podkreślić, że model relacyjny posiada istotne różnice w stosunku do modelu obiektowego. Dlatego też konieczne jest odpowiednie przetłumaczenie modelu dziedziny zapisanego jako model obiektowy w języku UML w model relacyjny. Poniżej przedstawiamy zasady takiej translacji.

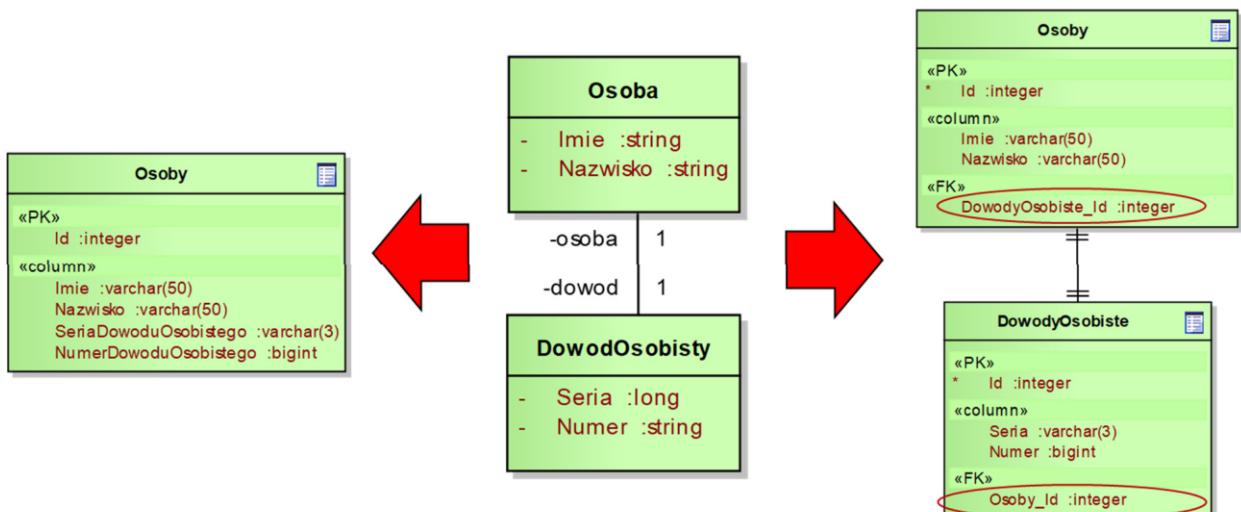
Rysunek 10.9 ilustruje podstawowe reguły projektowania tabel relacyjnych. Bierzemy pod uwagę klasy modelu dziedziny, które będą wymagały przechowywania ich obiektów w bazie danych. Dla takich klas tworzymy odpowiadające im tabele. Zwróćmy uwagę na to, że przyjętą konwencją jest nazywanie tabel w liczbie mnogiej, np. na podstawie klasy „Osoba” tworzymy tabelę „Osoby”. Proste atrybuty klas (typu napisowego, liczbowego itp.) zamieniamy w kolumny tabeli. Dla atrybutów typów złożonych stosujemy zasady opisane poniżej, dotyczące projektowania asocjacji. Na rysunku 10.9 widzimy również zasadę przechowywania obiektów w relacyjnej bazie danych. Obiekty utworzone np. w pamięci ulotnej (tu: obiekt „uzytkownik”) są zapisywane jako wiersze w tabeli, gdzie kolumny tej tabeli odpowiadają atrybutom klas.



Rysunek 10.9: Tworzenie tabel na podstawie klas modelu dziedziny

Na rysunku 10.10 widzimy dwie metody projektowania struktury bazy danych dla asocjacji o krotności „1” na obydwu końcach (asocjacje „1-1”). Takie asocjacje w modelu klas oznaczają, że jeden obiekt jednej z klas jest zawsze powiązany z jednym obiektem drugiej z klas. W takiej sytuacji, najprostszym rozwiązaniem (patrz lewa strona rysunku) jest zaprojektowanie jednej tabeli zawierającej kolumny

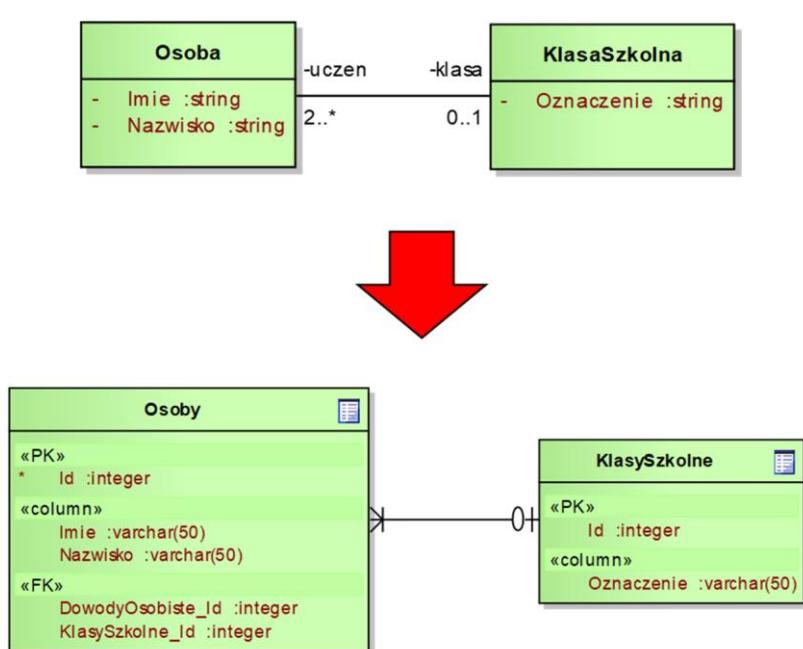
odpowiedzające atrybutom obydwu klas. Innym rozwiązańiem jest stworzenie dwóch tabel odpowiadających dwóm klasom i stanowienie odpowiedniego połączenia między nimi (patrz prawa strona rysunku). Połączenie tabel oznaczone jest linią, a krotności odpowiednimi symbolami (dwie kreseczki oznaczają krotność „1”). Zwróćmy uwagę na to, że w tabelach zostały umieszczone dodatkowe pseudo-kolumny oznaczone jako «PK» i «FK». Są to tzw. **klucze główne** (ang. primary key) oraz **klucze obce** (ang. foreign key). Klucz główny stanowi unikalny identyfikator wiersza w danej tabeli. Z kolei klucz obcy zawiera wartość klucza głównego w innej tabeli, który odpowiada wierszowi połączonym z danym wierszem. Dzięki mechanizmowi kluczy możliwa jest łatwe oznaczanie połączeń między wierszami (obiektami).



Rysunek 10.10: Tworzenie struktury tabel na podstawie asocjacji o krotności „1-1”

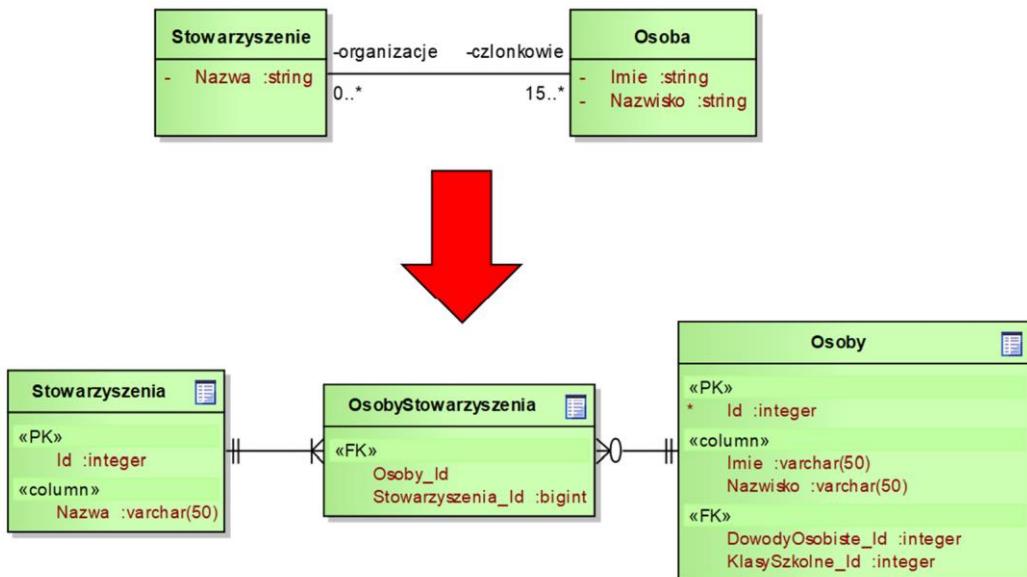
Rysunek 10.11 przedstawia sposób realizacji asocjacji o krotnościach „1” oraz „*” (wiele) – inaczej nazywanymi asocjacjami „1-N”. W naszym przykładzie, osoba może być uczniem co najwyżej jednej klasy szkolnej, natomiast do klasy szkolnej może być zapisanych wielu uczniów (co najmniej dwóch). W takiej sytuacji, podobnie jak w przypadku asocjacji „1-1” tworzymy dwie tabele. Podstawowa różnica polega na tym, że klucz obcy umieszczamy tylko w jednej tabeli – tej, która w powiązaniu posiada krotność „wiele”. Można tutaj zauważać pewną wadę technologii relacyjnej, gdyż nawet niewielka zmiana krotności może powodować konieczność przebudowy struktury bazy danych (np. zmianę kluczy obcych). Zwróćmy jeszcze uwagę na oznaczenia krotności w notacji ERD – krotność „N” oznaczana jest jako tzw. „kurza łapka”, a krotność „0” lub „1” jako kółko z kreseczką.

Do użytku!



Rysunek 10.11: Tworzenie struktury tabel na podstawie asocjacji o krotności „1-N”

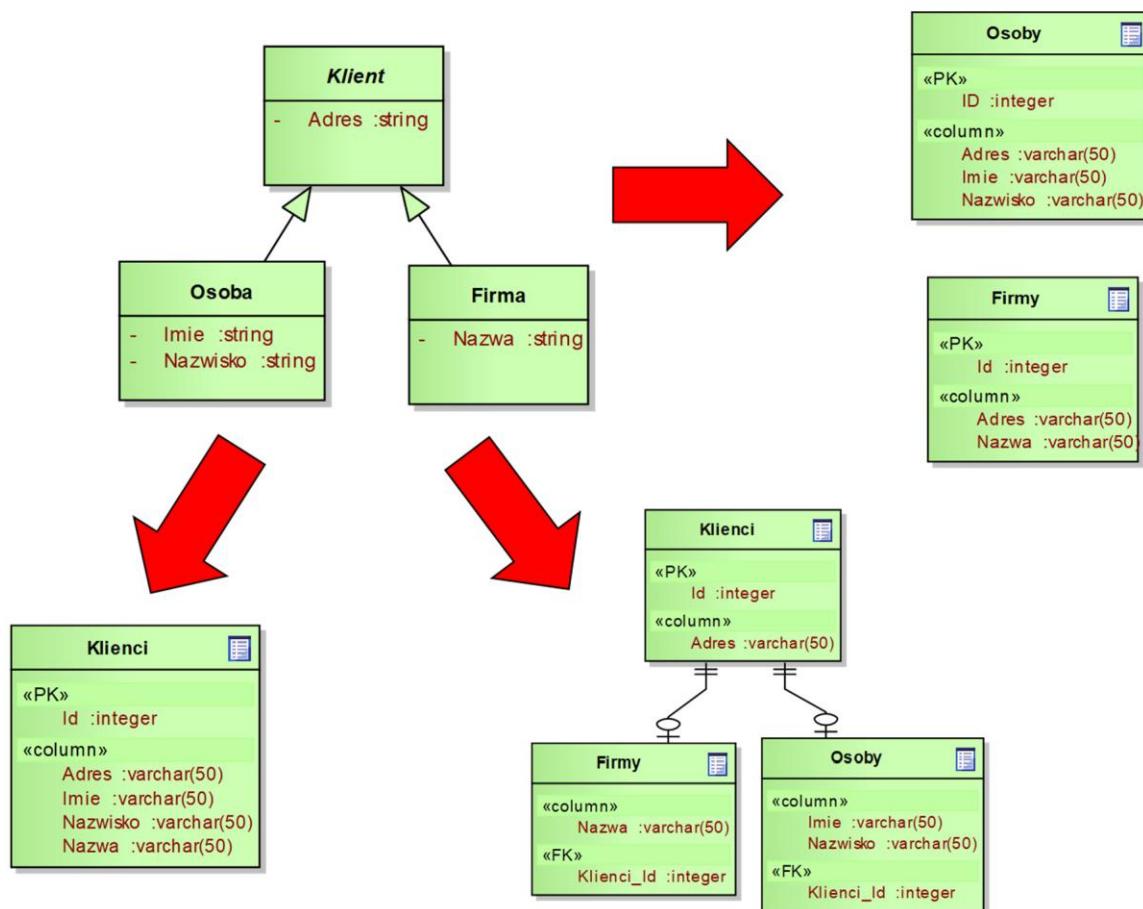
Rysunek 10.12 przedstawia najbardziej złożoną sytuację dotyczącą asocjacji, czyli realizację asocjacji „wiele” do „wielu” („N-N”). Mamy tu przykład sytuacji, kiedy osoba może należeć do wielu stowarzyszeń, a stowarzyszenie może liczyć wielu członków będących osobami. W technologii relacyjnej wymaga to ustanowienia dodatkowej tabeli (tu: „OsobyStowarzyszenia”), która zawiera klucze obce wskazujące na dwie tabele utworzone na podstawie klas. Rozwiążanie takie jest konieczne z uwagi na ograniczenia modelu relacyjnego. Tabela nie może np. zawierać listy kluczy obcych, tak jak to jest możliwe w przypadku stosowania paradygmatu obiektowego.



Rysunek 10.12: Tworzenie struktury tabel na podstawie asocjacji o krotności „N-N”

Ostatni przykład dotyczy traktowania relacji generalizacji. Na rysunku 10.13 widzimy dwie klasy („Osoba” i „Firma”), które specjalizują od klasy ogólnej („Klient”). W modelu relacyjnym nie możemy relacji generalizacji zamodelować bezpośrednio. Możemy natomiast zastosować jedno z kilku

rozwiązań pokazanych na rysunku. Najprostsze rozwiązanie polega na umieszczeniu wszystkich kolumn w jednej tabeli odpowiadającej klasie ogólnej. To rozwiązanie ma zaletę prostoty oraz niekiedy – lepszej wydajności. Podstawowa wada polega na redundancji danych – dla niektórych wierszy niektóre kolumny będą puste. Drugie rozwiązanie polega na stworzeniu tabel jedynie dla klas szczegółowych. W takiej sytuacji, w tabelach tych dodajemy atrybuty klasy ogólnej. Takie rozwiązanie nie jest jednak polecane dla bardziej złożonych hierarchii generalizacji. Ostatnie rozwiązanie polega na utworzeniu tabel dla każdej klasy występującej w relacji generalizacji. W takiej sytuacji w tabelach odpowiadających klasom specjalizowanym należy umieścić klucze obce wskazujące na wiersze w tabeli odpowiadającej klasie ogólnej.



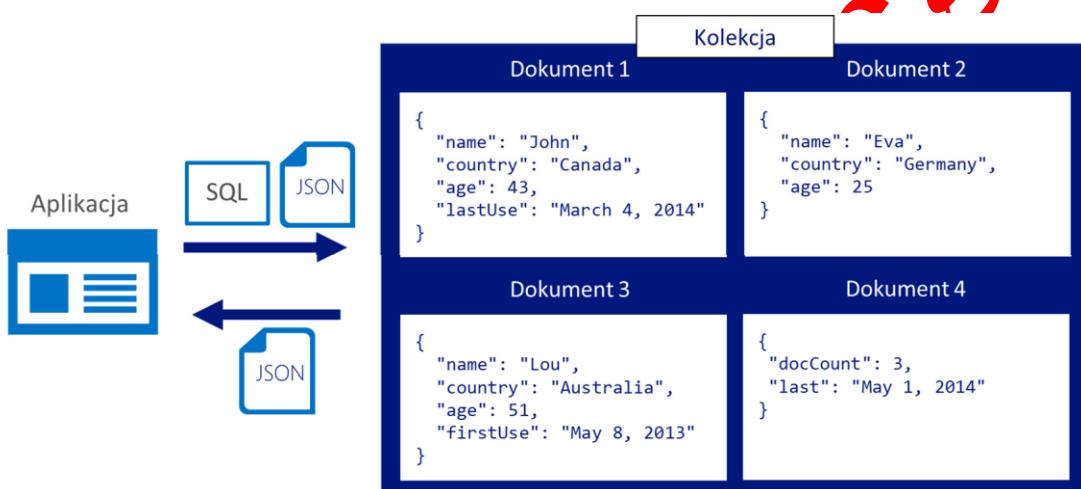
Rysunek 10.13: Tworzenie struktury tabel na podstawie generalizacji

We wszystkich powyższych przykładach w tabelach umieszczaliśmy kolumny o określonych typach. Typy te są charakterystyczne dla technologii baz danych relacyjnych. Konieczna jest zatem translacja typu określonego w modelu danych (obiektowym) na typ bazodanowy. Przykładowo, typ „string” powinniśmy zamienić na typ „varchar”.

Drugą popularną technologią baz danych są **bazy danych nierelacyjne**. Bazy takie często nazywane są bazami **NoSQL** (Not Only SQL), gdyż na ogół nie wykorzystują one języka SQL, lub język ten jest traktowany pomocniczo. Czasami do formułowania zapytań wykorzystywany jest dosyć okrojony wariant języka SQL. Podstawową zasadą wielu baz danych NoSQL jest przechowywanie danych w postaci dokumentów (bazy dokumentowe). Pozwala to uzyskać bardzo dużą elastyczność struktur takich baz danych. Ponadto przejście z modelu obiektowego do projektu bazy danych jest bardzo uproszczone. W

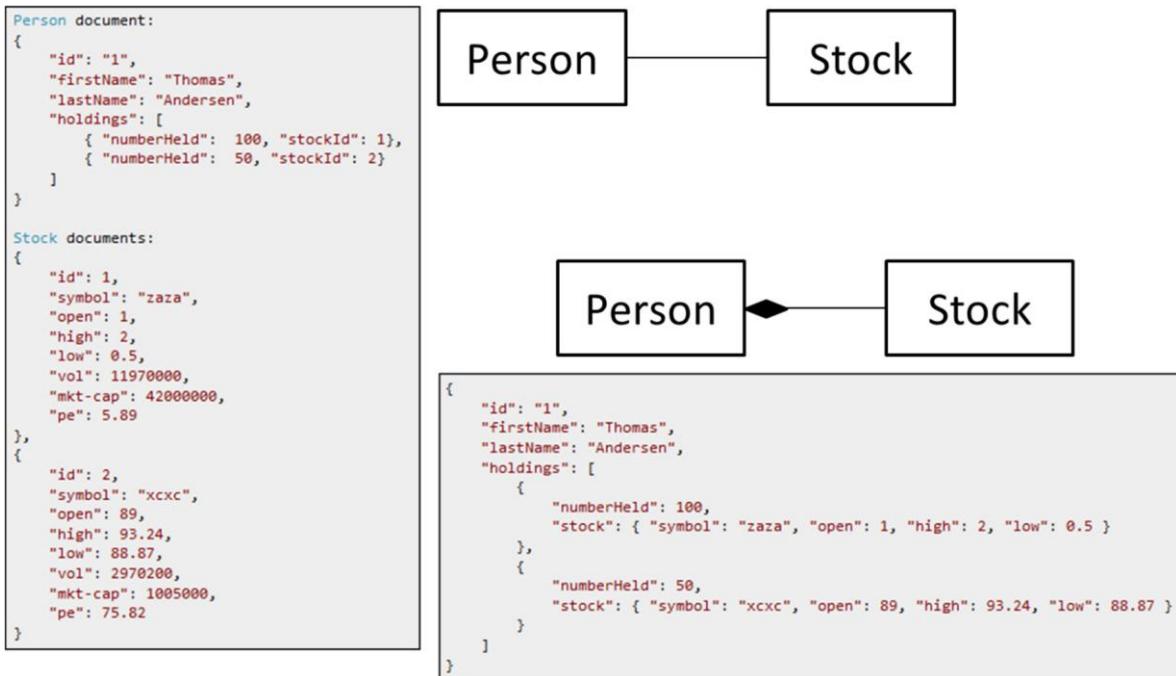
większości przypadków struktura bazy danych w pełni odpowiada modelowi danych wyrażonemu np. w postaci diagramów klas w języku UML. Nie jest zatem potrzebna specjalna translacja, jak w przypadku modelu relacyjnego przedstawionego wyżej.

Dokumenty przechowywane w bazie NoSQL są zapisywane w określonym formacie tekstowym, najczęściej jest to format JSON (JavaScript Object Notation). Podstawową zasadę działania baz NoSQL ilustruje rysunek 10.14. Kod aplikacji przesyła do bazy pliki w odpowiednim formacie. Pliki te są umieszczane w dokumentach zawartych w odpowiedniej kolekcji. Każda kolekcja grupuje dokumenty tego samego typu. Dostęp do dokumentów uzyskujemy poprzez formułowanie odpowiednich zapytań. Zapytania mogą być formułowane w języku zbliżonym do języka SQL lub w inny sposób, specyficzny dla danej bazy danych. Wynikiem zapytania jest odpowiedni dokument lub dokumenty spełniające zapytanie.



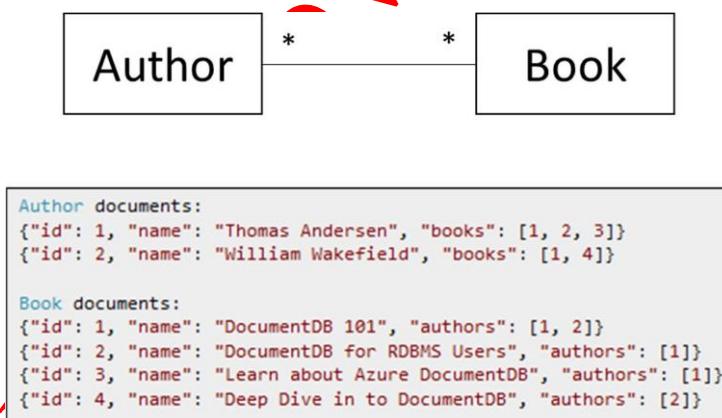
Rysunek 10.14: Podstawy działania baz danych nierelacyjnych

Przykładowa struktura dokumentów w bazie NoSQL przedstawiona jest na rysunkach 10.15 i 10.16. W pierwszym przypadku widzimy realizację krotności „1-1” dla zwykłej agregacji oraz dla kompozycji. W przypadku relacji asocjacji zostały zaprojektowane dwie kolekcje – jedna dla dokumentów klasy „Person” (osoba), a druga dla dokumentów klasy „Stock” (akcje giełdowe). Informacja o tym, jakie akcje posiada dana osoba zawarte są w dokumentach klasy „Osoba”. Zawierają one identyfikator akcji („stockId”) oraz liczbę posiadanych sztuk („numberHeld”). W przypadku relacji kompozycji zaprojektowany został tylko jeden typ dokumentów. Przechowuje on informacje o osobach oraz posiadanych przez nich akcjach.



Rysunek 10.15: Zależność struktury bazy nierelacyjnej od modelu danych – relacje „1-1”

Rysunek 10.16 ilustruje realizację relacji „N-N”. Jak widzimy, realizacja jest bardzo prosta. Polega ona na umieszczeniu w obydwu dokumentach wzajemnych referencji. W naszym przykładzie, w dokumentach klasy „Author” (autor) umieszczona jest lista referencji (identyfikatorów) do dokumentów klasy „Book” (książka). Podobnie, w dokumentach klasy „Książka” umieszczono referencje do dokumentów klasy „Authoir”. Przypomina to implementację tego typu sytuacji w kodzie obiektowym (lista referencji).



Rysunek 10.16: Zależność struktury bazy nierelacyjnej od modelu danych – relacje „N-N”

Jak widzimy, paradygmat baz danych nierelacyjnych dosyć dobrze odpowiada paradygmatowi obiektowemu. Oznacza to uproszczenie kodu dostępu do bazy danych, a tym samym skrócenie czasu na implementację. Bazy danych NoSQL są jednak zazwyczaj mniej wydajne niż bazy relacyjne. Dlatego też w wielu zastosowaniach nadal przeważa wykorzystanie baz relacyjnych.

Zadania

Zadanie 1

Zaprojektuj komponent warstwy logiki aplikacji realizujący zadany scenariusz przypadku użycia. Narysuj diagram klas zawierający klasę logiki aplikacji dla zadanego przypadku użycia oraz odpowiednie interfejsy komunikacji z warstwami widoku i logiki dziedzinowej.

Przypadek użycia: „Pokaż kokpit obliczeniowy”.

Scenariusz: 1.Użytkownik wybiera „Pokaż kokpit”. 2. System pobiera listę Zadań Obliczeniowych dla Użytkownika. 3. System pokazuje „Okno kokpitu obliczeniowego”.

Zadanie 2

Proszę narysować diagram sekwencji dla komponentu logiki aplikacji, realizujący zadany scenariusz przypadku użycia. Proszę przyjąć scenariusz podany w treści zadania 1.

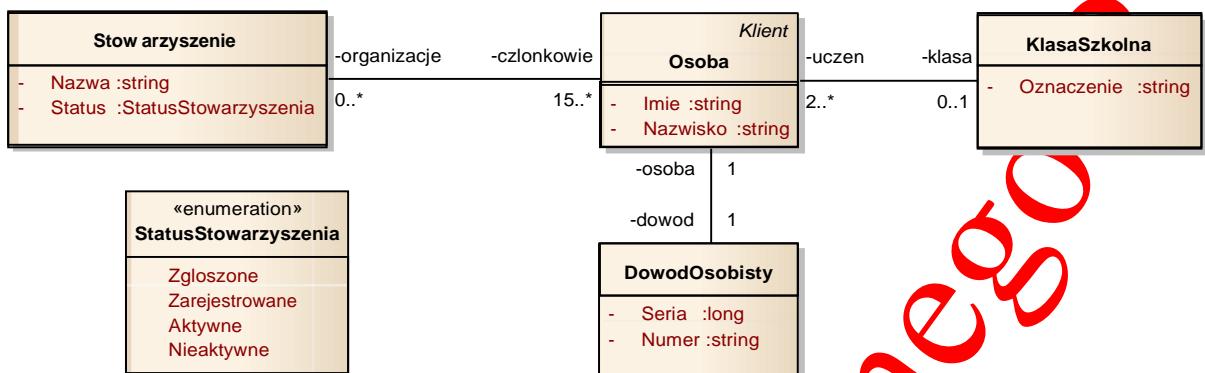
Zadanie 3

Uzupełnij diagram sekwencji z rysunku 10.4 o reakcję na rezultat wykonania operacji „zapiszModelSam”. W przypadku kiedy zapis się powiedzie, należy wyświetlić komunikat sukcesu, a kiedy się nie powiedzie – komunikat błędu.

Do użytku wewnętrznego SO PW

Zadanie 4

Zaprojektuj strukturę relacyjnej bazy danych dla zadanego modelu dziedziny. Narysuj odpowiedni diagram związków encji.



Do użytku wewnętrznego

Słownik pojęć

Kod „brudny”

Kod zapisany w danym języku oprogramowania, który jest zależny od konkretnej technologii (np. szkieletu technologicznego) i w konsekwencji – podatny na zmiany technologii.

Kod „czysty”

Kod zapisany w danym języku oprogramowania, który jest zależny jedynie scenariuszów interakcji użytkowników z systemem. Kod taki jest niezależny od technologii (np. szkieletu technologicznego) i stosuje jedynie tzw. „czyste” klasy (klasy POJO/POCO), czyli klasy realizujące wyłącznie logikę aplikacji i logikę dziedzinową.

Model dziedziny

Wzorzec projektowania logiki dziedzinowej, w którym logika jest rozproszona między klasy zgodne z modelem opisującym daną dziedzinę problemu.

Nierelacyjna baza danych

Technologia przechowywania danych, która nie korzysta z teorii relacyjnej, inaczej nazywana technologią NoSQL. Tego typu baza na ogół nie wykorzystuje języka SQL, lub jest on traktowany pomocniczo. Podstawową zasadą wielu baz danych NoSQL jest przechowywanie danych w postaci dokumentów. W większości przypadków struktura bazy danych w pełni odpowiada obiektowemu modelowi danych.

Relacyjna baza danych

Technologia przechowywania danych ustrukturalizowanych oparta na teorii relacyjnej. Podstawowym językiem umożliwiającym dokonywanie operacji na danych w bazach relacyjnych jest język SQL. Struktura relacyjnej bazy danych oparta jest na czterech podstawowych konstrukcjach – tabelach, kolumnach, wierszach i powiązaniach.

Szkielet technologiczny (framework)

Zestaw komponentów oraz bibliotek kodu, który definiuje ogólną strukturę systemu oprogramowania oraz mechanizmy jego działania. System oprogramowania tworzony jest poprzez rozbudowę i dostosowywanie komponentów szkieletu architektonicznego do wymagań.

Skrypt transakcyjny

Wzorzec projektowania logiki dziedzinowej, który polega na organizacji całej logiki w pojedyncze transakcje realizowane w sposób proceduralny. Wzorzec ten jest najczęściej stosowany podczas realizacji prostych operacji zapisywania oraz odczytywania danych oraz prostych obliczeń możliwych do obsługi w jednej procedurze.

Co trzeba zapamiętać

Projektowanie frontendu

Komponenty warstw prezentacji i logiki aplikacji tworzą zazwyczaj spójną całość, nazywaną frontendem. Głównym zadaniem podczas projektowania frontendu jest zaprojektowanie elementów

odpowiedzialnych za obsługę interfejsu użytkownika oraz logikę nawigacji między tymi elementami. Bardzo istotne jest również zaprojektowanie współpracy z warstwą logiki dziedzinowej. Projektując frontend powinniśmy wyraźnie wydzielić te elementy, które są silnie zależne od technologii interfejsu użytkownika oraz technologii wymiany danych. Bardzo wskazane jest dokonanie podziału na kod „brudny” i kod „czysty”. Klasy „brudne” są najczęściej zależne od konkretnego szkieletu technologicznego (ang. framework).

Projektowanie logiki dziedzinowej

Logika dziedzinowa może być umieszczona w tym samym węźle wykonawczym i być ściśle powiązana z elementami logiki aplikacji lub być dostępna zdalnie jako np. usługa webowa. Zasadnicza funkcjonalność logiki dziedzinowej zazwyczaj projektowana jest jako zestaw niezależnych komponentów, często traktowanych jako usługi. Dla każdego takiego komponentu projektowany jest jeden lub więcej interfejsów (fasad), zawierających odpowiednie zestawy operacji na obiektach dotyczących danej dziedziny problemu. Projekt kodu logiki dziedzinowej może być oparty na określonych wzorach, np. jako skrypt transakcyjny lub model dziedziny.

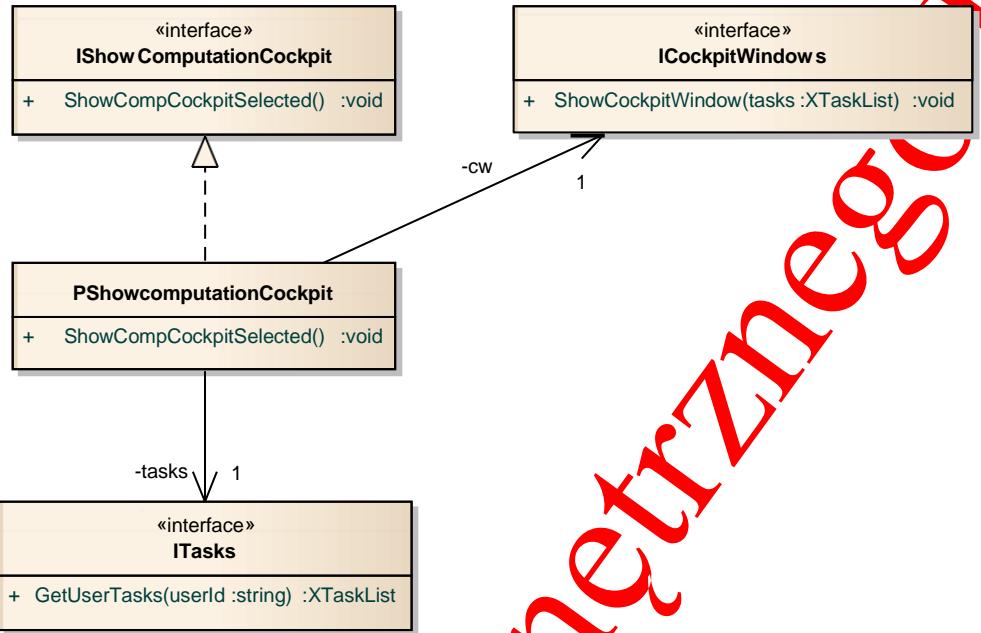
Projektowanie bazy danych

Komponenty warstwy przechowywania danych mogą być realizowane w różnych technologiach i mogą przechowywać dane różnego rodzaju w różnych formatach i strukturach. Najczęściej używanymi paradygmatami przechowywania danych są bazy danych relacyjne oraz bazy danych nierelacyjne. Projektowanie baz relacyjnych polega przede wszystkim na opracowaniu struktury bazy danych składającej się z tabel, kolumn, wierszy i powiązań. Podstawą takiego projektu może być model dziedzinowy, który należy przekształcić do modelu relacyjnego. Bazy danych nierelacyjne najczęściej są oparte na strukturach danych zgodnych z paradigmatem obiektowym, zatem nie wymagają przekształcania modelu dziedzinowego wyrażonego np. jako model klas.

Rozwiązania zadań

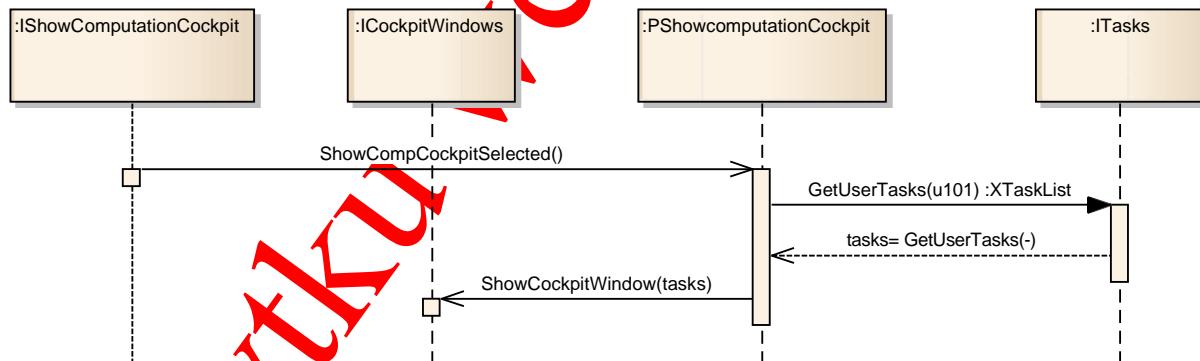
Rozwiązanie zadania 1

Diagram klas komponentu realizującego przypadek użycia „Pokaż kokpit obliczeniowy”.



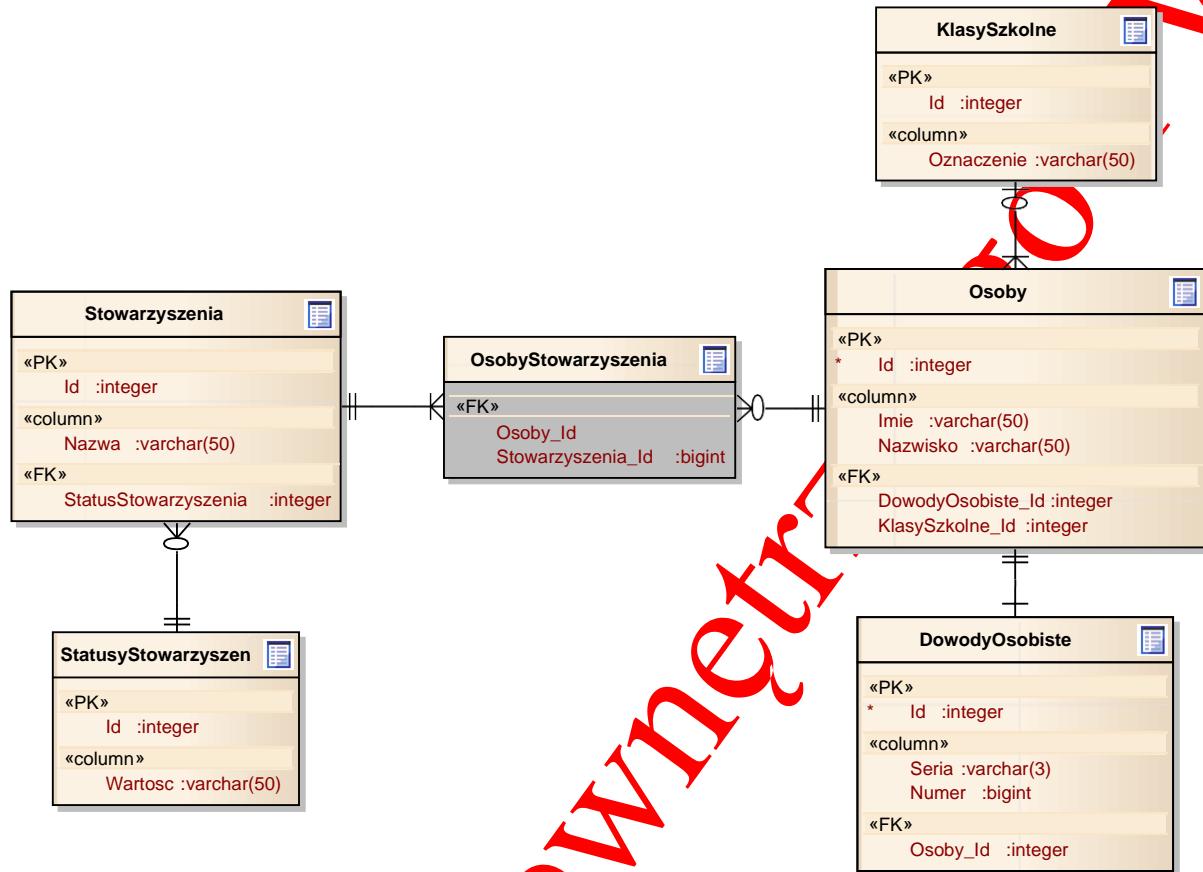
Rozwiązanie zadania 2

Diagram sekwencji opisujący działanie logiki aplikacji dla przypadku użycia „Pokaż kokpit obliczeniowy”.



Rozwiązanie zadania 4

Model związków encji dla zadanej dziedziny.

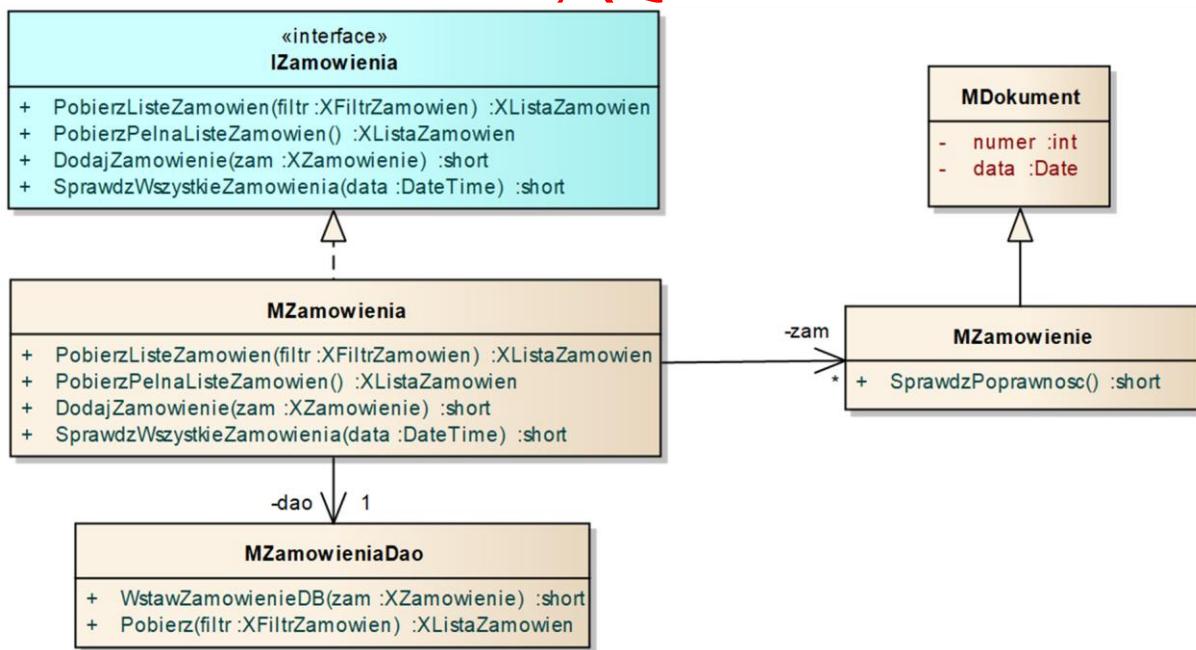


11. Podstawy implementacji oprogramowania

11.1 Kodowanie systemu na podstawie projektu

Implementacja systemu informatycznego oznacza realizację przez programistów założeń projektowych zgodnych w wymaganiami zamawiającego. Celem implementacji jest zapisanie struktury i funkcjonalności systemu w wybranym języku (lub językach) programowania przy wykorzystaniu wybranego środowiska implementacyjnego. Podstawowym zadaniem programisty jest zatem stworzenie kodu, który będzie zgodny z projektem stworzonym przez architekta i projektanta kodowanych komponentów. Tworzony kod powinien wypełniać ramy określone przez modele komponentów, dostarczając ich odpowiedniej funkcjonalności.

Przeniesienie projektu systemu opisanego modelami projektowymi do kodu nazywane jest **inżynierią w przód** (ang. forward engineering). Najbardziej typowym zadaniem jest tutaj przekształcenie modelu klas zapisanego w języku UML w kod zapisany w wybranym języku programowania. Przekształcenie takie powinno być czysto mechaniczne, zgodnie z semantyką języka UML. W niektórych sytuacjach konieczna jest jednak decyzja projektanta lub programisty. Dotyczy to na przykład decyzji odnośnie rodzaju struktur danych stosowanych do przechowywania referencji. Inżynierię w przód omówimy na przykładzie modelu pokazanego na rysunku 11.1. Widzimy tu fragment modelu klas stanowiącego projekt realizacji jednego z interfejsów komponentu warstwy logiki dziedzinowej.



Rysunek 11.1: Przykładowy projekt struktury komponentu

Po zastosowaniu odpowiednich reguł translacji, otrzymamy kod pokazany poniżej. Dla ustalenia uwagi pokazany jest kod interfejsu oraz dwóch klas, który ilustruje najważniejsze reguły. Kod został wygenerowany automatycznie w narzędziu CASE (patrz ostatni rozdział niniejszego modułu) i jest zapisany w języku C#. Dla zwięzości, pominięto kod dwóch operacji klasy „MZamowienia”.

```

///////////
// Implementation of the Interface IZamowienia
///////////

public interface IZamowienia {
    XListaZamowien PobierzListeZamowien(XFiltrZamowien filtr);
    XListaZamowien PobierzPelnaListeZamowien();
    short DodajZamowienie(XZamowienie zam);

}//end IZamowienia

///////////
// Implementation of the Class MZamowienia
///////////

public class MZamowienia : IZamowienia {

    private List<MZamowienie> zam;
    private MZamowieniaDao dao;

    public MZamowienia(){
    }

    public XListaZamowien PobierzListeZamowien(XFiltrZamowien filtr){
        return null;
    }

    public XListaZamowien PobierzPelnaListeZamowien(){
        return null;
    }

    // (...) pozostałe metody pominięte

}//end MZamowienia

///////////
// Implementation of the Class MZamowienie
///////////

public class MZamowienie : MDokument {

    public MZamowienie(){

    }

    public short SprawdzPoprawnosc(){
        return 0;
    }

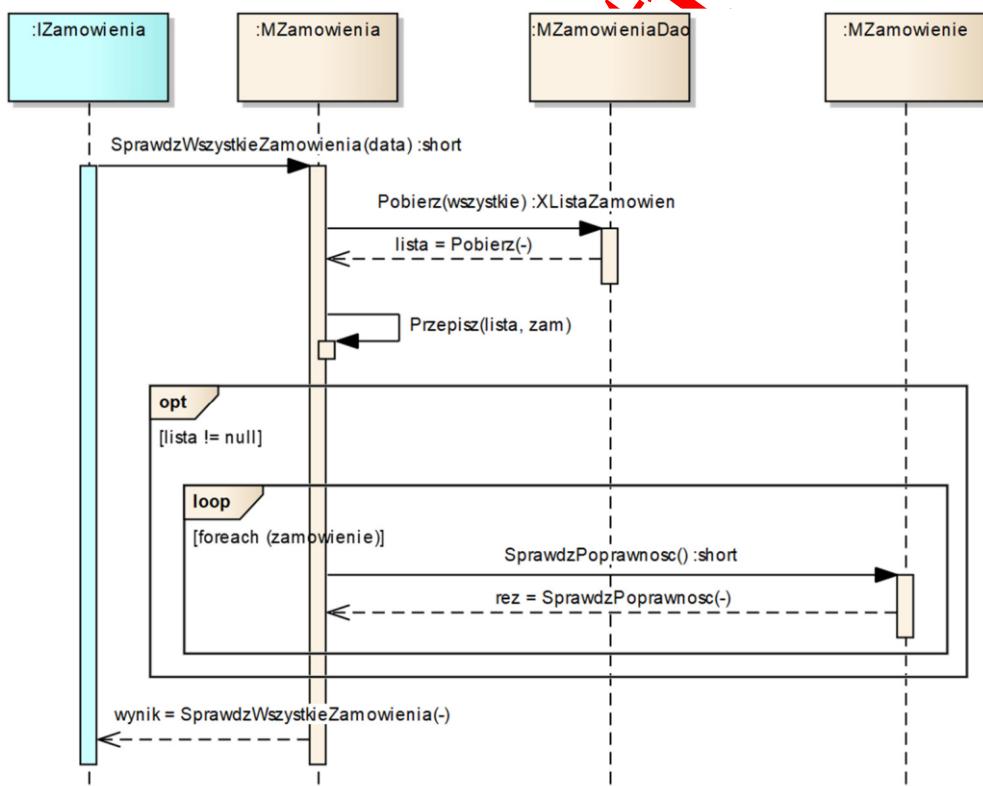
}//end MZamowienie

```

Jak widzimy, kod zawiera całą strukturę poszczególnych elementów kodu, zgodną z projektem. Notacja sygnatur operacji języka UML została odpowiednio zamieniona na notację sygnatur metod języka C#. Wygenerowane zostały standardowe konstruktory klas. Relacje realizacji interfejsów oraz generalizacji zostały zamienione na odpowiednie oznaczenia implementacji i dziedziczenia (nazwa elementu ogólnego poprzedzona znakiem „:”). Warto zwrócić uwagę na implementację w kodzie relacji asocjacji. Relacje te w naszym przykładzie są relacjami nawigowalnymi w jedną stronę. Ponadto, nawigowalne

końce tych asocjacji zostały oznaczone krotnościami i nazwami ról. W rezultacie, w kodzie klasy „MZamowienia” zostały utworzone dwa pola (atrybuty) – „zam” i „dao”, posiadające typy odpowiednich klas. Zauważmy, że pole „zam” zostało wygenerowane jako lista. Wynika to z tego, że źródłowa asocjacja posiada krotność „*” (wiele). Istnienie tak wygenerowanych pól pozwala obiektom klasy „MZamowienia” odpowiadać się do obiektów klas „MZamowienie” i „MZamowieniaDao”. Nie jest to możliwe w odwrotnym kierunku – klasy te (patrz kod klasy „MZamowienie”) nie posiadają analogicznych pól. Wynika to z tego, że brak jest nawigowalności przykładowych asocjacji w drugą stronę.

Generacja kodu najczęściej pozwala na znaczną parametryzację – można określić nie tylko język programowania dla wytwarzanego tak szkieletu kodu, ale także mapowanie typów UML na typy specyficzne dla języków programowania, sposób realizacji list, tablic, strategie przetwarzania elementów oznaczonych stereotypami czy też interpretację relacji agregacji i kompozycji. Inżynieria w przód dotyczy najczęściej modeli statycznych języka UML. Oznacza to, że kod wygenerowanych metod jest pusty – zawiera jedynie standardowe instrukcje powrotu („return”). Jest to oczywiste, gdyż źródłowy model klas nie zawiera informacji, które pozwoliłyby taki kod wygenerować. Dynamika działania kodu (treść metod) jest opisywana innymi modelami. Informacje zawarte w modelach dynamiki programisty może wykorzystać do napisania istotnych części kodu metod. Przykładem jest model sekwencji. Na rysunku 11.2 widzimy diagram zawierający kilka podstawowych konstrukcji tego modelu.



Rysunek 11.2: Przykładowy projekt działania operacji interfejsu

Na podstawie tego diagramu, programista może stworzyć kod metody „SprawdzWszystkieZamowienia” w klasie „MZamowienia”, który został pokazany poniżej. Kod ten odpowiada belce operacji zawartej między komunikatem synchronicznym „SprawdzWszystkieZamowienia(data)”, a odpowiadającym mu komunikatem zwrotnym na dole diagramu.

```
public short SprawdzWszystkieZamowienia(DateTime data)
{
    short wynik = 0;
    lista = dao.Pobierz(wszystkie);
    Przepisz(lista, zam);
    if (null != lista)
    {
        foreach (MZamowienie zamowienie in zam)
        {
            wynik = zamowienie.SprawdzPoprawnosc();
        }
    }
    return wynik;
}
```

Jak wynika z diagramu sekwencji, w ramach realizacji metody, obiekt klasy „MZamowienia” na początku przesyła komunikat synchroniczny „Pobierz” do obiektu typu „MZamowieniaDao”, po czym otrzymuje od niego odpowiedź (komunikat zwrotny). W kodzie odpowiada temu wywołanie procedury „Pobierz” na obiekcie „dao” (patrz kod klasy „MZamowienia”) i umieszczenie wyniku w zmiennej „lista”. Następnie, wywoływana jest prywatna metoda „Przepisz”, co odpowiada „zawiniętemu” komunikatowi o tej samej nazwie. Poniżej widzimy dwie zawarte w sobie ramki („opt” i „loop”). Zewnętrzna ramka odpowiada instrukcji „if”, a wewnętrzna – odpowiedniej instrukcji pętli (tutaj – pętla „foreach”).

Oczywiście, nie cały kod metod może być mechanicznie utworzony na podstawie diagramów sekwencji. Deklaracje zmiennych lokalnych, operacje przetwarzania danych (np. obliczenia) czy też konkretne wyrażenia logiczne muszą być napisane przez programistę. Należy także dodać odpowiednią obsługę błędów (wyjątki) i inne niezbędne elementy zależne od wybranej technologii i języka programowania. Ponadto, zazwyczaj diagramów sekwencji nie tworzymy dla wszystkich metod. Byłoby to zbyt uciążliwe i trudne do zarządzania. Dlatego też, diagramy opisujące dynamikę są najczęściej stosowane jako wskaźówki (wzorce) w zakresie tworzenia treści metod.

Dobrą praktyką implementacji jest utrzymywanie jej zgodności z modelami projektowymi. W ten sposób, programiści mają stale aktualną „mapę kodu”, co zwiększa efektywność pracy programistów. Często jednak programiści wprowadzają zmiany projektowe bezpośrednio w kodzie. Daje to możliwość szybkiego sprawdzenia efektywności konkretnych rozwiązań. Dotyczy to np. zmiany sygnatur operacji, dodawania nowych operacji, dodawania nowych klas itp. Każda taka zmiana w kodzie powoduje dezaktualizację modelu projektowego (np. modelu klas czy modelu komponentów). Aby zachować zgodność, można zastosować tzw. **inżynierię odwrotną** (ang. reverse engineering). Polega ona na odtworzeniu modelu (najczęściej – modelu klas) na podstawie kodu. Oczywiście, mechanizmy inżynierii odwrotnej, podobnie jak inżynierii w przód, są wbudowane w odpowiednie narzędzia CASE i są wykonywane automatycznie. Narzędzie automatycznie aktualizuje model na podstawie zmian dokonanych w kodzie. Warto również zauważyć, że wykorzystanie inżynierii odwrotnej jest jeszcze szersze. Dzięki niej możemy stworzyć „wizualną mapę” dla kodu, który nie posiada dokumentacji w postaci modelu klas. Często jest to tzw. **kod odziedziczony** (ang. legacy code), który został napisany wiele lat temu i dla którego brakuje dokumentacji. Inżynieria odwrotna pozwala w prosty sposób stworzyć dokumentację i poprawić możliwości analizy takiego kodu.

Praca nad kodem jest działaniem bardzo dynamicznym. Programy są tworzone przyrostowo, a także podlegają wielu zmianom wynikającym ze zmieniającej się technologii, czy też zmieniających się wymagań. W trakcie rozwoju systemu oprogramowania pojawia się konieczność wielokrotnego generowania kodu i jednoczesnego uzupełniania modelu zmianami pochodząymi z kodu. Odpowiedni poziom synchronizacji między kodem a modelem uzyskujemy przez zastosowanie cyklu łączącego

inżynierię w przód i inżynierię odwrotną. Cykl taki nazywamy **inżynierią okrężną** (ang. *round-trip engineering*).

11.2. Dobre praktyki w zakresie kodowania

Programista, tworząc kod porusza się w ramach określonych przez modele projektowe. Jednocześnie, modele projektowe są zazwyczaj aktualizowane równolegle z tworzeniem nowego kodu. Dobrze wykonany, dobrej jakości projekt kodu znacznie ułatwia pracę programisty. Jednoczenie jednak, bardzo istotne jest zapewnienie jakości samego kodu. Projekt stanowi dobrą dokumentację kodu, poprzez dostarczenie wizualnej „mapy” struktury kodu oraz opisanie wybranych elementów dynamiki działania kodu. Dobre praktyki dotyczące projektowania oprogramowania zostały przedstawione w poprzednich rozdziałach. Tutaj skoncentrujemy się na zaprezentowaniu dobrych praktyk w zakresie kodowania, czyli wypełniania treścią zaprojektowanych wcześniej procedur (metod klas). Przy tym, należy podkreślić, że dobre praktyki kodowania często pokrywają się z dobrymi praktykami projektowania. Zatem, zawarte tutaj wskazówki można również zastosować podczas tworzenia modeli projektowych.

Kod, tak jak inne produktu procesu wytwarzania oprogramowania, jest najczęściej tworzony w zespołach. Jest on zatem czytany przez osoby o różnych kwalifikacjach, doświadczeniu i nawykach. Dobre praktyki programowania zostały wypracowane w wyniku doświadczenia wielu lat pracy różnych zespołów programistów. Praktyki te w obiektywny sposób opisują pewne pożąданie cechy kodu – niezależnie od języka programowania, rozwiązań technicznych, charakteru organizacji informatycznej czy dziedziny projektu, w obrębie którego kod powstawał.

Podstawową przesłanką, którą każdy programista powinien uwzględniać podczas pracy z kodem jest fakt, że nie pisze on kodu dla siebie. Troska kodera o przyszłego czytelnika objawia się w szeroko rozmianej łatwości przyswajania znaczenia kodu. Kod powinien być czytelny i zrozumiały przy minimalnym nakładzie pracy, a także, na ile to możliwe, prosty do zmiany dla kompetentnej osoby w obrębie danej organizacji czy środowiska. Niestety, wielu programistów tworzy kod nadmiernie skomplikowany, stosując różnego rodzaju „sztuczki programistyczne”, które bardzo utrudniają współpracę w zespole. Co więcej, często takie „sztuczki” obracają się przeciwko im autorom, którzy po kilku miesiącach wracają do swojego kodu i mają problem w jego zrozumieniu. Konserwacja (aktualizacja, poprawa błędów) takiego kodu jest często na tyle złożona, że łatwiej i taniej jest napisać go od początku. Jak wynika z powyższego, kodu nie pisze się ani dla siebie, ani dla swoich obecnych współpracowników. Kod powstaje aby realizować założenia projektu w taki sposób, aby w przyszłości dowolna osoba (w tym – również autor) mogła z nim komfortowo pracować.

Dobre praktyki kodowania można podzielić na praktyki redaktorskie, merytoryczne oraz związane z nawykami codziennej pracy. Praktyki redaktorskie dotyczą pracy z kodem traktowanym jako tekst. Obejmują one kwestie dotyczące formatowania tekstu, jego komentowania oraz konwencji nazowniczych.

- *Formatowanie tekstu.* Kod powinien zawierać wcięcia i odstępy organizujące go w logiczne bloki. Linie nie powinny być zbyt długie, aby można je było łatwo objąć wzrokiem. W przypadku wystąpienia długich wyrażeń logicznych, należy je odpowiednio załąmać poprzez kilka wierszy.
- *Komentowanie kodu.* Komentowanie jest ważne, gdyż często kod nie jest wystarczająco przejrzysty. Może na przykład zawierać pozornie niejednoznaczne odwołania, czy też mieć nieewidentne motywacje. Komentarze mogą występować w kodzie na wielu poziomach abstrakcji. Mogą służyć nakreśleniu roli modułu (to jednak może być opisane w modelu projektowym), opisaniu szczegółów kontraktów i sygnatur dla operacji, a także wyjaśnieniu działania

konkretniej operacji. Współczesne środowiska programistyczne oferują możliwości rozróżniania różnych typów komentarzy. Komentarze wyjaśniające działanie kodu w sposób ogólny, mają często swoją własną składnię, która pozwala na generowanie dokumentacji. Przykładem są odpowiednie skladnie komentarzy dla języka Java (format JavaDoc) oraz C# (znaczniki w formacie XML). Odpowiednie sformatowanie komentarzy pozwala na automatyczne przetwarzanie w celu wygenerowania dokumentacji, która może być np. widoczna podczas szybkiego przeglądania nagłówków metod klas. Komentarze lokalne dotyczą jednego lub kilku wierszy kodu (instrukcje przepływu sterowania, przypisania i wywołania procedur/funkcji itd.). Mogą one mieć postać pojedynczych wierszy lub postać dłuższych bloków tekstu. Należy umiejętnie korzystać z obu form. Sama zawartość komentarzy powinna w sposób wystarczający i zwięzły wyjaśniać opisywany fragment kodu. Informacja taka powinna być możliwie kompletna, ale unikająca oczywistości – programista, jak pisarz, nie powinien nudzić czytelnika i marnować jego czasu. Za każdym razem powinniśmy odpowiadać sobie na pytanie „czy ten kod jest oczywisty (dla mnie/dla innych)?”. Dobrym sposobem na poprawienie lub uściślenie stylu pisania komentarzy, jest wracanie do nich po jakimś czasie. Wskazówką do poprawy własnego stylu komentarzy jest sytuacja, gdy powrocie do pracy nad danym kodem po dłuższym czasie, jego zrozumienie sprawia nam trudność.

- *Konwencje nazewnictwa.* Każdy język programowania, organizacja, a często nawet konkretne projekty mają ustalone konwencje nazewnictwa dla klas, zmiennych, funkcji itd. Konwencje te mają na celu ujednolicenie nazw stosowanych w kodzie, przy zachowaniu ich czytelności i podkreśleniu roli nazw nawet dla najmniej istotnych elementów (takich jak zmienne tymczasowe). Każdy programista powinien zapoznać się z obowiązującymi go konwencjami, a także przyjąć do wiadomości ogólne zasady nazewnictwa w środowiskach programistów.

Poniższy przykładowy fragment kodu ilustruje stosowanie dobrych praktyk w zakresie praktyk redaktorskich.

```
/// <summary>
/// [GetSupportedResourceRange] Get ResourceRange for the given ComputationRelease.
/// </summary>
/// <param name="releaseUid">The Uid of the ComputationRelease</param>
/// <returns>A single ResourceRange record.</returns>
[HttpGet("range")]
public IActionResult GetSupportedResourceRange([FromQuery] string releaseUid)
{
    try
    {
        var reservationRange = _taskManager.GetSupportedResourceRange(releaseUid);
        if (null == reservationRange)
            return HandleError("Release not found", HttpStatusCode.BadRequest);

        return Ok(new XReservationRange(reservationRange));
    }
    catch (Exception e)
    {
        return HandleError(e);
    }
}
```

Jak widzimy, tekst metody został sformatowany z użyciem odpowiednich wcięć oraz pustych wierszy. Wcięcia wyraźnie podkreślają odpowiednie zagnieżdżenie instrukcji. Puste wiersze oddzielają określone fragmenty metody (tutaj – wyraźnie oddzielona instrukcja powrotu z procedury). W tekście zachowano określony standard nazewnictwa, przyjęty dla zastosowanego języka programowania (tutaj – C#). Na przykład – zmienne lokalne pisane są z małej litery w tzw. notacji „camel case” (np. „reservationRange”). Pola prywatne klasy pisane są ze znakiem podkreślenia na początku (np. „_taskManager”), a procedury publiczne pisane są z wielkiej litery (np. „GetSupportedReservationRange”).

Zwróćmy też uwagę na to, że procedura posiada ogólny komentarz w odpowiedni sposób sformatowany. Komentarz opisuje znaczenie procedury oraz jej parametrów. Formatowanie pozwala na wygenerowanie dokumentacji widocznej dla programistów innych modułów (bez wchodzenia w szczegóły procedury). Brakuje komentarzy lokalnych (wewnątrz kodu procedury), gdyż kod jest prosty i nie wymaga dodatkowego wyjaśniania.

Praktyki merytoryczne dotyczące pisania kodu w dosyć dużym stopniu pokrywają się z dobrymi praktykami w zakresie projektowania. Często jednak, w trakcie pisania kodu okazuje się, że konieczne jest dokonanie modyfikacji projektu. Praktyki merytoryczne obejmują zasady strukturalizacji kodu, definiowania zmiennych i stałych, zasad przetwarzania danych oraz testowalności.

- *Projektowanie podczas kodowania.* Projekt oddawany w ręce programistów nie jest „martwy”. Programiści podczas kodowania poszczególnych elementów komponentów mogą odkrywać różnego rodzaju problemy wymagające dokonania poprawek w projekcie. Dotyczy to np. konieczności dokonania dalszej dekompozycji klas, podziału interfejsów, rozbicia zbyt skomplikowanych metod na mniejsze, wydzielenia procedur w celu unikania powielania kodu, itd. Pamiętajmy, że wprowadzając zmiany w kodzie, powinniśmy jednocześnie nanosić je w modelu projektowym, stosując zasady inżynierii odwrotnej.
- *Zapewnienie parametryzowalności kodu.* Wszystkie używane w kodzie stałe (napisy, liczby, symbole itp.) powinny być definiowane poza kodem, który się do nich odwołuje. Dobrą praktyką jest centralizacja definiowania stałych wykorzystywanych globalnie (np. kodów błędów). Dokonuje się tego w specjalnie do tego przeznaczonych plikach nagłówkowych lub klasach. Wszelkie zmiany tych wartości są wtedy łatwiejsze, a czytelność kodu o wiele większa. Praktyką podnoszącą elastyczność kodu i jego separację od stałych jest przenoszenie wartości parametrów konfigurujących wykonanie kodu do plików zewnętrznych. Ich ew. modyfikacje mogą być wtedy wykonywane bez potrzeby rekompilacji, a nawet powtórnego uruchomienia aplikacji.
- *Unikanie stosowania „magicznych wartości”* (ang. *magic numbers*). Zagadnienie to związane jest z omówioną wyżej parametryzowalnością kodu. Należy unikać formuł przekazywanych bezpośrednio w postaci liczb, a nie określonych symbolicznie. Wiąże się to bowiem z obniżeniem czytelności kodu. Osoba czytająca kod nie jest świadoma pełnego znaczenia konkretnej wartości zawartej w kodzie, np. wywołanie:

```
process.setPriority(5);
```

jest o wiele mniej jasne niż:

```
process.setPriority(PROC_PRIORITY_HIGH);
```

gdzie wcześniej zdefiniowano: `PROC_PRIORITY_HIGH = 5`. Unikanie „magicznych wartości” ogranicza też szanse pomyłek związanych z błędami literowymi (np. zamiana „5” na „3” nie będzie w żaden sposób wskazana jako błąd przez kompilator).
- *Posługiwanie się wzorcami projektowymi.* Programista, podobnie do projektanta, często ma możliwość zastosowania ogólnie przyjętych i sprawdzonych rozwiązań. Oprócz wzorców stosowany głównie podczas projektowania, programiści mogą stosować wzorce stosowane lokalnie – bezpośrednio w kodzie procedur. Przykładami takich wzorców są wzorce fabryka abstrakcyjna (ang. Abstract Factory), singleton, komenda (ang. Command), leniwa inicjalizacja (ang. Lazy Initialization) czy konstruktor prywatny (ang. Private Constructor). Dodatkowo, konkretne szkielety technologiczne mogą posiadać własne wzorce projektowania kody. Omówienie szczegółów poszczególnych wzorców jest poza zakresem niniejszych materiałów.
- *Uwzględnianie kodu dla testów.* Moment powstawania kodu dla testów jest zależny od metodyki i procesu twórczego przyjętego w projekcie. Zazwyczaj przyjmuje się, że kod umożliwiający przeprowadzenie testów kodu właściwego powinien powstawać równolegle. Ułatwia to wcześnie wyłapywanie błędów i ewentualne korygowanie rozwiązań projektowych. Dzięki temu, poważne błędy mogą być wykryte na tyle wcześnie, aby móc uniknąć pracochłonnego ich

poszukiwania oraz znacznej przebudowy kodu. Naprawa błędów w zaawansowanym stadium implementacji może bowiem być bardzo kosztowna.

- *Przestrzeganie walidacji parametrów wejściowych.* Tworzony kod powinien „pilnować” kontraktów określonych dla implementowanych przez siebie modułów. Powinien nie tylko zapewniać zwracanie poprawnych danych, ale także walidować wartości parametrów wejściowych. Nigdy nie należy zakładać, że procedura albo funkcja zostanie wywołana z poprawnymi danymi. Walidację należy zacząć od sprawdzania pustych wskaźników i referencji. Należy także kontrolować puste łańcuchy, nie ujęte w słownikach stałych kody, nieprawidłowe sygnały wejściowe itp. Bardzo pomocne są przy tym proste testy automatyczne sprawdzające zachowanie wybranych operacji pod wpływem wadliwych parametrów wejściowych.
- *Przestrzeganie zasad obsługi raportowania wykonania kodu (logowania).* Szczególną uwagę należy przywiązywać do dobrego zdefiniowania oraz przestrzegania reguł raportowania wykonywanych operacji w odpowiednich dziennikach. Dotyczy to m.in. poziomów logowania, formatu wpisów, użytej technologii logowania, a także hierarchii wyjątków (procedur reakcji na błędy czasu wykonania). Należy zdefiniować politykę reakcji na błędy, unikać niepotrzebnego filtrowania wyjątków, a z drugiej strony – zasypywania nimi klas wywołujących. Przestrzeganie tych zasad znacznie ułatwia wykrywanie błędów, w tym już po oddaniu systemu do użytku. Szczegółowe zasady logowania oraz obsługi wyjątków są specyficzne dla konkretnych języków programowania i technologii, dlatego też ich omówienie wykracza poza zakres niniejszych materiałów.
- *Dbałość o dane.* Oprogramowanie często przetwarza dane istniejące od wielu lat. Kod powinien zatem szanować takie „odziedziczone” dane (ang. *legacy data*). W szczególności zbiory danych (np. bazy danych) nie powinny być „zaśmiecone” danymi specyficznymi dla konkretnego systemu. Nieco innym zagadnieniem jest kwestia dostępu do danych w systemach wielowątkowych. Tworzony kod powinien zawierać odpowiednie mechanizmy (np. semafory), które zapewniają zabezpieczenie określonych danych przed ich jednoczesnym odczytem lub zapisem przez wiele wątków jednocześnie.
- *Dbałość o wykorzystanie zasobów maszyn.* Należy pamiętać, że kod będzie pracował w określonym środowisku wykonawczym o ograniczonych zasobach (pamięci, procesorach, sieci). Dlatego też należy przestrzegać odpowiednich metod odzyskiwania pamięci, stosowania przetwarzania równoległego (wielowątkowości) czy optymalizacji kodu dla technologii, z których kod korzysta.
- *Branie pod uwagę przenośności.* Coraz częstszym wymogiem stawianym przez użytkowników jest możliwość uruchamiania aplikacji w różnych środowiskach wykonawczych (np. Android, iOS). Rynek obfituje w technologie i rozwiązania wspierające przenośność. Ich stosowanie wymaga jednak przestrzegania określonych reguł, które czasami są pomijane (z różnych powodów) podczas implementacji. Należy zatem pamiętać, że kod może być uruchamiany na różnych architekturach sprzętowych, pod kontrolą różnych (wersji) systemów operacyjnych itd. O regułach przenośności należy pamiętać szczególnie podczas odwołań do funkcji systemowych i przy dostępie do zasobów zewnętrznych w stosunku do aplikacji.
- *Branie pod uwagę skalowalności.* Skalowalność oznacza możliwość reakcji kodu na różne poziomy obciążenia obliczeniami. Oczywiście, skalowalność zapewniania jest przed odpowiednie rozwiązaniem projektowe. Przykładem jest stosowanie kolejek zadaniowych oraz modułów obliczeniowych z dynamiczną liczbą instancji. Często jednak skalowalność zależy od szczegółowych rozwiązań programistycznych. Kod, który wymaga skalowalności powinien być pisany z myślą, że może on być on wywoływany wielokrotnie w krótkich odstępach czasu. Nawet drobne błędy w tym zakresie mogą powodować problemy stabilności całego systemu. Dobrym rozwiązaniem jest tutaj kooperacja programistów z architektami, którzy powinni informować, które miejsca w systemie będą szczególnie obciążane. Tworzenie odpowiednich testów obciążeniowych dla konkretnych fragmentów kodu pozwoli wcześnie rozwiązać wspomniane problemy.
- *Przestrzeganie zasad paradygmatu obiektowego.* Częstym błędem jest pisanie kodu klas w stylu wywodzącym się z języków proceduralnych. Przykładem jest tutaj zbytne rozbudowywanie

kodu jednej klasy, zamiast dokonania podziału odpowiedzialności między kilka klas. Inny błędem tego rodzaju jest nieuzasadnione korzystanie z pól statycznych nie będących stałymi, co jest odpowiednikiem zmiennych globalnych w językach proceduralnych.

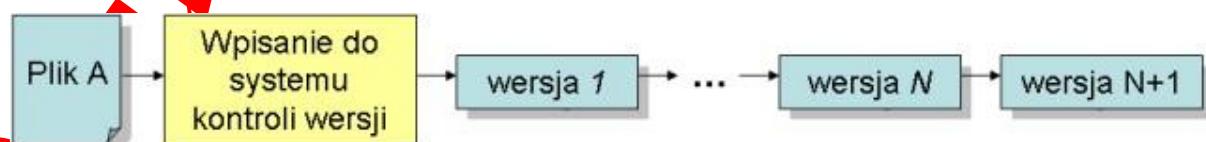
Trzecią grupą dobrych praktyk są praktyki związane z pewnymi nawykami w pracy z kodem. Dotyczą one przede wszystkim pracy zespołowej i przestrzegania zasad wersjonowania kodu.

- *Zapewnienie „czystości” własnego kodu.* Bardzo istotne jest, aby kod podlegający kontroli wersji był „czysty”, czyli pozbawiony błędów komplikacji, ostrzeżeń oraz innych wad utrudniających (lub wręcz uniemożliwiających) innym pracę i testowanie. (np. błędy krytyczne czasu uruchomienia występujące na starcie aplikacji). Każdy programista powinien zbudować od zera projekt i przeprowadzić na nim testy przed zapisaniem go w repozytorium. Zapis w repozytorium powinien posiadać odpowiedni komentarz ułatwiający innym zrozumienie powodu i zakresu wprowadzonych zmian.
- *Praca na aktualnym kodzie.* Przed rozpoczęciem pracy z nowym fragmentem kodu powinniśmy się upewnić, że jest on aktualny. W tym celu należy pamiętać, aby zsynchronizować kod na lokalnej maszynie z kodem znajdującym się w repozytorium. Należy też upewnić się, że zaktualizowana lokalna jego kopia jest „czysta” przez uruchomienie testów jednostkowych.
- *Umiejętnie korzystanie ze środowiska programistycznego.* Dobrym nawykem jest dogłębne poznanie możliwości stosowanego środowiska programistycznego. W wielu sytuacjach, taka wiedza znacznie ułatwia i przyspiesza pisanie kodu, szczególnie kodu powtarzalnego. Bardzo pomocne w pracy są funkcje takie jak auto-uzupełnianie, fragmenty-szablony kodu (ang. *snippets*) gotowe do użycia, czy kontrola składni oraz podpowiadanie możliwej naprawy błędów składowych. Korzystanie z tego typu funkcji i dobrze skonfigurowanego środowiska znacząco podnosi produktywność programisty.

Bardzo istotnym elementem dobrych praktyk programistycznych jest zatem umiejętnie korzystanie z narzędzi. Stosowanie narzędzi w pełnym cyklu inżynierii oprogramowania omawiamy w ostatnim rozdziale niniejszego modułu.

11.3. Zarządzanie wersjami, konfiguracją i zmianami

Zarządzanie wersjami (ang. *version control*) oznacza zapisywanie kolejnych zmian w pliku lub zestawie plików. Każda zapisana zmiana tworzy wersję, do której można wrócić w dowolnym momencie. Zarządzanie zmianami w projektach konstrukcji oprogramowania najczęściej dotyczy plików z kodem źródłowym, lecz może ono być stosowane do dowolnego rodzaju plików (np. plików zawierających modele projektowe, dane binarne itd.). Do zarządzania wersjami stosujemy specjalne narzędzia, które nazywamy **systemami zarządzania wersjami** (ang. Version Control System). Ilustrację podstawowej zasady działania takiego systemu widzimy na rysunku 11.3. Plik zostaje najpierw wstawiony do systemu, a następnie, każda wprowadzona zmiana w treści pliku jest automatycznie zapisywana przez system jako osobna wersja.

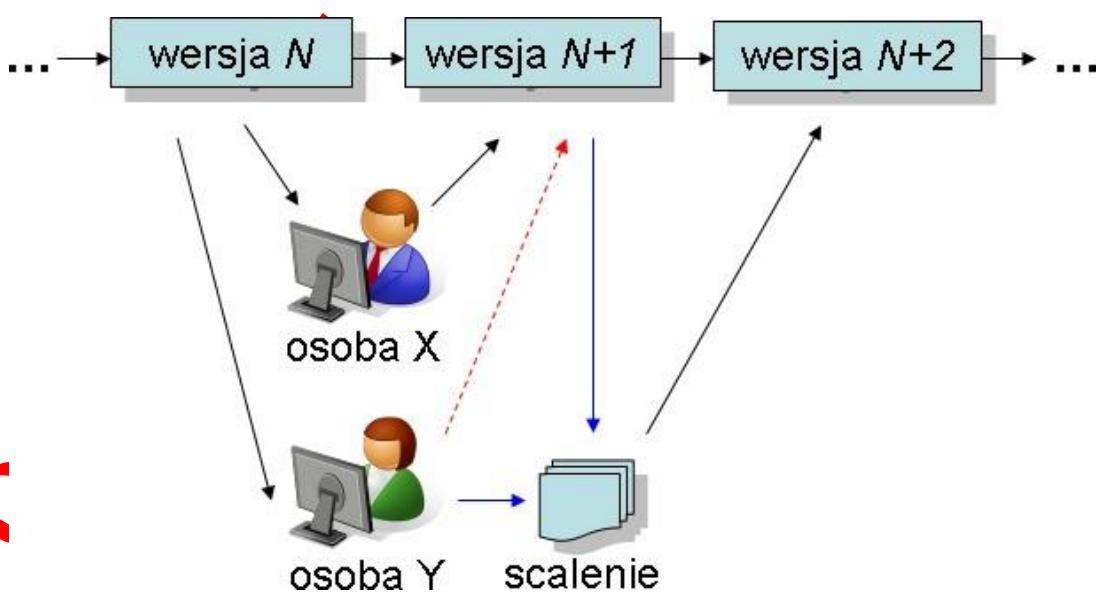


Rysunek 11.3: Kolejne wersje pliku w systemie zarządzania wersjami

Istnieją dwa główne **modele zarządzania wersjami**: scentralizowane (klient-serwer) oraz rozproszone. **Model scentralizowany** polega na ustanowieniu centralnego serwera, na którym przechowywane są wszystkie wersjonowane pliki i dokonywane są wszystkie zapisy zmian w tych plikach. Z serwerem współpracują maszyny klienckie (robocze), na które można pobierać pliki z serwera i po wprowadzeniu zmian – zapisywać je z powrotem na serwerze. Model ten ma wiele zalet i przez wiele lat był on najpowszechniej stosowany przez zespoły budujące oprogramowanie. Podstawową wadą takiego modelu jest duża zależność od dostępności centralnego serwera. W razie awarii, tracimy całą funkcjonalność związaną z wersjonowaniem, a w skrajnych przypadkach możemy stracić treść wersjonowanych plików. Z tego powodu, współcześnie dużą popularność zyskał **model zdecentralizowany**. W tym systemie nadal mamy serwer, który umożliwia przechowywanie oraz wersjonowanie plików. Różnica polega na tym, że maszyny klienckie posiadają możliwość lokalnego wersjonowania plików i przechowują lokalnie kopie całej historii wersji. W ten sposób, każda maszyna lokalna ma możliwość odtworzenia całego zestawu wersjonowanych plików, nawet w razie awarii serwera. Najpopularniejszym obecnie systemem realizującym model zdecentralizowany jest system **Git**. W niektórych projektach stosowane są starsze systemy scentralizowane – Concurrent Version System (**CVS**) i Subversion (**SVN**).

Niezależnie od modelu, zasada działania systemów zarządzania wersjami jest oparta na kilku podstawowych zasadach, które omawiamy poniżej. Podstawowym założeniem działania takich systemów jest to, że nad plikami pracuje zespół osób. W szczególności, osoby te (np. programiści) mogą dokonywać jednocześnie zmian w tych samych plikach. Dlatego też, oprócz prostego zapisywania kolejnych wersji, konieczne jest ustanowienie mechanizmów pracy grupowej.

Istnieją dwie podstawowe strategie zarządzania równoległyymi zmianami w plikach. Pierwsza polega na tworzeniu **blokad** (ang. lock), a druga na wykonywaniu **scaleni** (ang. merge). Pierwsza strategia zakłada możliwość zablokowania przez użytkownika wersjonowanego zasobu, by uniemożliwić innym zapis (stworzenie nowej wersji). Jednocześnie, odczyt jest cały czas dopuszczalny. Strategia polegająca na scaleniach zakłada, że istnieje mechanizm tworzenia spójnego i użytecznego pliku w oparciu o dwie różne jego wersje. Mechanizm scalania ilustruje rysunek 11.4. Przedstawia on sytuację, gdy osoby X i Y pracują jednocześnie na tej samej wersji pliku. Osoba X zapisuje swoje zmiany jako pierwsza. Następnie, zapisz zmiany próbuje osoba Y. Otrzymuje ona wtedy informację o nieaktualności swojego pliku. System sprawdza różnice i umożliwia scalenie zmian i zapis w postaci kolejnej wersji.



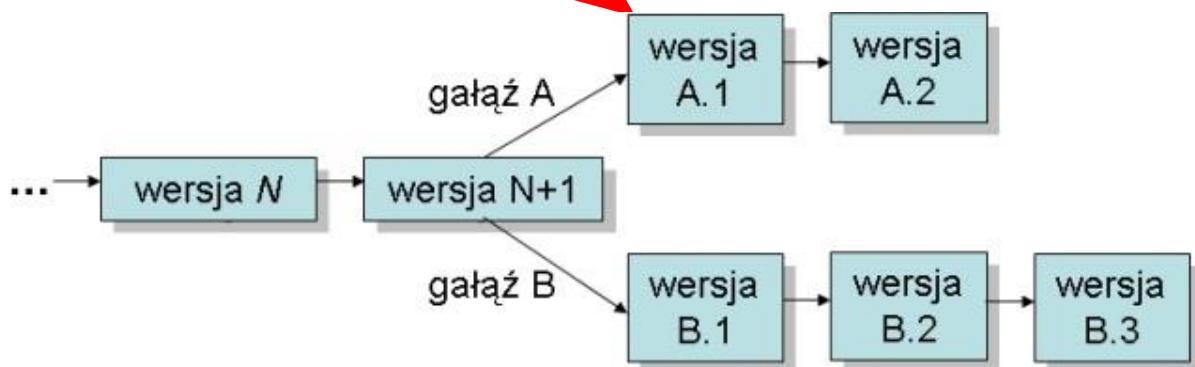
Rysunek 11.4: Scalanie zmian w systemie kontroli wersji

Każdy plik podlegający wersjonowaniu posiada dodatkowy opis, który tworzony jest dla każdej jego wersji. Typowo, opis ten zawiera następujące elementy:

- numer wersji/rewizji (ang. *revision number*) sugerujący kolejność (porządkujący) w historii rozwoju zasobu i/lub projektu; może mieć postać liczbową (np. „275”, „11.2”) lub symboliczną (1.41-alpha)
- identyfikator osoby wprowadzającej zmianę
- datę archiwizacji/zatwierdzenia danego stanu
- krótki słowny opis stanu, posiadający funkcję wskazówki do zrozumienia powodu zaistnienia stanu (może to być tekst typu „podniesienie wydajności funkcji F modułu M” lub „poprawka błędu #T65-342”)

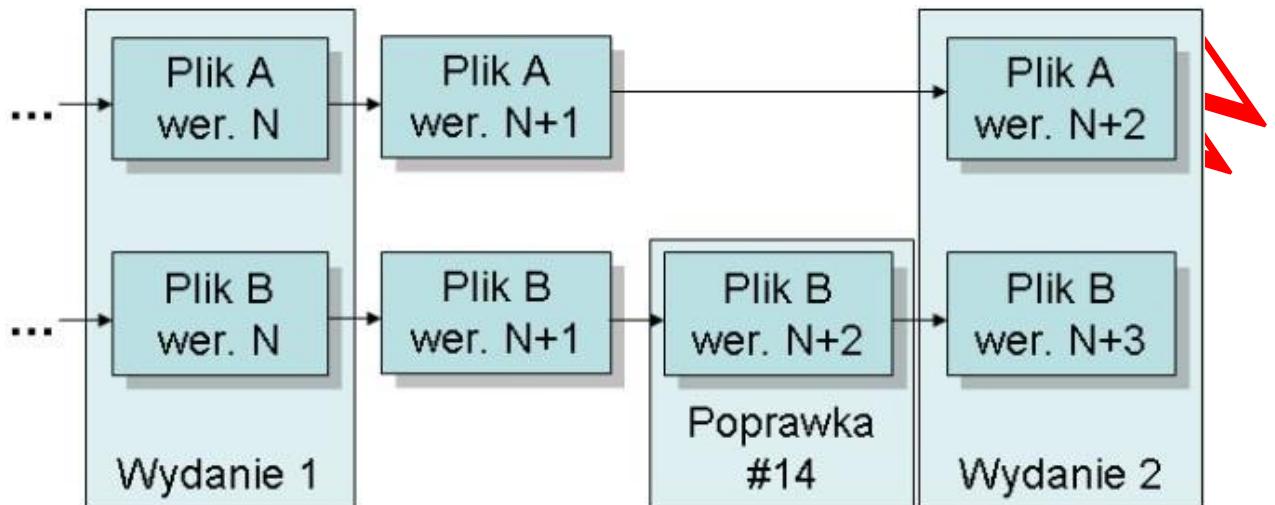
Od strony technicznej każda kolejna rewizja przechowywana jest najczęściej jako tzw. **delta** (zmiana) w stosunku do rewizji ją poprzedzającej. Często delta jest kompresowana w celu dalszego ograniczenia potrzebnych na jej przechowywanie i przesyłanie zasobów. W wyniku równoległej pracy nad plikami wielu osób może powstać wiele różniących się wersji tych samych plików. Najczęściej, pliki dotyczące całego projektu traktowane są jako całość. Kolejne wersje dotyczą wtedy nie pojedynczych plików, lecz całego systemu (zestawu plików).

Praca nad kolejnymi wersjami zasobu (pliku lub zestawu plików) nie musi być liniowa. Różne osoby mogą tworzyć różne warianty składające się z wielu wersji. Takie warianty nazywamy **gałęziami** (ang. branch). Gałęzie mogą być rozwijane całkowicie niezależnie, jednak w pewnym momencie powinny być ze sobą synchronizowane. Podstawą synchronizacji jest gałąź główna, zwana **pniem** (ang. trunk). Jeśli twórca uzna, że jego wariant (gałąź) jest już stabilny, może ją scalić z pniem. Rysunek 11.5 ilustruje tworzenie kilku gałęzi na bazie gałęzi głównej, czyli pnia.



Rysunek 11.5: Schemat wersjonowania dla zasobu o dwóch gałęziach

Wersje zasobów są numerowane zgodnie z konwencją przyjętą w danym systemie zarządzania wersjami. Niezależnie jednak od tego, możliwe jest tworzenie tzw. **znaczników** (ang. tag). Znacznik to symboliczny opis (nazwa) ułatwiający szybki powrót do opisanej tym znacznikiem wersji. W systemach, gdzie każdy plik ma osobną numerację, znacznik może dotyczyć wielu wersji różnych plików. Przykłady znaczników znajdują się na rysunku . Etykieta „Poprawka #14” dotyczy pojedynczej wersji wybranego pliku, a etykiety „Wydanie 1” i „Wydanie 2” odnoszą się do zbiorów plików. Takie „przekrojowe” oznaczanie plików pozwala na odpowiednie dopasowanie do siebie różnych, powstających równolegle elementów systemu (dokumentacja, kod, pliki konfiguracyjne itd.) i pomaga zachować spójność procesu twórczego.



Rysunek 11.6: Znaczniki dla plików w systemie kontroli wersji

Podczas pracy w systemie zarządzania wersjami możemy wykonywać różne operacje umożliwiające równoległą pracę nad różnymi plikami. Pierwszą czynnością jest dokonanie operacji **check-out**, czyli stworzenie lokalnej kopii roboczej, zawierającej aktualną wersję zasobów, nad którymi będziemy pracować. Po dokonaniu zmian, wykonujemy operację **check-in**. W wyniku tej operacji tworzona jest kolejna wersja w aktualnej gałęzi. Ważne jest, aby przez utworzeniem nowej gałęzi dokonać **aktualizacji** (ang. update), czyli pobrać aktualną wersję zasobów z gałęzi głównej. Synchronizacja gałęzi z pnem odbywa się w wyniku operacji **scalania** (ang. merge). Istotnym elementem tej operacji jest pokazanie przez system różnicy między wersjami oraz pomoc w wybraniu zmian, które zostaną włączone do pnia.

Wpisanie zarządzania wersjami w centrum procesu wytwarzania oprogramowania pozwala na usprawnienie pracy zespołów. Zarządzanie wersjami powiązane jest również z **zarządzaniem zmianami**. Zmiany dotyczą każdego systemu informatycznego, niezależnie od fazy w jakiej się znajduje. Są one wynikiem okoliczności związanych ze sposobem rozwoju oprogramowania, a także czynników zewnętrznych: zmian prawnych, organizacyjnych, cyklu koniunkturalnego, potrzeb użytkowników, rozwoju infrastruktury, wyników analizy działania systemu w obrębie wspieranego biznesu itd.

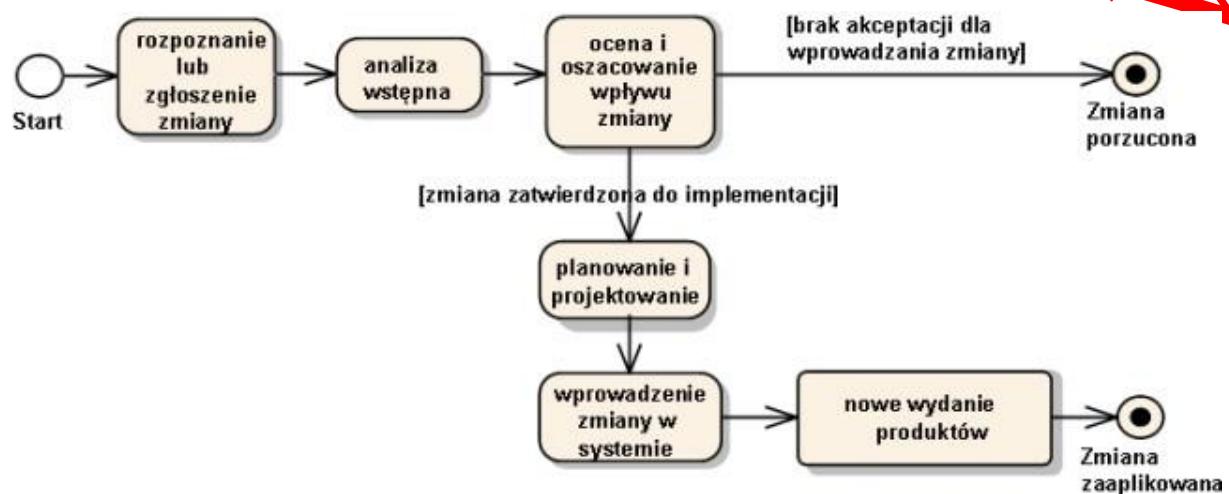
Jak już wielokrotnie wspominano, rozwój oprogramowania jest złożony: w procesie wytwarzczym bierze udział wiele osób o różnych kompetencjach, tworzą one i wymieniają między sobą artefakty różnego rodzaju, opisujące system na wielu poziomach abstrakcji, mających wiele właściwości i znaczeń dodatkowo wpływających na znaczącą liczbę interakcji obserwowanych podczas produkcji systemu. Gdy weźmiemy pod uwagę dodatkową liczbę czynników wynikających ze zmian i pogłębiających poziom komplikacji oprogramowania, odpowiednie zarządzanie zmianami i ich wpływem na proces wytwórczy wydaje się nieodzowne.

W celu sprawnego zarządzania zmianami niezbędne jest wykorzystanie systemu kontroli wersji. Pozwala to na śledzenie historii zmian zasobów powiązanych z projektem, a także utrzymywanie relacji między wersjami plików w repozytorium a dokumentacją zmian. Zmiany nie powinny dotyczyć jedynie kodu, ale również innych produktów pracy – wymagań, modeli projektowych itd. Każda rewizja produktu pracy powinna posiadać szereg atrybutów, np.:

- źródło zmiany,
- opis zmiany,
- opis dodatkowych efektów zmiany (np. wymaganych zmian w innych niż zmieniany komponentach); wartym podkreślenia jest istotność utrzymywania „śladów” (ang. traces) dotyczących propagacji zmian w systemie,
- wyliczenie zaangażowanych w pracę nad zmianą osób bądź ról projektowych,

- atrybuty porządkujące (identyfikator zmiany, data itp.)

Zmiany powinny być wprowadzane w sposób uporządkowany, najlepiej zgodnie z wypracowaną w projekcie procedurą. Przykładowy proces aplikacji przedstawia rysunek 11.7.



Rysunek 11.7: Uproszczony proces aplikacji zmiany w postaci diagramu aktywności

Proces ten składa się z dwóch zasadniczych faz – analizy zmiany i wprowadzenia zmiany. Faza analizy składa się z następujących czynności.

- rozpoznanie lub zgłoszenie zmiany: źródło zmiany może być wiele, najbardziej popularna to prośba zmiany (ang. *change request*) zgłoszana przez klienta,
- analiza wstępna: wstępne ustalenie („zrozumienie” przez zespół analityków) czym jest i czego dotyczy zmiana,
- ocena i oszacowanie wpływu zmiany, jej kosztów itd.: ewaluacja czynników pozamerytorycznych dotyczących zmiany (takich jak uwarunkowania techniczne czy ekonomiczne),

Po analizie podejmowana jest decyzja co do ewentualnej aplikacji zmiany. W przypadku, gdy decyzja jest pozytywna przeprowadzane są następujące czynności.

- planowanie i projektowanie: tworzenie planu wprowadzenia zmiany,
- wprowadzenie zmiany w kodzie: czynność ta obejmuje jedną iterację cyklu życia projektu – stworzenie projektu szczegółowego dot. wprowadzanych zmian, ich bezpośrednią implementację oraz testowanie systemu (regresywne i integracyjne),
- nowe wydanie produktów: „łaty” (ang. *patches*) i aktualizacje dla aplikacji, poprawki dokumentacji, szkolenia i informacja dla klientów itp.

Warto pamiętać, że proces wprowadzania zmian nie dotyczy poprawek, czyli zmian w systemie wynikających z jego wadliwego, niezgodnego z wymaganiami funkcjonowania.

Z zagadnieniem wprowadzania zmian i tworzenia na ich bazie nowych produktów łączy się pojęcie **zarządzania konfiguracją**. Konfiguracja systemu oprogramowania składa się z szeregu produktów procesu wytwórczego, takich jak kod źródłowy, kod wykonywalny, modele projektowe itd. Produkty podlegające zarządzaniu nazywamy elementami konfiguracji. Zbiory elementów konfiguracji tworzą kolejne wersje całego systemu, wśród których wyróżnia się wydania. Wydanie systemu zawiera wszystkie elementy konfiguracji przekazywane klientowi. Jest to wykonywalny kod aplikacji, a także dotyczącego pliki konfiguracyjne, dane z których korzysta program, dokumentacja (użytkownika, techniczna, marketingowa), kod źródłowy oraz programy instalacyjne.

Wprowadzanie zmian w konfiguracji powoduje powstawanie nowych wydań tworzonego systemu oprogramowania. Liczba i terminy oddania do użytku tych wydań mogą być określone w planie projektu i wynikać bezpośrednio ze specyfikacji wymagań (np. przez określenie pewnej liczby docelowych platform sprzętowych czy systemowych). Każdy z takich wydań może wymagać innych wartości parametrów określających jego działanie. Podobna sytuacja występuje dla systemów o budowie modularnej – każdy z modułów może posiadać wiele wariantów i w różny sposób współpracować z pozostałymi komponentami. Procedury i dokumentacja zarządzania konfiguracją powinny regulować i opisywać kolejne warianty systemu i jego podzespołów, sposoby ich tworzenia i konfiguracji, znane zagadnienia związane z ich działaniem oraz uwagi klientów nt. ich funkcjonowania. Specjalistyczne narzędzia wspierające zarządzanie konfiguracją pozwalają na przechowanie oraz późniejsze wydobycie tej wiedzy.

W obrębie danego projektu powinien być zdefiniowany dokładny plan opisujący standardy i procedury zarządzania konfiguracją. Plan taki może obejmować następujące zagadnienia:

- produkty procesu wytwarzanego podlegające zarządzaniu,
- określenie osób odpowiedzialnych za zarządzanie konfiguracją i komunikację z zespołem projektowym,
- schemat wersjonowania, pozwalający na jednoznaczne umiejscowienie produktu procesu wytwarzanego w czasie, a także jego identyfikację w zależności od innych parametrów,
- schemat opisujący atrybuty produktów, takich jak miejsce danego elementu w procesie wytwarzanym czy status (np. „planowany”, „implementowany”, „stabilny”),
- opis narzędzi wspierających zarządzanie konfiguracją.

Dobre prowadzone zarządzanie konfiguracjami jest szczególnie istotne dla projektów długotrwałych, skierowanych do szerokiej bazy użytkowników działających na wielu platformach sprzętowych i systemowych, aplikacji zakładających wysoką konfigurowalność czy szczególne interakcje z zewnętrznym oprogramowaniem. Systemy tego typu generują ogromną ilość wersji, wydań i poprawek, co powoduje, że opanowanie relacji w jakie wchodzą (wzajemnie i z innymi komponentami) jest kluczowe dla sukcesu projektu.

Zadania

Zadanie 1

Dla poniższej klasy napisz odpowiadający jej kod w wybranym języku programowania. Zastosuj konwencje nazewnicze właściwe dla wybranego przez siebie języka. Opisz zastosowane konwencje nazewnicze.

CRejestracjaStudenta
- status_rejestracyjny: char
- data_rejestracji: Date
~ utwórz(status: char): short
~ przechowaj_ostatnią(s: char, d: Date): void
~ zmień_ostatnią(s: char): short

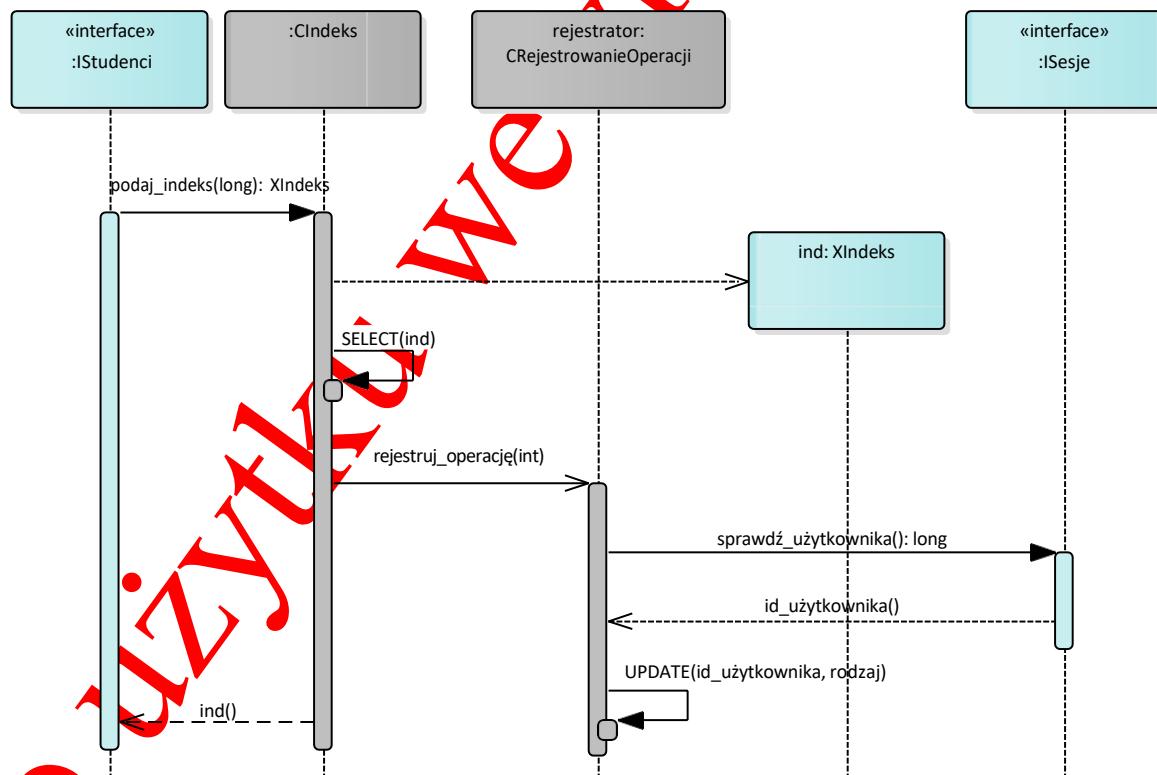
Zadanie 2

Stwórz model klas odpowiadający poniższemu szkieletowi kodu.

```
public class CTablaStudentow {  
  
    private int liczbaStudentowNaLiscie;  
    private CTablaSymboli translacja;  
  
    public void pokazStudenta(int nr) {}  
    void zbudujTabeleTranslacji(XListaStudentow lis) {}  
    private void ustalLiczbeStudentow(XListaStudentow lis) {}  
    public void kasuj() {}  
  
}  
  
public class CTablaSymboli {  
  
    public void dodaj(int numer, long numer) {}  
    public long podajId(int numer) {}  
  
}
```

Zadanie 3

Dla poniższego diagramu sekwencji napisz w wybranym języku programowania kod klas „CIndex” oraz „CRejestrowanieOperacji”.



Zadanie 4

Stwórz diagram sekwencji odpowiadający poniższemu fragmentowi kodu.

```
public class COknoListyStudiujacych {  
    IPrzegladanieListyStudentow przegladanieListyStudentow;  
    void przyciskPokazStudenta(int idStudenta) {  
        przegladanieListyStudentow.pokazStudenta(idStudenta);  
    }  
}
```

Zadanie 5

Przedstaw ciąg poleceń (etapów) dla systemu kontroli wersji w celu wprowadzenia nowego kodu do głównej gałęzi repozytorium.

Zadanie 6

Mamy zadany kod z głównej gałęzi oraz nowy kod. Podaj treść kodu połączeniu.

```
public class CStudent {  
    private long id;  
    public XStudent podajStudenta(long idStudenta) {  
        XStudent srud = new XStudent(idStudenta);  
        SELECT(stud);  
        return stud;  
    }  
}
```

```
public class CStudent {  
    private long id;  
    private CRejestracjaStudenta rejestracjaStudenta;  
    public short zmienOstatniaRejestracje(char status) {  
        return rejestracjaStudenta.zmienOstatnia(status);  
    }  
}
```

Słownik pojęć
Dużytk
Inżynieria odwrotna

Odtworzeniu modelu projektowego (najczęściej – modelu klas) na podstawie kodu.

Inżynieria okrężna

Cykł polegający na sukcesywnym stosowaniu inżynierii w przód i inżynierii odwrotnej.

Inżynieria w przód

Przeniesienie projektu systemu opisanego modelami projektowymi do kodu.

Co trzeba zapamiętać

Kodowanie na podstawie projektu

Podstawowym zadaniem programisty jest napisanie kodu, który będzie zgodny z projektem stworzonym przez architekta i projektanta kodowanych komponentów. Przeniesienie projektu systemu opisanego modelami projektowymi do kodu nazywane jest inżynierią w przód. Część kodu może być oparta bezpośrednio np. na modelach klasa oraz sekwencji. Dobrą praktyką implementacji jest utrzymywanie jej zgodności z modelami projektowymi. W ten sposób, programiści mają stale aktualną „mapę kodu”. Aby zachować zgodność aktualizowanego kodu z projektem, można zastosować inżynierię odwrotną.

Dobre praktyki kodowania

Dobrej jakości kod tworzony jest na podstawie dobrej jakości projektu. Oznacza to, że dobre praktyki kodowania często pokrywają się z dobrymi praktykami projektowania. Dobre praktyki kodowania są niezależnie od języka programowania, rozwiązań technicznych, charakteru organizacji informatycznej czy dziedziny projektu, w obrębie którego kod powstawał. Można je podzielić na praktyki redaktorskie, merytoryczne oraz związane z nawykami codziennej pracy. Praktyki redaktorskie dotyczą formatowania tekstu, komentowania kodu oraz konwencji nazewnictw. Praktyki merytoryczne dotyczą m.in. zasad projektowania podczas kodowania, zapewnienia parametryzowalności kodu, posługiwania się wzorcami projektowymi, przestrzegania walidacji parametrów wejściowych, dbałości o dane i zasoby maszyny, brania pod uwagę przenośności i skalowalności. Praktyki związane z nawykami dotyczą m.in. zapewnienia „czystości” własnego kodu, pracy na aktualnym kodzie, umiejętności korzystania ze środowiska programistycznego.

Zarządzanie wersjami, konfiguracją i zmianami

Zarządzanie wersjami oznacza zapisywanie kolejnych zmian w pliku lub zestawie plików, czyli tworzenie kolejnych wersji. Zarządzanie zmianami w projektach konstrukcji oprogramowania najczęściej dotyczy plików z kodem źródłowym, lecz może ono być stosowane do dowolnego rodzaju plików. Do zarządzania wersjami stosujemy specjalne narzędzia, które nazywamy systemami zarządzania wersjami. Istnieją dwa główne modele zarządzania wersjami: scentralizowane (klient-serwer) oraz rozproszone. Zarządzanie zmianami polega na stosowaniu odpowiedniego procesu obsługi zmian, które pojawiają się w różnych fazach projektu. Zmiany mogą być wynikiem okoliczności związanych ze sposobem rozwoju oprogramowania, a także czynników zewnętrznych: zmian prawnych, organizacyjnych, cyklu koniunkturalnego, potrzeb użytkowników, rozwoju infrastruktury, wyników analizy działania systemu w obrębie wspieranego biznesu itd. Zarządzanie konfiguracją polega na tworzeniu kolejnych wydań systemu składających się z szeregu produktów procesu wytwarzego, takich jak kod źródłowy, kod wykonywalny, modele projektowe, dokumentacja techniczna, itd.

Rozwiązania zadań

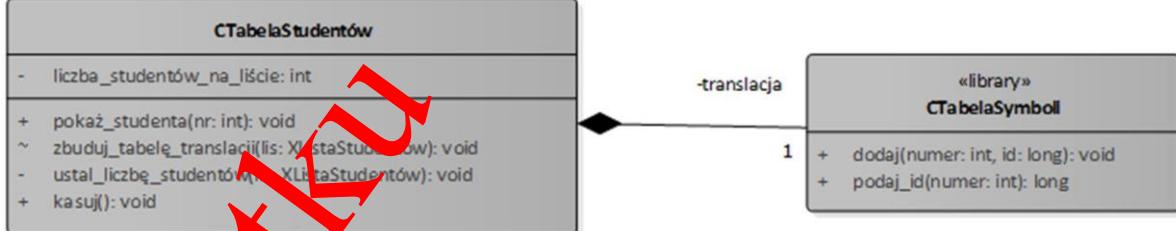
Rozwiązanie zadania 1

```
public class CRejestracjaStudenta {  
  
    private char statusRejestracyjny;  
    private DateTime dataRejestracji;  
  
    short utworz(char status) {  
        statusRejestracyjny = status;  
        dataRejestracyji = DateTime.Now;  
    }  
  
    void przechowajOstatnia(char s, DateTime d) {  
        statusRejestracyjny = s;  
        dataRejestracyji = d;  
    }  
  
    short zmienOstatnia(char s) {  
        statusRejestracyjny = s;  
    }  
  
}
```

Zastosowane konwencje nazewnicze:

- nazwa klasy w notacji „Pascal case” (wielkie litery na początku wszystkich wyrazów)
- pola prywatne w notacji „camel case” (mała pierwsza litera, wielkie litery na początku kolejnych wyrazów)
- metody w notacji jak wyżej

Rozwiązanie zadania 2



Do użycia

Rozwiązań zadania 3

```
public class CIndeks {  
  
    «interface» :IPrzeglądarkaListStudentów  
    :COknoListyStudiujących  
    «okno» :Lista studiujących  
  
    ...  
  
    void pokaż_studenta(int id) {  
        long idUzytkownika = sesje.sprawdzUzytkownika();  
        UPDATE(idUzytkownika, id);  
    }  
}
```

Rozwiązań zadania 4

Uwaga: obiekt „Lista studiujących” oznacza przykładowy element systemu operacyjnego, który wywołuje zdarzenie „przycisk_pokaz_studenta”.

Rozwiązań zadania 5

Dla systemu Git, ciąg poleceń/etapów jest następujący: (Stage Files ->) Commit -> Push -> Merge Request -> Merge

Rozwiązanie zadania 6

```
public class CStudent {  
  
    private long id;  
    private CRejestracjaStudenta rejestracjaStudenta;  
  
    public XStudent podajStudenta(long idStudenta) {  
        XStudent srud = new XStudent(idStudenta);  
        SELECT(stud);  
        return stud;  
    }  
  
    public short zmienOstatniaRejestracje(char status) {  
        return rejestracjaStudenta.zmienOstatnia(status);  
    }  
}
```

Do użytku wewnętrzny

12. Podstawy testowania

12.1. Rola testowania w inżynierii oprogramowania

Testowanie oprogramowania jest dyscypliną inżynierii oprogramowania zajmującą się określaniem metod oraz przeprowadzaniem badania poprawności działania oprogramowania. Przy tym, przez poprawność rozumiemy zgodność oprogramowania z wymaganiami. Na przestrzeni lat rozwinięto wiele metod, które pozwalają zespołom testującym nadążyć za coraz szybciej powstającymi programami, a także opanować coraz bardziej złożone interakcje między systemami informatycznymi. Celem tych wyśników jest podniesienie jakości wytwarzanego oprogramowania – zarówno postrzeganej z perspektywy doskonałości technicznej kodu, jak i zgodności funkcjonalności aplikacji z potrzebami ich użytkowników.

Podczas powstawania systemu informatycznego konieczne jest stwierdzenie, czy jego działanie nie wykazuje odstępstw od pewnych narzuconych warunków. Warunki te powstają poprzez analizę potrzeb klienta oraz określenie możliwości zespołu tworzącego program. Odstępstwa od tak ustalonej specyfikacji nazywa się **błędami oprogramowania** i obejmują one takie sytuacje jak:

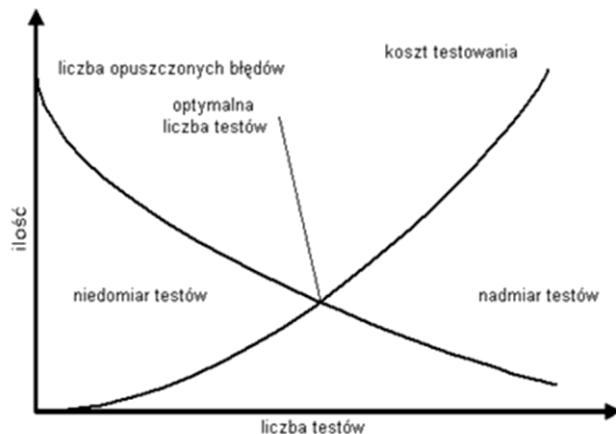
- oprogramowanie nie wykonuje czegoś, co powinno wykonywać,
- oprogramowanie robi coś, czego nie powinno robić,
- oprogramowanie nie wypełnia warunków dotyczących cech pozafunkcjonalnych (szybkość działania, łatwość użycia itd.),
- oprogramowanie zachowuje się w sposób nie przewidziany specyfikacją.

Źródła błędów są różnorakie. Część błędów powstaje w wyniku złego zaprojektowania systemu informatycznego. Co gorsza, usterki tego typu często ujawniają się dopiero w późnych fazach cyklu życia oprogramowania. Podobnie jest z błędami wynikającymi z wadliwej specyfikacji wymagań. To źródło defektów oprogramowania jest niebezpieczne z jeszcze jednego powodu. Projekt realizowany według niewłaściwych założeń, mimo że w odniesieniu do nich jest poprawny, w rzeczywistości jest nie do przyjęcia dla zamawiającego.

Istotną część błędów stanowią tzw. **pluskwy** (ang. bug) – są to usterki umiejscowione w kodzie programu i wynikające ze zmęczenia, przeoczeń bądź nieuwagi programistów. Pozostałe źródła błędów to inne nieprawidłowości w kodzie programu oraz defekty związane ze wadliwym sprzętem, specyfiką środowiska programistycznego, niekompatybilnością urządzeń itp. Celem procesu testowania oprogramowania jest możliwie wcześnie znajdowanie nieprawidłowości działania aplikacji, a także odpowiednie opisywanie ich w celu ułatwienia programistom dokonania poprawek. W procesie testowania tworzy się dane testowe wykorzystywane podczas badania reakcji systemu na sygnały wejściowe. Zbiór danych testowych powinien być odpowiednio duży (przynajmniej rzędu wielkości zbioru danych przetwarzanych docelowo przez system) oraz zróżnicowany (zawierać odpowiednią ilość typowych informacji, ale nie pomijać nawet „egzotycznych” wyjątków).

W toku projektu istotne jest przygotowanie odpowiedniej części budżetu w celu właściwego przetestowania tworzonego systemu. Na testy można wydać ogromne pieniądze, ale należy pamiętać, że nigdy nie ma pewności, że tworzone oprogramowanie jest bezbłędne. Z tego powodu potrzebne jest uzyskanie odpowiedniej równowagi między środkami poświęconymi na testowanie, a uzyskanym efektem (jakością oprogramowania). Zilustrowane to zostało na rysunku 12.1. Przekroczenie pewnej

optymalnej liczby (zakresu) testów powoduje na tyle duży wzrost kosztów, że staje się to nieopłacalne biorąc pod uwagę możliwości znalezienia dodatkowych błędów.



Rysunek 12.1: Ustalenie odpowiedniej liczby zadań testowych

Nie istnieje jednoznaczna miara, pozwalająca ocenić skuteczność przyjętych strategii testowania. Jednym z podejść do problemu weryfikacji procesu testowania jest badanie pokrycia testami zbioru przetwarzanych przez system danych oraz kodu wykonywanego podczas uruchamiania testów. Miara pokrycia kodu i danych pozwala ocenić, które fragmenty kodu (linie, instrukcje) zostały „zbadane” oraz jaki zakres danych został wykorzystany przy weryfikacji systemu.

Inną miarą jest szacunkowe określenie liczby błędów w systemie i liczby błędów wykrytych. Można w tym celu stosować metodę posiewową. Technika ta polega na tym, że w programie ukrywa się celowo pewną ilość sztucznie wytworzonych błędów. Muszą być one podobne do tych prawdziwych. Następnie zespół testujący, nieświadomy umieszczonej błędów, jest oceniany poprzez odpowiednie oszacowanie bazujące na liczbie wykrytych przez niego usterek.

Jeżeli:

B – liczba wykrytych błędów

b – liczba posianych błędów, które zostały wykryte

P – liczba posianych błędów

to szacunkowa liczba błędów przed wykonaniem testów wynosi

$$(B - b) * \frac{P}{b}$$

a szacunkowa liczba błędów pozostałych po wykonaniu testów wynosi

$$(B - b) * \frac{P}{b} - 1$$

Skuteczność przeprowadzonych testów jest kluczowa dla zapewnienia dobrej jakości tworzonego systemu oprogramowania. Dobrze zaplanowane i przeprowadzone testy pozwalają zespołowi deweloperskiemu na szybkie poprawienie wykrytych błędów oraz dokonanie modyfikacji usprawniających cały proces wytwarzczy.

12.2. Podstawowe metody testowania

W czasie wczesnego rozwoju inżynierii oprogramowania rozwijano teoretyczne metody testowania (na przykład formalne dowodzenie poprawności algorytmów Dijkstry). Nie sprawdzają się one jednak w warunkach zasobów pieniężnych i czasowych jakimi dysponują współczesne zespoły wytwarzające oprogramowanie. Popularne obecnie techniki testowania można podzielić na metody nieinwazyjne (metody czarnej skrzynki), metody związane z analizą powstającego kodu (metoda szklanej skrzynki) oraz metody analizy specyfikacji (testy statyczne). Wymienione podejścia do testowania, niezależnie od ich rodzaju, można przeprowadzać na wielu poziomach abstrakcji. Poniżej przedstawiono krótki opis podstawowych metod testowania.

Najczęściej używanymi metodami testowania są metody kierujące się zasadą **testów czarnej skrzynki** (ang. black box testing). Polegają one na sprawdzaniu działania programu (systemu informatycznego) bazującym całkowicie na monitorowaniu danych wejściowych i wynikowych oraz porównywaniu ich z oczekiwaniemi (założeniami). Podczas takich testów nie wnika się w strukturę kodu programu oraz działa się nieinwazyjnie. Testowanie tego typu obejmuje szereg różnych rodzajów testów związanych z funkcjonalnością oprogramowania. Jest metodą testowania dynamicznego. Najczęściej stosuje się ją podczas testów funkcjonalnych oprogramowania. Zdarza się jednak, że bada się w ten sposób specyfikację programu, już podczas fazy projektowej – staje się wtedy metodą statyczną.

Zaletą metody czarnej skrzynki jest możliwość przeprowadzania testów tego typu na każdym etapie rozwoju systemu informatycznego. Jak już wspomniano, testować można w ten sposób specyfikację (od najwcześniejszych faz jej powstawania), ale także badać program podczas implementacji, sprawdzając określone funkcje niezależnie od powstającego równolegle kodu lub (gdy prace nad produktem zbliżają się ku końcowi) przeprowadzając testy typu „build and smoke” – codzienne komplikacje i uruchomienia całości systemu („build”) połączone z poszukiwaniem tej części funkcjonalności, która nie jest jeszcze kompletna, zgodna z założeniami lub wręcz wadliwa („smoke”). Może być to wreszcie testowanie gotowego programu przez osoby z tworzącego go zespołu programistycznego lub też ludzi z zewnątrz, takich jak wynajęci testerzy, beta-testerzy, a także przez programistów, którzy dołączyli do zespołu w późnym etapie tworzenia produktu.

Tutaj pojawia się druga ważna zaleta omawianej metody: gwarantuje ona „świeżość spojrzenia” i dużą obiektywność. Działanie programu jest badane bardziej od strony oczekiwania użytkownika, niż z perspektywy zawiłości technicznych pewnych rozwiązań. Testerzy nie koncentrują się na tym jak program działa, ale czy działa. Nie ma też mowy o braku obiektywności – nie jest istotne kto, gdzie i kiedy napisał daną część kodu – ważne czy spełniona jest jedna z założonych funkcjonalności. Testowanie metodą czarnej skrzynki pozwala wykryć błędy, o których „nie śniło się” programistom – bada zachowania nie ujęte w założeniach projektu.

Wadą metody jest niemożność dopasowania parametrów testów do postawionego problemu – np. można jedynie domyślać się, które funkcjonalności wymagają bardziej złożonych (i mogących przez to zawierać więcej błędów) algorytmów, albo jakie części programu obciążają najbardziej sprzęt.

By przeprowadzić udany test metodą „czarnej skrzynki” można wykorzystać pewne techniki weryfikacji oprogramowania bazujące na zasadach nieingerencji i nieznajomości kodu programu. Należą do nich metody:

- tworzenia klas równoważności,
- warunków granicznych,
- zmian stanów,

- niedoświadczonego użytkownika.

Stosowanie metody **klas równoważności** wynika z podstawowego faktu śledzenia wszystkich możliwych stanów w systemach komputerowych. W dowolnym, nawet najprostszym procesie informatycznym, liczba możliwych stanów, danych wejściowych, konfiguracji sprzętu, zachowań użytkownika jest ogromna, a z punktu widzenia ograniczonego czasem i funduszami zespołu programistycznego wręcz nieskończona. Z drugiej strony podczas badania jakości oprogramowania niezbędne jest przetestowanie jak najszerzego spektrum możliwych sytuacji z jakimi będzie miał do czynienia sprawdzany program. Rozwiązaniem powyższego konfliktu jest metoda klas równoważności, która pomaga w wyborze odpowiednich zadań testowych ograniczając liczbę zadań testowych, nie zmniejszając jednocześnie skuteczności testów.

Klasą równoważności nazywa się zbiór zadań testowych, które testują to samo lub ujawniają te same błędy. Głównym zagadnieniem podczas testowania metodą klas równoważności jest identyfikacja klas równoważności, czyli taki podział zagadnień testowych, który pozwala uzyskać odpowiednie pokrycie testowanego oprogramowania i jednocześnie daje zastosować się w praktyce. Przy zbyt dużej redukcji liczby klas równoważności istnieje ryzyko eliminacji testów, które ujawniają część błędów. Natomiast przy wysokiej liczbie klas część testów jest wielokrotnie i bezproduktywnie powtarzana. Identyfikację klas równoważności można przeprowadzać ze względu na podobne dane wejściowe, podobne stany programu, podobne dane oczekiwane, podobne zachowania użytkownika itp. Nie istnieje jeden jednoznaczny podział na klasy równoważności ani ścisła metoda jego wyznaczania – panuje duża umowność, a jedynym obowiązującym kryterium jest dostateczne pokrycie testowe.

Uwagę testera powinno zwrócić stworzenie klas równoważności dla wartości domyślnych, zerowych, początkowych i innych przypadków, gdy przyszły użytkownik nie wprowadzi do programu oczekiwanych danych. Inną klasą równoważności mogą stanowić wartości założenia błędne, nieprawidłowe, np. znaki w miejsce liczb lub znaki sterujące zamiast ciągów liter.

Podczas podziału na klasy równoważności często ujawnia się problem warunków granicznych, zasadzający się na dilemacie jednoznacznego zaliczenia danego zadania testowego do jednej z kilku grup, które „zbiegają się” lub „stykają” w danym punkcie. Wtedy metoda klas równoważności zaczyna uzupełniać się z metodą warunków granicznych.

Testowanie **warunków granicznych** można również nazwać „praktycznym zastosowaniem klas równoważności dla testowania danych”. Ideą tej metody jest określenie przedziałów jakie mogą przyjmować dane wejściowe oraz sprawdzenie zachowania się programu dla danych z okolicy granic tych przedziałów. Wynika to z tego, że w naukach informatycznych (a więc także w programowaniu) konieczne jest ścisłe deklarowanie zakresów liczb, wielkości tablic, rodzajów zmiennych; są to tzw. definicje warunków brzegowych, których komputer wymaga bezwzględnego spełnienia, inaczej odrzuca pewne wartości jako błędne. Błędy graniczne występują często jako proste pomyłki programistów wynikające z nieuwagi lub zmęczenia.

W ogólności, jeśli x – wartość wprowadzana, $\langle a, b \rangle$ - przewidziany w specyfikacji projektu zakres dla tej danej (może być również typu nierównościowego), to test należy przeprowadzić dla:

$$\begin{aligned}
 x &= a - 1 \\
 x &= a \\
 x &= a + 1 \\
 x &= b - 1 \\
 x &= b \\
 x &= b - 1
 \end{aligned}$$

Jeśli program odpowiednio reaguje na takie dane, to najprawdopodobniej (o ile przedział został wyznaczony poprawnie) będzie działał prawidłowo również dla danych ze środka przedziału. Własność ta nie jest przechodnia i spełnienie założeń projektu dla danych z wnętrza przedziału nie implikuje odpowiedniego traktowania przez program tych z jego pogranicza. Wyżej wymienione wartości x można uzupełnić o inne np.: $x = (a+b)/2, x = a-2$, jednak nie jest to niezbędne. Warto zwrócić uwagę, że określenie „program reaguje poprawnie” może odnosić się zarówno do akceptacji i przetworzenia danych z podanego przedziału, ale także do sygnalizacji wprowadzenia błędnych wartości.

Powyższe przedstawienie sposobów identyfikowania wartości granicznych jest czysto teoretyczne i ma znaczenie tylko poglądowe – obrazuje problem na zbiorze liczb całkowitych. W programach występuje wiele innych typów danych wymagających odrębnego potraktowania. Spośród nich można rozpatrzyć typy takie jak numeryczny, znakowy, opisujący pozycję, ilość, szybkość, położenie, wielkość itp. Dla takich typów należy uwzględnić atrybuty rodzaju pierwszy/ostatni, minimum/maksimum, największy/najmniejszy, początek/koniec, pusty/pełny, najmłodszy/najstarszy, najwolniejszy/najszybszy, następny/najdalszy, najwyższy/najniższy, ponad/poniżej itd. Oczywiście dla danych zastosowań i zdefiniowanych przez użytkownika typów danych powyższe warunki mogą przyjąć zupełnie inną postać. Istotne jest, aby wyodrębnić możliwie dużo warunków brzegowych – znacznie podnosi to jakość testowania.

Istnieje także problem testowania wewnętrznych warunków granicznych – ograniczeń wynikających np. z budowy sprzętu (procesora) lub działania kompilatora. Na różnych platformach sprzętowych zakresy liczbowe dla typów zmiennych określane są najczęściej jako potęgi dwójki, jednak różnią się od siebie. Ważne jest, aby sprawdzić twórców programu czy dostosowali się do specyfiki sprzętu, np. zbadać zachowanie się zmiennych, które (jak przypuszczamy) przechowywane są jako liczby typu całkowitego w dodatkowych warunkach granicznych typu (0, 1, 7, 15, 127, 255, 511, 1023, ...) albo zmiennych znakowych przez wprowadzanie znaków typu „powrót karetki” albo „koniec pliku”.

Testowanie warunków granicznych to tylko sprawdzenie programu dla konkretnych danych. Działaniem, które może ujawnić wiele błędów jest test zmian stanów. Testowanie **zmian stanów** programu skupia się na sprawdzeniu poprawności przejść między trybami działania programu i nie przywiązuje dużej wagi do wprowadzanych danych (o ile nie są one ściśle połączone ze zmianami stanów).

Mimo że liczba możliwych stanów programu jest ograniczona, to liczba przejść między nimi może być ogromna. Dodatkowo, tester obciążony jest zadaniem sprawdzenia wszystkich stanów oraz wszystkich między nimi połączeń. Złożoność tego testu, tak jak dla testów danych, najczęściej zdecydowanie przekracza możliwości zespołu testującego. Przypomina to problem komiwojażera, gdzie dla tylko pięciu miast liczba możliwych tras między nimi wynosi 120. Jeżeli n – liczba stanów, to liczba możliwych przejść między stanami wynosi $n!$. Aby poradzić sobie z tak gwałtownie wzrastającą złożonością zadania, stosuje się podział stanów według klas równoważności, przy czym używa się oczywiście odmiennych kryteriów podziału niż dla danych.

Pierwszym krokiem podczas testowania zmian stanów jest wykorzystanie mapy stanów – powinna być ona stworzona razem ze specyfikacją projektu. Mapa stanów może być przedstawiona na wiele sposobów, np. jako diagram przepływów, diagram maszyny stanów albo tabela stanów. Taki opis przejść między stanami powinien zawierać wszystkie możliwe stanu programu, sygnały wymuszające przejścia między stanami, sygnały wyjściowe, dane, komunikaty lub inne zmiany określonych warunków przy przejściu do rozpatrywanych stanów.

Redukcja problemu dużej ilości testów może być oparta na kilku prostych regułach. Przede wszystkim każdy stan powinien być odwiedzony przynajmniej raz. Tester powinien zwracać uwagę na najczęściej wykorzystywane przejścia między stanami i im szczegółowo się przyjrzeć. Ważne są badania stanów awaryjnych i powrotów z nich. W trakcie użytkowania sytuacje awaryjne występują raczej rzadko, ale kod obsługujący takie zdarzenia musi być bezbłędny, aby nie spowodować poważniejszej awarii lub utraty danych.

Warto do testowania stanów wykorzystać testy automatyczne. Można wtedy wywołać wiele ciekawych efektów i ujawnić nowe błędy – wprowadzając losowość zmieniać stany rzadko uczęszczanymi ścieżkami, przechodzić między trybami pracy powtarzając wielokrotnie te same operacje, czy wreszcie obciążyć program w stopniu maksymalnym dopuszczanym przez specyfikację, a następnie przekroczyć to obciążenie. Badając programy działające w takich mniej standardowych sytuacjach (które przecież mogą zaistnieć podczas rzeczywistego użytkowania) można uzyskać nową wiedzę na temat pozostałych przez programistów niedociągnięć. Metoda testowania zmian stanów szczególnie dobrze wpisuje się w ideę testowania metodą czarnej skrzynki, ponieważ na przejścia między trybami działania programu najlepiej jest spojrzeć z punktu widzenia użytkownika, bez wnikania w wartości zmiennych czy używane funkcje.

Inną metodą testowania, pozwalającą spojrzeć na błędy oprogramowania z perspektywy jego odbiorcy jest metoda **niedoświadczonego użytkownika**. Jest to najczęściej ostatnia faza testowania, która przebiega przy udziale użytkowników końcowych systemu. Są to najczęściej osoby, które z systemem mają do czynienia po raz pierwszy, czyli nie nabrały określonej rutyny wykonywania czynności w sposób powtarzalny. W idealnej sytuacji powinny to być osoby niepowiązane z klientem i wydawcą oprogramowania oraz obsługujące system bez specjalnych szkoleń i specjalistycznej wiedzy. Jest to ciekawa metoda testowania, pozwalająca odkryć błędy pomijane przez osoby poruszające się w systemie w standardowy sposób. Testerzy dostają do wykonania określone zadania, odpowiadające zestawom przypadków użycia. Wykonują je zgodnie ze swoją intuicją i dostępnymi w systemie instrukcjami. Mogą popełniać różnego rodzaju błędy – np. wprowadzać dane w niewłaściwy sposób lub wybierać opcje w niewłaściwej kolejności. Błędne działanie systemu może np. polegać na niewłaściwej sygnalizacji błędnego zachowania użytkownika, czy też braku informacji o tym, które dane są niewłaściwie wprowadzone. Zebrane w ten sposób informacje od użytkowników są cenną wskazówką dla usprawnienia funkcjonowania systemu. Oczywiście, niedoświadczony programista może wykryć klasy błędów wykrywane również innymi metodami.

Przeciwagą dla metod czarnej skrzynki są metody „szklanej (białej) skrzynki” (ang. glass-, white-box testing). Wykorzystują one wiedzę o kodzie programu w celu podniesienia skuteczności testów. Testy bazujące na wglądzie w strukturę programu polegają przede wszystkim na badaniu kodu – czy to podczas formalnych i nieformalnych przeglądów czy poprzez analizę raportów narzędzi-analizatorów kodu. Tego typu testy oceniają najczęściej poprawność kodu ze względu na praktyki i reguły jego tworzenia, ustalone dla danej organizacji czy projektu. Metodą pośrednią między testem „czarną” a „białą skrzynką” jest test „szarej skrzynki”. Polega on na ograniczonym wykorzystywaniu wiedzy o kodzie programu, często napisanego w języku skryptowym (np. strony WWW). Szczegółowe omówienie metod analizy kody leży poza zakresem niniejszych materiałów.

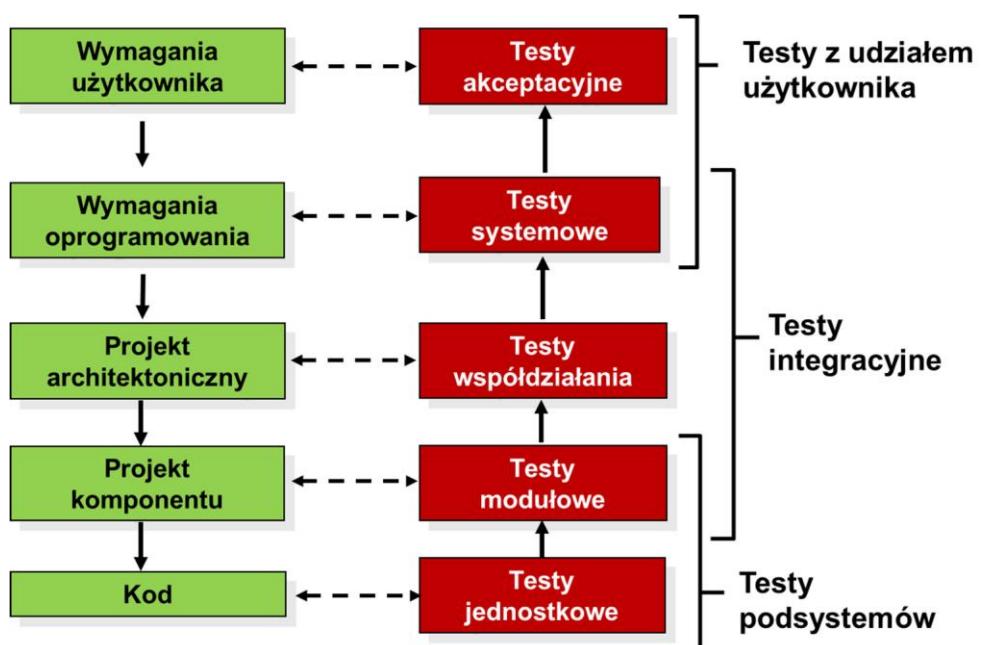
Zagadnieniem obejmującym część metod wymienionych powyżej jest testowanie integracyjne. Proces ten obejmuje sprawdzanie jakości elementów systemu scalanych na różnych poziomach analizy systemu. Głównym zagadnieniem podczas przeprowadzania testów integracji jest określenie dokładnej przyczyny ew. błędów. Innym problemem jest często nierównomierny rozwój testowanych komponentów – wymagane jest wtedy stosowanie atrap imitujących funkcjonalność elementów niekompletnych na danym etapie testowania. Z tego powodu testy integracyjne są najczęściej kombinacją testów z następujących (weryfikacja rozpoczyna się od największych części systemu) i wstępujących (w pierwszej kolejności badane pod kątem błędów są małe moduły).

Jak już wspomniano istotnym elementem testów integracyjnych jest sprawdzanie interfejsów, przez które komunikują się komponenty. Nawet jeśli interfejsy same w sobie działają poprawnie, to możliwe są sytuacje, gdy zostają nieprawidłowo użyte przez np. niewłaściwe przekazanie parametrów lub błędna interpretację opisującego je kontraktu.

Powyżej wymienione metody dają szeroki wybór metod, które można wykorzystać planując proces testowania, jednak ostatnią fazą testowania jest zawsze odbiór systemu przez klienta. Testowanie w tej fazie opiera się na tworzeniu scenariuszy testowych i zestawów przypadków testowych na podstawie przypadków użycia zawartych w wymaganiach systemowych.

12.3. Poziomy testowania

Testy możemy skategoryzować w zależności od ich rodzaju oraz etapu procesu wytwarzania oprogramowania, na którym należy zacząć je stosować. Tego typu klasyfikacja została przedstawiona na rysunku 12.2. Rysunek przedstawia kilka poziomów testowania wraz z odpowiadającymi im produktami procesu wytwarzania oprogramowania, które są weryfikowane na danym poziomie. Warto zwrócić uwagę, że różne rodzaje testów mogą być i typowo są wykonywane przez osoby pełniące różne role w projekcie. Na przykład, testy jednostkowe wykonywane są przez programistów, a testy akceptacyjne – przez dedykowanych testerów we współpracy z użytkownikami.

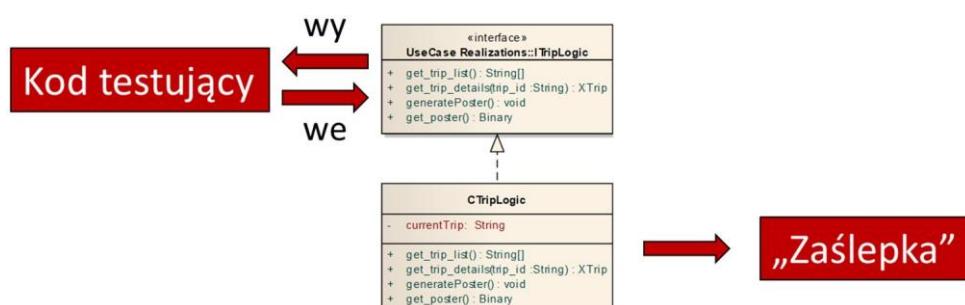


Rysunek 12.2: Różne poziomy testowania systemu

Najbardziej szczegółowym rodzajem testów są **testy jednostkowe**. Dotyczą one kodu systemu i są tworzone bezpośrednio przez programistów go piszących. Rolą testów jednostkowych jest sprawdzenie działania elementarnych składników kodu, np. klas oraz ich metod. Sprawdzana jest logika działania procedury lub funkcji poprzez zadanie danych wejściowych (parametrów) i sprawdzenie, czy dane wyjściowe (wyniki) są zgodne z oczekiwaniemi. Dobrze przygotowane testy jednostkowe pozwalają również na przeprowadzanie **testów regresyjnych**. Są to testy powtarzane w identyczny sposób w kolejnych iteracjach projektu, mające na celu sprawdzenie, czy wprowadzone w kodzie zmiany (np. optymalizacja, refaktoryzacja) nie spowodowały powstania nowych błędów.

Tworzenie testów jednostkowych posiada silne wsparcie narzędziowe. Istnieje wiele dedykowanych bibliotek ułatwiających pisanie i przeprowadzanie takich testów. Same testy najczęściej uruchamiane są z pomocą tzw. sterowników testów, które potrafią zlecić wykonanie całych ich zbiorów, informując o uzyskanym stopniu powodzenia. Automatyzacja testów jest bardzo istotnym aspektem dyscypliny testowania i jest omówiona w następnym rozdziale.

Testy jednostkowe dążą do sprawdzania działania rozpatrywanych elementów niezależnie od innych z nimi powiązanymi. Jeśli dany element wymaga do działania innych elementów, stosuje się tzw. zaślepki, które imitują działanie tych elementów. Do tworzenia zaślepek na potrzeby testów jednostkowych również istnieje wiele dedykowanych bibliotek, które ułatwiają ten proces. Schemat wykorzystania zaślepek w testach jednostkowych został pokazany na rysunku 12.3. W podanym przykładzie testowany jest komponent widoczny poprzez jego interfejs (np. API). Na ten interfejs wprowadzane są odpowiednie zestawy danych testowych („we”), przygotowane metodami podanymi w poprzedniej sekcji (np. metodą warunków granicznych). Wprowadzenie danych uruchamia odpowiednią procedurę, która jest wykonywana przez kod komponentu (tu: klasę „CTripLogic”). Ważne jest to, że kod komponentu może potrzebować dostępu do interfejsu innego komponentu. W takiej sytuacji, tworzona jest „zaślepka”, czyli kod podający standardowe dane testowe. Dane te są wykorzystywane przez testowany komponent do realizacji jego funkcjonalności. W rezultacie, komponent zwraca dane wynikowe („wy”). Dane te są poddawane analizie pod kątem zgodności z oczekiwaniemi. W całej procedurze, zaślepka ma wyeliminować wpływ ewentualnych błędów komponentu zewnętrznego, który nie podlega testowaniu tym zestawem testów.



Rysunek 12.3: Schemat testowania jednostkowego z wykorzystaniem zaślepek

Testy kodu w których zaczynamy testować większą liczbę współpracujących ze sobą elementów naraz (np. cały komponent) możemy nazwać **testami modułowymi**, które stanowią wyższy poziom testów jednostkowych. Wykonywane są one również przez programistów, najczęściej z wykorzystaniem tych samych narzędzi. Rozpoczynają jednak już one kolejną kategorię testów, nazywanych **testami integracyjnymi**, których rolą jest sprawdzanie poprawnej współpracy wielu różnych elementów. W skład tej kategorii wchodzą również **testy współdziałania i testy systemowe**. Pierwsze z nich sprawdzają współpracę par współpracujących ze sobą elementów. Nacisk kładziony jest na sprawdzenie poprawności komunikacji poprzez wcześniej ustalone interfejsy. Drugie skupiają się na testowaniu poprawnej

współpracy wszystkich elementów systemu koniecznych do dostarczenia poszczególnych funkcjonalności. Testy współdziałania przeprowadzane są przez osoby dokonujące integracji systemu z poszczególnych komponentów (integratorzy). Testy systemowe obejmują cały system i w wielu metodykach są wykonywane przez wydzielona rolę testera.

Ostatni rodzaj testów stanowią **testy akceptacyjne**, które mają dowieść zgodności systemu z wymaganiami, co pozwoli na jego ostateczny odbiór przez klienta. Typowo dokonywane są one w oparciu o scenariusze przypadków użycia uzupełnione o odpowiednie dane testowe. W testach akceptacyjnych biorą często udział przedstawiciele klienta lub przyszli użytkownicy systemu. Wykonują oni scenariusze testowe, które są omówione w kolejnej sekcji.

12.4. Testowanie przypadków użycia systemu

Najczęściej używaną jednostką wymagań funkcjonalnych, omawianą w poprzednich modułach jest przypadek użycia lub historia użytkownika. Testowanie takich jednostek jest wykonywane na poziomie testów systemowych i testów akceptacyjnych. Testy polegają na projektowaniu, a następnie wykonywaniu tzw. **scenariuszy testowych**. Scenariusz testowy opisuje kolejne kroki interakcji użytkownika z systemem, prowadzące do uzyskania określonego efektu. Istotne jest to, że poprawność działania systemu jest stwierdzana jedynie poprzez obserwowanie działania systemu uruchamiane za pośrednictwem jego interfejsu zewnętrznego (głównie – interfejsu użytkownika). Często, aby sprawdzić działanie określonego przypadku użycia konieczne jest zaprojektowanie scenariuszy testowych uwzględniających uruchomienie innych przypadków użycia systemu.

Jak wspomniano w module III, funkcjonalność przypadków użycia opisuje się przez określenie ich scenariuszy. Dla każdego przypadku użycia definiujemy scenariusz podstawowy (zakończony sukcesem) oraz scenariusze alternatywne (warianty scenariusza głównego osiągające cel inną drogą lub kończące się niepowodzeniem). Wykorzystanie tych scenariuszy w testach akceptacyjnych polega na przekształceniu ich na scenariusze testowe. Kolejne zdania scenariuszy przypadków użycia stanowią kolejne kroki scenariuszy testowych. Kroki scenariuszy opisujące zachowania aktorów odpowiadają danym oraz decyzjom użytkownika wprowadzanym w odpowiednich momentach przeprowadzania testów. Wynikami procesu testowania są reakcje systemu zachodzące pod wpływem takich sygnałów, np. wyświetlane na ekranie określone dane.

Jak wspomniano powyżej, scenariusze testowe wykonywane są metodą czarnej skrzynki. Oznacza to, że często konieczne jest zestawienie ze sobą scenariuszy dla kilku przypadków użycia. Odpowiedni przykład takiego złożenia widzimy na rysunku 12.4. Pokazane jest tam kilka scenariuszy testowych sprawdzających działanie przypadku użycia „Dodaj model samochodu” (patrz moduł III). Zwróćmy uwagę na to, że scenariusze te zbudowane są na bazie scenariuszy dwóch przypadków użycia – przypadku badanego, jak również przypadku użycia „Wyświetl listę modeli samochodów”.

<p>ST213: Dodaj model samochodu – sukces (POZYTYWNY)</p> <p>A. Wyświetl listę modeli samochodów – główny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Lista modeli samochodów 2. System pobiera listę modeli samochodów 3. System wyświetla okno listy modeli samochodów <p>Wynik → Okno listy modeli samochodów – pusta lista</p> <ol style="list-style-type: none"> 4. Kierownik wybiera przycisk Zamknij <p>B. Dodaj model samochodu - główny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Dodaj 2. System wyświetla okno dodawania modelu samochodu 3. Kierownik wprowadza model samochodu <p>Dane wejściowe → plik „modele_poprawne.txt”</p> <ol style="list-style-type: none"> 4. System waliduje model samochodu [model samochodu poprawny] 5. System zapisuje model samochodu 6. System wyświetla komunikat o sukcesie operacji 7. Kierownik wybiera przycisk OK <p>C. Wyświetl listę modeli samochodów – główny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Lista modeli samochodów 2. System pobiera listę modeli samochodów 3. System wyświetla okno listy modeli samochodów <p>Wynik → Okno listy modeli samochodów – lista zawiera dodany model samochodu</p> <ol style="list-style-type: none"> 4. Kierownik wybiera przycisk Zamknij 	<p>ST214: Dodaj model samochodu – porażka (POZYTYWNY)</p> <p>A. Wykonanie ST213 dla całego pliku „modele_poprawne.txt.”</p> <p>B. Wyświetl listę modeli samochodów – główny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Lista modeli samochodów 2. System pobiera listę modeli samochodów 3. System wyświetla okno listy modeli samochodów <p>Wynik → Okno listy modeli samochodów – pusta lista</p> <ol style="list-style-type: none"> 4. Kierownik wybiera przycisk Zamknij <p>B. Dodaj model samochodu - alternatywny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Dodaj 2. System wyświetla okno dodawania modelu samochodu 3. Kierownik wprowadza model samochodu <p>Dane wejściowe → plik „modele_niepoprawne.txt”</p> <ol style="list-style-type: none"> 4. System waliduje model samochodu [model samochodu niepoprawny] 5. System wyświetla komunikat o błędnych danych 6. Kierownik wybiera przycisk OK <p>C. Wyświetl listę modeli samochodów – główny</p> <ol style="list-style-type: none"> 1. Kierownik wybiera przycisk Lista modeli samochodów 2. System pobiera listę modeli samochodów 3. System wyświetla okno listy modeli samochodów <p>Wynik → Okno listy modeli samochodów – lista NIE zawiera dodawanego model samochodu</p> <ol style="list-style-type: none"> 4. Kierownik wybiera przycisk Zamknij
<p>ST215: Dodaj model samochodu – sukces (NEGATYWNY)</p> <p>A. Wykonanie ST213-A</p> <p>B. Wykonanie ST213-B</p> <p>C. Wykonanie ST214-C</p>	<p>ST216: Dodaj model samochodu – porażka (NEGATYWNY)</p> <p>A. Wykonanie ST214-A</p> <p>B. Wykonanie ST214-B</p> <p>C. Wykonanie ST213-C</p>

Rysunek 12.4: Scenariusze testowe dla przykładowego przypadku użycia

Na rysunku 12.4 widzimy dwa scenariusze pozytywne oraz dwa negatywne. **Scenariusz pozytywny** ma na celu zbadanie zachowania systemu, które jest pożądane z punktu widzenia wymagań użytkowników (oprogramowanie wykonuje coś, co powinno być wykonane). **Scenariusz negatywny** ma na celu upewnienie się, że system nie wykazuje zachowań niezgodnych z wymaganiami (oprogramowanie nie wykonuje czegoś, co nie powinno być wykonane). Przykładowo, scenariusz ST2134 jest scenariuszem pozytywnym badającym zachowanie systemu, kiedy przypadek użycia „Dodaj model samochodu” powinien zakończyć się sukcesem (prawidłowym dodaniem samochodu). W pierwszym etapie (punkt „A” scenariusza) uruchamiamy scenariusz główny przypadku użycia „Wyświetl listę modeli samochodów”. Upewniamy się, że lista jest pusta. Potem, wykonywany jest scenariusz główny przypadku dodawania samochodu (punkt „B”). W jego trakcie wprowadzane są dane testowe, które zdefiniowane są w odpowiednim pliku („modele_poprawne.txt”). Jeśli wszystko przebiega prawidłowo, system powinien wyświetlić komunikat o sukcesie operacji. Nie jest to jednak dowód poprawności działania systemu. Musimy ponownie uruchomić przypadek użycia wyświetlający listę modeli (punkt „C”). Powinniśmy wtedy uzyskać efekt w postaci wyświetlenia listy modeli samochodów zapełnionej odpowiednio wcześniej wprowadzonymi modelami samochodów. Oczywiście, scenariusz ST213 powinien być wykonany wiele razy, dla wszystkich zestawów danych zawartych w pliku „modele_poprawne.txt”.

Drugi scenariusz testowy jest również pozytywny, lecz sprawdza działanie systemu w przypadku wprowadzenia przez użytkownika nieprawidłowych danych. Zwrócić uwagę na to, że o tym, czy jest to test pozytywny, czy negatywny nie decyduje to, czy testowany przypadek użycia ma się zakończyć sukcesem. W naszym przykładzie, scenariusz ST214 jest pozytywny, mimo, że dodanie modelu samochodu kończy się niepowodzeniem. Testuje on bowiem prawidłowe działanie systemu – jego reakcję na błędnie wprowadzone dane. Wykonanie tego scenariusza jest podobne jak scenariusza ST213, lecz w jego trakcie wprowadzane są dane z innego pliku – „modele_niepoprawne.txt”. Są to dane wychodzące

poza prawidłowy zakres danych, np. przygotowane metodą warunków granicznych. Prawidłowe zachowanie systemu powinno oczywiście polegać na tym, że w wynikowa lista modeli samochodów nie zawiera dodawanego modelu samochodu.

Scenariusze ST215 i ST216 są scenariuszami negatywnymi. Ich wykonanie jest takie samo jak scenariuszy ST213 i ST214, lecz zakończenie jest zmienione. Jeśli system zachowa się zgodnie z jednym z tych scenariuszy – oznacza to błąd. Przykładowo, błędem jest, jeśli po dodaniu nowego modelu samochodu (punkt „B” scenariusza ST215, odpowiadający punktowi „B” z ST213) nie zostanie on zawarty na liście wyświetlanej na koniec scenariusza (punkt „C”, odpowiadający punktowi „C” z ST214).

Tak przygotowane scenariusze testowe powinny pokrywać całą funkcjonalność opisaną przypadkami użycia wykonywanymi w danej iteracji projektu. Pokrycie testami powinno być zgodne z zasadami przedstawionymi na początku rozdziału. Podstawową zasadą jest oczywiście uwzględnienie wszystkich przypadków użycia oraz uruchomienie każdego ich scenariusza.

Zadania

Zadanie 1

Napisz test akceptacyjny metodą czarnej skrzynki dla przypadku użycia „Dodanie nowego użytkownika”. Załącz określone scenariusze odpowiednich przypadków użycia, niezbędnych do wykonania pełnego testu.

Zadanie 2

Podaj zakres danych wejściowych dla testów zadanego modułu (np. mnożenie liczb ze znakiem) metodą czarnej skrzynki.

Zadanie 3

W jaki sposób można przetestować przypadek użycia z zadania 1 metodą białej skrzynki?

Zadanie 4

Zaprojektuj interfejs zawierający dwie operacje z jednym oraz dwoma parametrami numerycznymi oraz typu napisowego. Zaprojektuj test wykorzystujący metodę wartości granicznych dla komponentu realizującego zaprojektowany interfejs. W ramach projektu testu podaj zbiór danych testowych składający się z 5 elementów.

Słownik pojęć

Błąd oprogramowania

Odstępstwo oprogramowania od ustalonej specyfikacji.

Pluskwa

Usterka (błąd oprogramowania) umiejscowiona w kodzie programu i wynikająca ze zmęczenia, przeoczeń bądź nieuwagi programistów.

Scenariusz pozytywny

Rodzaj scenariusza testowego, który ma na celu zbadanie zachowania systemu, które jest pożądane z punktu widzenia wymagań użytkowników (oprogramowanie wykonuje coś, co powinno być wykonane).

Scenariusz negatywny

Rodzaj scenariusza testowego, który ma na celu upewnienie się, że system nie wykazuje zachowań niezgodnych z wymaganiami (oprogramowanie nie wykonuje czegoś, co nie powinno być wykonane).

Scenariusz testowy

Rodzaj testu akceptacyjnego lub integracyjnego, który polega na wykonaniu sekwencji kolejnych kroków interakcji użytkownika z systemem, prowadzących do uzyskania określonego efektu.

Test

Przeprowadzenie badania poprawności działania oprogramowania, czyli sprawdzenie zgodności oprogramowania z wymaganiami.

Test akceptacyjny

Test mający dowieść zgodności systemu z wymaganiami, co pozwoli na jego ostateczny odbiór (akceptację) przez klienta.

Test białej (szklanej) skrzynki

Test wykorzystujący wiedzę o kodzie programu, polegające na badaniu kodu poprzez formalne i nie-formalne przeglądy lub poprzez analizę raportów narzędzi-analizatorów kodu.

Test czarnej skrzynki

Test bazujący całkowicie na monitorowaniu danych wejściowych i wynikowych oraz porównywaniu ich z oczekiwaniemi (założeniami).

Test integracyjny

Test, który sprawdza poprawną współpracę wielu różnych elementów. Test taki polega np. na sprawdzeniu poprawności komunikacji poprzez wcześniej ustalone interfejsy lub prawidłowości współpracy wszystkich elementów systemu koniecznych do dostarczenia poszczególnych funkcjonalności.

Test jednostkowy

Test sprawdzający działanie elementarnych składników kodu, np. klas oraz ich metod. Sprawdzana jest logika działania procedury lub funkcji poprzez zadanie danych wejściowych (parametrów) i sprawdzenie, czy dane wyjściowe (wyniki) są zgodne z oczekiwaniemi.

Test modułowy

Rodzaj testu jednostkowego, w którym testowana jest większa liczba współpracujących ze sobą elementów kodu naraz (np. cały komponent).

Test regresyjny

Test powtarzany w identyczny sposób w kolejnych iteracjach projektu, mający na celu sprawdzenie, czy wprowadzone w kodzie zmiany (np. optymalizacja, refaktoryzacja) nie spowodowały powstania nowych błędów.

Co trzeba zapamiętać

Rola testowania

Testowanie oprogramowania jest dyscypliną inżynierii oprogramowania zajmującą się określaniem metod oraz przeprowadzaniem badania poprawności działania oprogramowania. Odstępstwa od ustalonych wymagań odnośnie systemu nazywamy błędami oprogramowania. Obejmują one sytuacje, kiedy oprogramowanie nie wykonuje czegoś, co powinno wykonywać, oprogramowanie robi coś, czego nie powinno robić, oprogramowanie nie wypełnia warunków dotyczących cech pozafunkcjonalnych, oprogramowanie zachowuje się w sposób nie przewidziany specyfikacją. Istotną część błędów stanowią pluskwy – usterki umiejscowione w kodzie programu i wynikające ze zmęczenia, przeoczeń bądź nieuwagi programistów. Nie jesteśmy w stanie udowodnić prawidłowości działania systemu poprzez sprawdzenie wszystkich możliwych scenariuszy dla wszystkich możliwych danych. Potrzebne jest uzyskanie odpowiedniej równowagi między środkami poświęconymi na testowanie, a uzyskanym efektem (jakością oprogramowania).

Podstawowe metody testowania

Najczęściej używanymi metodami testowania są metody kierujące się zasadą testów czarnej skrzynki. By przeprowadzić udany test metodą czarnej skrzynki należy wykorzystać techniki weryfikacji oprogramowania bazujące na zasadach nieingerencji i nieznajomości kod programu. Należą do nich metody tworzenia klas równoważności, warunków granicznych, zmian stanów oraz niedoświadczonego użytkownika. Przeciwagą dla metod czarnej skrzynki są metody szklanej (białej) skrzynki. Wykorzystują one wiedzę o kodzie programu w celu podniesienia skuteczności testów. Testy bazujące na wglądzie w strukturę programu polegają przede wszystkim na badaniu kodu – czy to podczas formalnych i nieformalnych przeglądów czy poprzez analizę raportów narzędzi-analizatorów kodu.

Poziomy testowania

Testy możemy skategoryzować w zależności od ich rodzaju oraz etapu procesu wytwarzania oprogramowania, na którym należy zacząć je stosować. Różne rodzaje testów mogą być i typowo są wykonywane przez osoby pełniące różne role w projekcie. Na przykład, testy jednostkowe wykonywane są przez programistów, a testy akceptacyjne – przez dedykowanych testerów we współpracy z użytkownikami. Najbardziej szczegółowym rodzajem testów są testy jednostkowe. Dotyczą one kodu systemu i są tworzone bezpośrednio przez programistów go piszących. Rolą testów jednostkowych jest sprawdzenie działania elementarnych składników kodu, np. klas oraz ich metod. Testy regresyjne są to testy powtarzane w identyczny sposób w kolejnych iteracjach projektu, mające na celu sprawdzenie, czy wprowadzone w kodzie zmiany nie spowodowały powstania nowych błędów. Testy kodu w których testujemy większą liczbę współpracujących ze sobą elementów naraz (np. cały komponent) nazywamy testami modułowymi. Rolą testami integracyjnymi jest sprawdzanie poprawnej współpracy wielu różnych elementów. W skład tej kategorii wchodzą również testy współdziałania i testy systemowe. Najwyższym poziomem testów są testy akceptacyjne, które mają dowieść zgodności systemu z wymaganiami, co pozwoli na jego ostateczny odbiór przez klienta.

Testowanie przypadków użycia

Testowanie akceptacyjne najczęściej polega na testowania działania przypadków użycia i jest wykonywane na poziomie testów systemowych i testów akceptacyjnych. Testy polegają na projektowaniu, a następnie wykonywaniu scenariuszy testowych. Scenariusz testowy opisuje kolejne kroki interakcji użytkownika z systemem, prowadzące do uzyskania określonego efektu. Scenariusz pozytywny ma na celu zbadanie zachowania systemu, które jest pożądane z punktu widzenia wymagań użytkowników. Scenariusz negatywny ma na celu upewnienie się, że system nie wykazuje zachowań niezgodnych z wymaganiami.

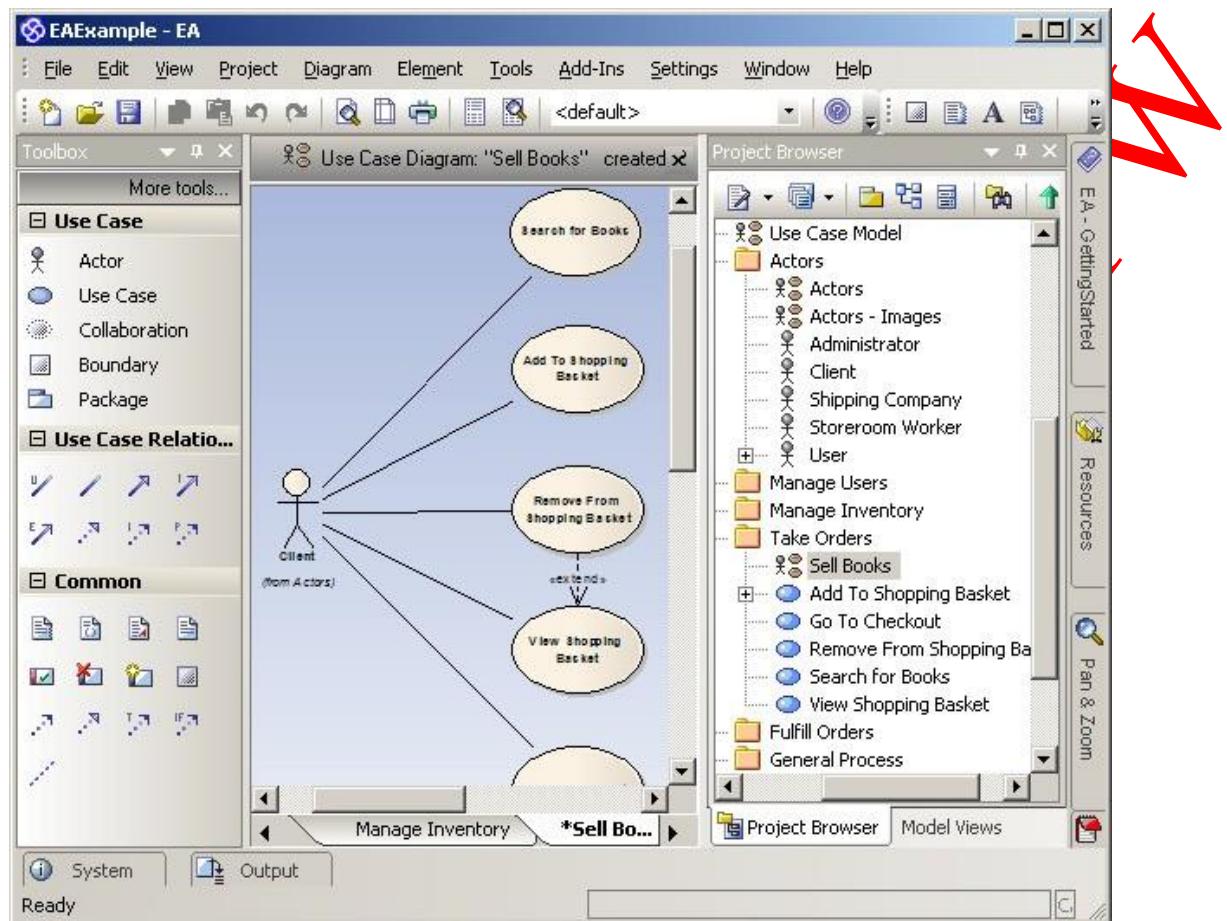
13. Narzędzia i metody automatyzacji inżynierii oprogramowania

Jak już wspomniano w poprzednich rozdziałach, w procesie wytwarzania oprogramowania jednym z najważniejszych zagadnień jest opanowanie ogromu informacji towarzyszącej powstającym systemom. Na wiedzę gromadzoną w trakcie trwania projektów składają się dokumenty powstałe przy współpracy z klientem, modele, projekty, kod, testy, dokumentacja, podręczniki, multimedia, wersje produktów itd. Aby praca informatyków była nie tylko wydajna i zakończona sukcesem, ale w ogóle możliwa, niezbędne jest powstanie odpowiedniego środowiska, w którym te tworzone artefakty są właściwie uporządkowane. Takie środowisko pracy powinno pozwalać nie tylko na łatwą wymianę informacji między uczestnikami projektu, ale również na automatyzację możliwie wielu czynności na wszystkich etapach procesu wytwórczego.

W celu usprawnienia pracy zespołów konstruujących systemy oprogramowania powstały różnego rodzaju narzędzia, które nazywamy narzędziami **komputerowego wsparcia inżynierii oprogramowania CASE** (ang. Computer Aided Software Engineering). Narzędzia te pozwalają usprawnić czynności związane z wieloma aspektami rozwoju produktów systemowych: od zarządzania projektami informatycznymi, przez integrację powstających modułów i zarządzanie zmianami, po wsparcie dla modelowania systemów na różnych poziomach i ułatwianie implementacji. W kolejnych sekcjach omawiamy różne rodzaje narzędzi oraz metody ich stosowania w celu poprawy efektywności wytwarzania oprogramowania.

13.1. Narzędzia automatyzacji analizy i projektowania oprogramowania

Narzędzia wspierające modelowanie obiektowe spełniają bardzo istotną rolę w „warsztacie” twórców oprogramowania. Zgodnie z tym, o czym mówimy w modułach II i III, modelowanie oprogramowania dotyczy praktycznie wszystkich faz rozwoju produktów software'owych. Podstawową funkcjonalnością typowego narzędzia służącego do modelowania obiektowego jest wsparcie dla rysowania diagramów w wybranej notacji, szczególnie – języka UML, ale często również innych, takich jak BPMN czy ERD. Podstawowe wsparcie narzędzia polega na udostępnieniu użytkownikowi możliwości łatwego tworzenia diagramów, modyfikacji elementów diagramów oraz ich przeglądania. Rysunek 13.1 przedstawia wygląd typowego interfejsu użytkownika. Po lewej stronie widoczny jest przybornik z elementami modelu, w części środkowej znajduje się edytowany diagram, a po prawej podgląd struktury modelu (drzewo modelu). Oczywiście, układ interfejsu użytkownika może być dowolnie zmieniany.



Rysunek 13.1: Przykładowe narzędzie wspierające modelowanie

Zaletą narzędzi modelowania jest to, że „pielęgnowa” **poprawności notacji** nie pozwalaając użytkownikowi na umieszczenie na diagramie danego typu, niewłaściwych elementów czy na nieprawidłowe zapisanie konstrukcji języka. Istotną funkcją narzędzi modelowania jest także umożliwienie zarządzania modelem: odpowiedniego porządkowania ich struktury (np. w formie pakietów), a także wygodnego przehodzenia między poziomami abstrakcji i śledzenia zależności między fragmentami systemów opisanych z różnym stopniem szczegółowości. Cechą ta jest niezmiernie istotna, ponieważ systemy reprezentowane są przez modele na wielu poziomach abstrakcji: poczynając od wymagań, a kończąc na „mapach kodu” – modelach projektowych.

Istotnym elementem funkcjonalności narzędzi modelowania jest inżynieria kodu. Korzystając z funkcji z nią związanych można **generować szkielet kodu** odzwierciedlający wiedzę zawartą w modelu („inżynieria w przód”), tworzyć modele w celu zrozumienia istniejącego kodu („inżynieria odwrotna”), a także wspierać proces ciągłej synchronizacji kodu z modelem. Narzędzia najczęściej pozwalają na szeroką parametryzację tych procesów, nie tylko przez określenie docelowego języka programowania, ale również poprzez określenie sposobu interpretacji konstrukcji języka modelowania podczas generowania kodu.

Ważnym zagadnieniem związanym z tworzeniem modeli jest udostępnianie efektów pracy pozostałym osobom uczestniczącym w projekcie w dogodnej dla nich formie. Większość narzędzi pozwala na **generację dokumentacji modeli**. Jest to, podobnie jak inżynieria kodu, proces parametryzowalny. Dostępna jest mnogość formatów docelowych dokumentów (np. statyczny lub dynamiczny HTML, pliki RTF, pliki graficzne itd.). Możliwa jest także konfiguracja zakresu generowanej treści, a także często istnieje wsparcie dla tworzenia szablonów dokumentów. Niektóre, bardziej zaawansowane narzędzia

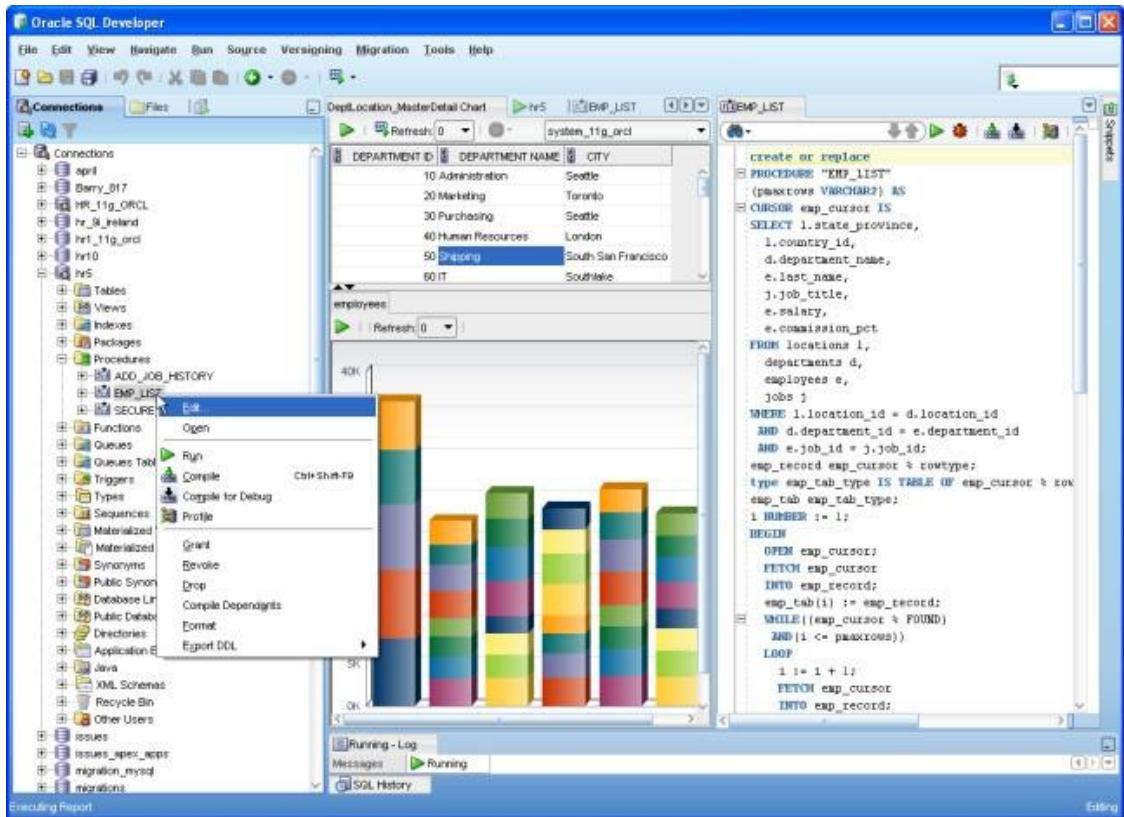
umożliwiają wymianę modeli za pośrednictwem systemów kontroli wersji stosowanych przy zarządzaniu kodem. Zwiększa to efektywność pracy grupowej nad modelami.

Czasami zachodzi potrzeba wymiany modeli między zespołami korzystającymi z różnych narzędzi modelowania dla języka UML. W takiej sytuacji stosowany jest standardowy format XMI (ang. *XML Meta-data Interchange*) oparty na języku XML. Niestety współpraca narzędzi CASE poprzez wymianę plików XMI w praktyce okazuje się dużym problemem. Wytwórcy oprogramowania odmiennie interpretują i korzystają ze standardu, a mnogość wersji języka UML i różne sposoby jego reprezentacji wewnętrznej w narzędziach powodują częsty brak kompatybilności wymienianych plików.

Modelowanie w języku UML dotyczy przede wszystkim logiki systemu i jego kodu. Równie istotnym aspektem jest modelowanie oraz implementacja i utrzymywanie baz danych. Do tego celu służą **narzędzia do zarządzania bazami danych**. Samo modelowanie struktury bazy danych odbywać się może w sposób podobny do modelowania obiektowego – przy wykorzystaniu notacji ERD lub UML. Narzędzia CASE dedykowane bazom danych dostarczają wielu funkcji ułatwiających budowę systemów bazodanowych. Narzędzia takie pozwalają na wygodne tworzenie kopii zapasowych, importowanie lub eksportowanie danych, kreację raportów dotyczących danych i ich schematów, ustalanie opcji wydajności, a także tzw. „strojenie” (ang. *tunning*), czyli optymalizację wydajności bazy danych. Niektóre posiadają także funkcje podobne do tych dostarczanych przez narzędzia typu IDE (patrz niżej). Umożliwiają np. uruchamianie poleceń administracyjnych albo związanych z danymi czy odpluskwanie (ang. *debugging*) przetwarzanych przez silnik bazy skryptów. Często dostępne w tego typu narzędziach są opcje analizy danych pozwalające na wizualne tworzenie zapytań (odpowiadających zapytaniom w języku SQL).

Rysunek 13.2 przedstawia widok przykładowego narzędzia zarządzania bazami danych. Umożliwia ono przeglądanie struktury instancji baz danych (lewy panel), specyfikacji konkretnej tabeli (środek-góra), tworzenie raportów (środek-dół) oraz pisanie kodu (prawy panel). Warto tutaj zauważyć, że systemy zarządzania bazami danych posiadają możliwości uruchamiania kodu logiki dziedzinowej bezpośrednio w ramach silnika bazy danych. Nie jest to jednak zalecane podejście i współcześnie rzadko stosowane. Niektóre narzędzia pozwalają dodatkowo na tworzenie kodu logiki aplikacji i interfejsu użytkownika opartego o wybrane schematy danych. Taki generowany kod podlega oczywiście parametryzacji, a generator pozwala na automatyczne stworzenie szablonów interakcji.

Do użytku



Rysunek 13.2: Przykładowe narzędzie wsparające zarządzanie bazami danych

Często używaną grupą narzędzi są narzędzia do **modelowania interfejsu użytkownika**. Narzędzia takie pozwalają na tworzenie szkiców interfejsu użytkownika, nazywanych **interfejsami szkieletowymi** (ang. wireframe). W podstawowej swej funkcjonalności zastępują one kartkę papieru i ołówek – pozwalają przede wszystkim na zobrazowanie rozmieszczenia elementów interfejsu użytkownika. Interfejs szkieletowy nie oddaje ostatecznego wyglądu produktu, lecz jedynie odzwierciedla jego kontury. Z takiego szkicu nie dowiemy się np. jaki będzie kolor elementów ekranowych czy styl użytych czcionek. Główną zaletą takiego podejścia do modelowania interfejsu użytkownika jest możliwość koncentracji na meritum, czyli właściwej treści prezentowanej użytkownikom. Pozwala to zidentyfikować ograniczenia struktury interfejsu użytkownika i odpowiednio rozplanować wyświetlanie różnych typów informacji. Komunikację z użytkownikiem ułatwia to, że dyskusja dotyczy układu elementów, a nie ich wyglądu graficznego (kolor, czcionka itp.).

Na dalszych etapach modelowania interfejsu użytkownika możemy użyć narzędzi do tworzenia **makiety** (ang. mockup). Narzędzia takie mogą być zintegrowane z narzędziami do tworzenia interfejsów szkieletowych. Wystarczy wtedy po prostu przełączyć tryb modelowania. Makieta umożliwia przedstawienie docelowego wyglądu interfejsu użytkownika, łącznie ze wszystkimi szczegółami graficznymi (kształt elementów, kolor, ramki, czcionki itp.). Od rzeczywistego interfejsu użytkownika makieta różni się tym, że jest statyczna – nie posiada funkcjonalności, a wyświetlane elementy są tylko przykładowe. Niektóre narzędzia do makietowania pozwalają na symulację nawigacji między poszczególnymi ekranami. Pozwala to na lepsze zorientowanie się w funkcjonowaniu modelowanego interfejsu użytkownika.

Warto podkreślić, że narzędzia do modelowania interfejsu użytkownika nie służą do implementacji. Czasami narzędzia tego typu umożliwiają wygenerowanie elementów kodu, na bazie którego można

stworzyć gotowy prototyp działający w docelowym środowisku. Jednak cel podstawowy takich narzędzi to umożliwienie szybkiego przedyskutowania i zatwierdzenia wyglądu aplikacji. Implementacja tak opracowanego wyglądu jest zadaniem zespołu deweloperskiego. Używa on w tym celu odpowiednich narzędzi wsparcia implementacji, o których mówimy poniżej.

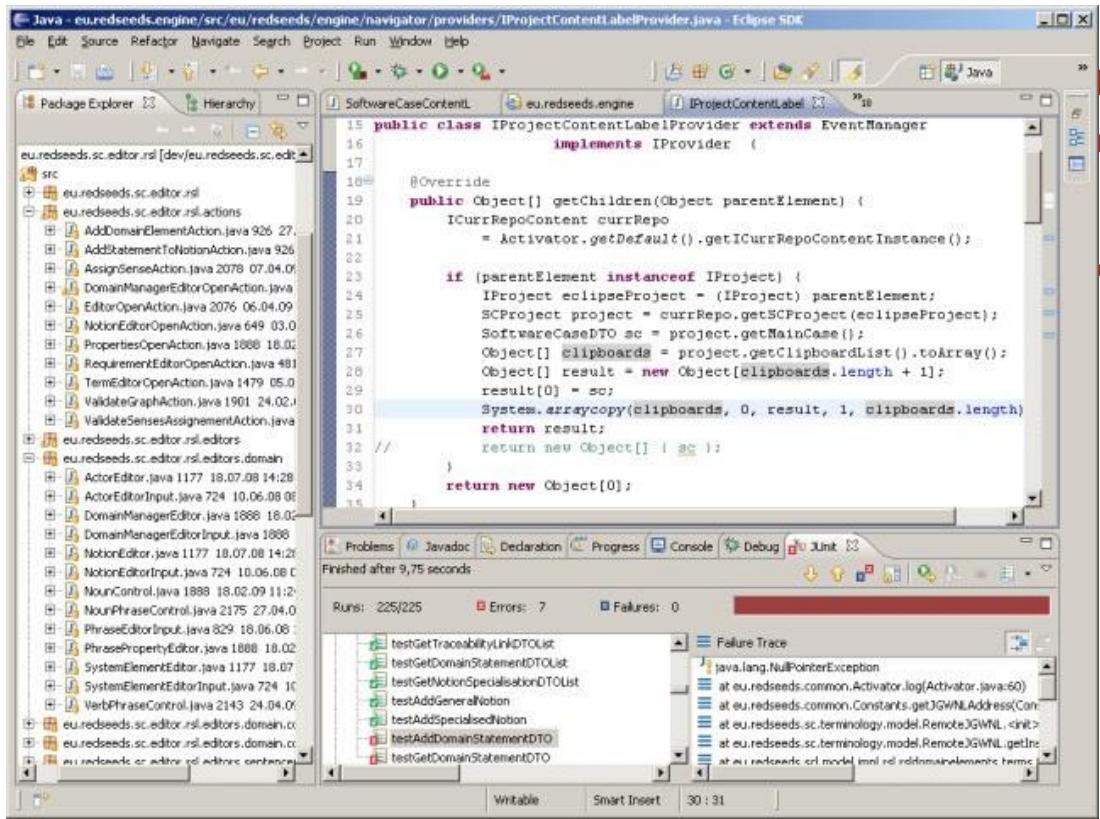
13.2. Narzędzia wsparcia implementacji i testowania oprogramowania

Klasą narzędzi CASE używaną w praktycznie wszystkich projektach są narzędzia wspierające tworzenie kodu. Jest to bardzo rozwinięta grupa, posiadająca wiele funkcji ułatwiających życie programistom. Podstawowym zadaniem takich narzędzi jest zwiększenie produktywności przez zintegrowanie wszystkich funkcjonalności używanych w typowym cyklu życia kodu w jedną spójną całość. Narzędzia takie nazywamy **zintegrowanymi środowiskami programistycznymi** (ang. IDE – Integrated Development Environment).

Podstawowym elementem IDE jest **edytor kodu**. Współczesne edytory kodu posiadają wiele funkcji pomagających w pisaniu poprawnego składniowo kodu. Typowo pozwalają one kolorować kod (różnić różne elementy składni kolorami), dokonywać automatycznego formatowania (np. wcinanie tekstu), wykonywać autouzupełnianie (np. podpowiadanie nazw zmiennych) i inne funkcje edytorskie. Część środowisk zintegrowanych pozwala traktować kod nie tylko jako tekst, ale pewną strukturę. Kod może być przeglądany w postaci prostych diagramów klas, a jego hierarchia przedstawiona jako struktura drzewiasta, którą można poddawać łatwej edycji. Wiele spośród IDE posiada opcję wstawiania często wykorzystywanych fragmentów programów (ang. snippets), które można parametryzować – są to konstrukcje takie jak pętle, polecenia sterujące czy odpowiedzialne za obsługę wyjątków. Do kategorii automatycznego przetwarzania struktury kodu należą też opcje refaktoryzacji kodu (ang. code refactoring) pozwalające na zmianę sposobu organizacji kodu – od prostego przemianowania klas (odniesienia do refaktoryzowanej klasy są automatycznie zmieniane w zawierającym je kodzie) po złożone operacje na strukturze pakietów czy hierarchii klas. Inną użyteczną operacją dostępną w popularnych IDE jest generowanie szkieletów kodu np. dla klas implementujących określone interfejsy.

Prawdziwa siła wsparcia, jakie IDE udzielają programiście leży jednak w tym, w jaki sposób łączą się one z narzędziami zewnętrznymi. Ścisła integracja z **kompilatorami** pozwala na natychmiastowe wskazywanie błędów w kodzie, a często nawet automatyczne ich poprawianie. Popularne narzędzia modelowania umożliwiają współpracę z IDE, czy to przez udostępnienie funkcjonalności związanych z modelowaniem w samych IDE, czy przez możliwość edycji kodu modelowanych elementów w środowisku modelowania. Inną, standardową już dzisiaj funkcjonalnością, jest wygodna współpraca z różnymi repozytoriami kodu i systemami wersjonowania. Środowiska programistyczne ułatwiają także tworzenie środowisk testowych przez automatyczną generację szkieletu skryptów weryfikujących aplikację oraz możliwość analizy i kontroli uruchomienia już zaimplementowanych testów.

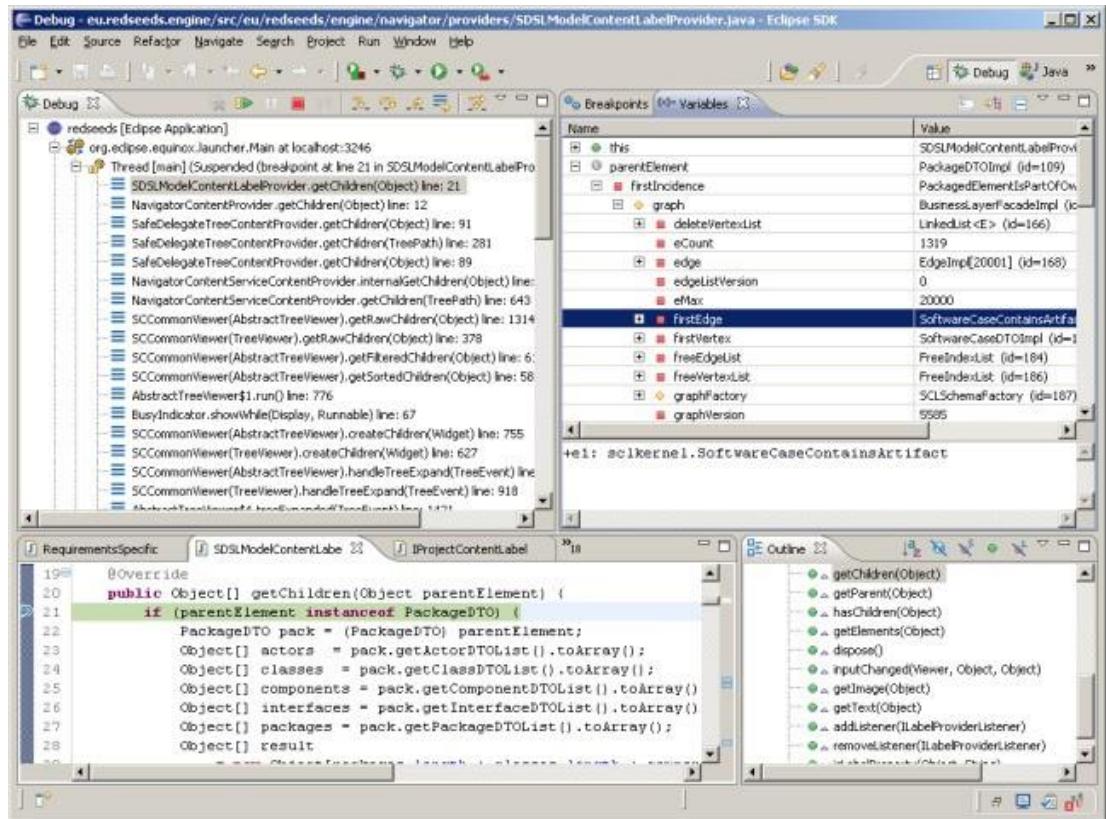
Rysunek 13.3 ilustruje przykładowe środowisko zintegrowane (tu: Eclipse). Na lewym panelu widoczna jest przeglądarka hierarchii kodu, na górnym środkowym panelu widzimy edytor kodu, na dolnym środkowym panelu – efekt działania wtyczki testów jednostkowych. Widoczne są także zakładki dla wielu innych narzędzi.



Rysunek 13.3: Widok IDE Eclipse

Ważnym narzędziem w obrębie IDE jest tzw. **odpluskwiacz** (ang. debugger) pozwalający na analizę programów podczas ich wykonywania. Uruchomiony w ramach odpluskwiacza proces może być zatrzymywany w czasie wykonywania wybranych przez programistę instrukcji czy też wykonywany krokowo. Debugger pozwala na obejrzenie stanu pamięci przypisanej do tworzonego programu. Sam proces odpluskwiacza pozwala na rozwiązywanie problemów, których przezwyciężenie tradycyjnymi metodami (np. poprzez analizę kodu) jest bardzo pracochłonne. Na rysunku 13.4 przedstawiono debugger uruchomiony w środowisku Eclipse. Widoczny jest podgląd kodu z zaznaczonym aktualnie przetwarzanym wierszem (lewo-dół), uruchomione procesy i związane z nimi instancje klas (lewo-góra) oraz podgląd stanu zmiennych programu (prawo-góra).

Do użytku

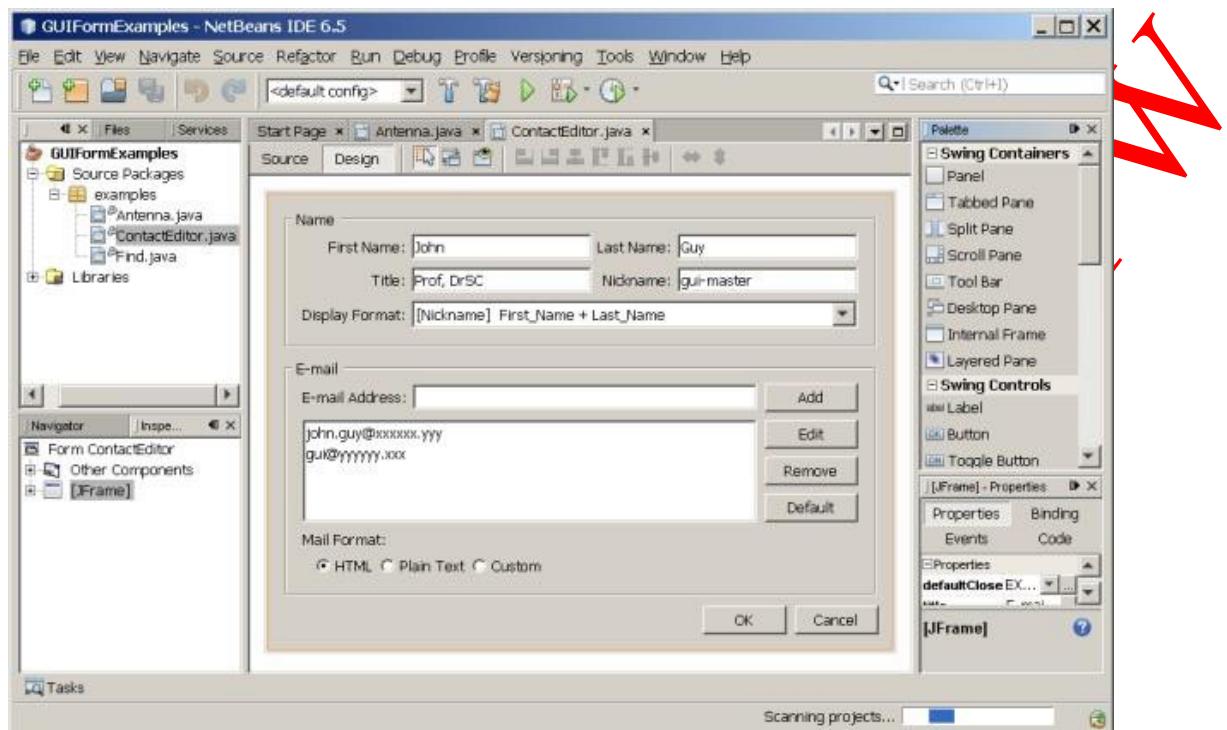


Rysunek 13.4: Debugger uruchomiony w środowisku Eclipse.

Nowoczesne narzędzia klasy IDE posiadają często wsparcie dla **prototypowania graficznych interfejsów użytkownika**. Takie wsparcie polega na umożliwieniu wygodnego rysowania projektu GUI za pomocą zestawu elementów-kontrolek, takich jak przyciski, panele, listy itd. Gdy użytkownik tworzy widoki związane z GUI, jego środowisko automatycznie generuje kod pozwalający na wyświetlenie projektowanych ekranów, a także obsługę zdarzeń – akcji użytkownika. Edytory GUI dla technologii internetowych pozwalają także na wygodne projektowanie stron WWW połączonych z logiką aplikacji, realizowaną przez pisany kod.

Kolejnym rodzajem narzędzi, z jakimi integrują się środowiska programistyczne są serwery aplikacji na jakich uruchamiany jest tworzony produkt. Współpraca IDE z docelowymi bądź testowymi środowiskami uruchomieniowymi polega na wsparciu dla łatwej kontroli serwera aplikacji z poziomu narzędzi programistycznych, najczęściej poprzez umożliwienie zatrzymania, uruchomienia i reinicjacji serwera, a także poprzez otwarcie dla zapisu katalogów, do których wgrywane są kompilaty, pliki konfiguracyjne i inne zasoby.

Kombinacja generatorów kodu (nie tylko związanych z modelami) i edytorów GUI pozwala na **szbkie prototypowanie aplikacji** (RAD – ang. *Rapid Application Development*). Wczesne utworzenie aplikacji uruchamialnych i posiadających zbliżony do docelowego interfejs użytkownika pozwala na dyskusje z odbiorcą oprogramowania na temat jego docelowego kształtu, zanim w rozwój zainwestowane zostaną poważne zasoby. Rysunek 13.5 przedstawia tworzenie prototypu interfejsu użytkownika w środowisku NetBeans. Widoczny jest przybornik z elementami UI i właściwości wybranych elementów na prawym panelu oraz podgląd tworzonego formularza na środkowym.



Rysunek 13.5: Tworzenie prototypu interfejsu użytkownika w środowisku NetBeans

Praca z kodem skutkuje powstaniem odpowiednim modułów wykonywalnych, które wspólnie tworzą gotowy system. Istotnym problemem, szczególnie w większych projektach software'owych jest integracja, czyli procesu kombinacji systemu ze stale aktualizowanych modułów. W rozproszonym systemie pracy grupowej lokalna integracja przeprowadzana przez każdego uczestniczącego w procesie często okazuje się praco- i czasochłonna. Każdy pracujący z kodem musi zaktualizować swoją roboczą kopię systemu, skompilować ją, uruchomić, przetestować, wprowadzić własne zmiany, ponownie skompilować, przetestować i dopiero zapisać zmiany w repozytorium centralnym. W przypadku większych projektów liczba zmian zachodzących w projekcie okazuje się być tak duża, a projekt tak skomplikowany, że czas integracji poważnie wpływa na efektywność całego zespołu. Ponadto, integracja dokonywana lokalnie może powodować niespójność systemu spowodowaną zastosowaniem zdezaktualizowanych wersji modułów.

Aby wyeliminować problemy z integracją lokalną wprowadzono zcentralizowane systemy **ciągłej integracji** (ang. continuous integration). Są one najczęściej ściśle powiązane z systemami **ciągłego wdrażania** (ang. constant deployment). Centrum takich systemów są odpowiednie narzędzia, które potrafią automatycznie – na bazie przygotowanych wcześniej modułów – stworzyć działające wersje rozwijanego produktu. Takie działające wersje (wydania) systemu mogą być następnie w sposób automatyczny zainstalowane (wdrożone) w środowisku wykonawczym (testowym lub produkcyjnym). Przed instalacją przeprowadzane są – również automatycznie – zestawy testów mających na celu sprawdzenie, czy system działa prawidłowo.

Z integracją systemów ściśle powiązane jest zagadnienie zarządzania wersjami. Zasada działania narzędzi do wersjonowania oprogramowania zostały omówione w pierwszym rozdziale niniejszego modułu. Wersjonowane repozytorium jest bazą, na której swoje funkcjonowanie opierają systemy ciągłej integracji, ale również systemy **zarządzania konfiguracją i zmianami**. Służą one do dokumentowania dialogu między osobami tworzącymi produkt a testerami i użytkownikami systemu. Komunikacja ta dotyczy problemów związanych z funkcjonowaniem aplikacji, ale zawierać może też propozycje udoskonalień produktu.

Jednym z istotnych elementów systemów zarządzania zmianami jest narzędzie do śledzenia błędów i problemów (ang. bug and issue tracker). Pozwala ono przede wszystkim na dokładne opisanie przykładów niepożądanego działania systemu oraz postulowanych zmian funkcjonalności. Użytkownicy systemu, poprzez odpowiedni interfejs (najczęściej sieciowy) wypełniają formularze opisujące błędy lub problemy. Opis taki, poza tekstem dokumentującym zagadnienie, zawiera szereg atrybutów usprawniających zarządzanie ogółem zmian projektu, np. identyfikatory wydania i modułu programu, w których spostrzeżono problem, poziom krytyczności błędu, osoby i zagadnienia związane z danym wpisem. Na podstawie sformalizowanego opisu łatwo odpowiedzieć na pytania typu „w którym komponente jest najwięcej poważnych usterek?”, „który programista ma przypisanych najwięcej zadań?”, „które wydanie wymagało najmniej poprawek?”, „które komponenty wymagały zmiany po rozwiązaniu zagadnienia Z?”. Większość popularnych systemów śledzenia zagadnień pozwala na generację raportów opartych o informacje zawarte w opisach.

Odpowiednio opisane zagadnienia rozsypane są do osób odpowiedzialnych za dane moduły albo wydania w celu poddania procedurze obsługi zmiany. Warto podkreślić, że narzędzia do śledzenia błędów i problemów pozwalają na organizację przepływu informacji, dokumentów i pracy. Pozwalają one na wygodną i jednocześnie formalną komunikację między przedstawicielami producenta oprogramowania a użytkownikami aplikacji. Wymiana informacji jest dokumentowana, a jej przebieg można skonfigurować, dostosowując do schematu przepływu pracy charakterystycznego dla danego projektu lub organizacji.

Ze śledzeniem błędów powiązane jest oczywiście wykrywanie błędów. Głównym sposobem wykrywania błędów jest testowanie, które zostało omówione w poprzednim rozdziale. Istnieje szereg narzędzi pozwalających na usprawnienie procesu testowania. Umożliwiają one automatyzację powtarzalnych i kosztownych operacji badania rozmaitych aspektów poprawności programów.

Jedną z podstawowych funkcjonalności narzędzi wspierających testowanie jest możliwość **testowania interfejsu użytkownika** (ang. UI testing). Tego typu testowanie pozwala na zbadanie reakcji interfejsu użytkownika, poprawności jego struktury (zachowanie pozycji elementów na ekranie), a także przejść między stanami aplikacji (np. kolejność pokazywanych ekranów). Podstawą jest tutaj zapisywanie (nagrywanie) typowych przebiegów interakcji użytkowników z badanym systemem. Proces rejestracji dialogu użytkownik-aplikacja polega na zapisie decyzji użytkownika (naciśniętych przez niego przycisków, wybranych opcji menu, wprowadzonych danych itd.). Taki zapis (akcje użytkownika oraz ich parametry) może być później modyfikowany.

Zapisy działań użytkowników mogą być dokonywane zgodnie ze specyfikacją scenariuszy przypadków testowych. Tak zapisane scenariusze mogą być podstawą testów regresyjnych na poziomie testowania akceptacyjnego. Użycie takich narzędzi znaczco obniża koszty weryfikacji systemu w procesie jego ciągłego rozwoju. Każde kolejne uruchomienie procedury testowej opiera się na już raz wykonanej (i zarejestrowanej) pracy testera. Narzędzia pozwalają także na rozdzielenie akcji użytkownika (czyli np. kliknąć myszą czy wybranych opcji) i związanych z nimi danych (czyli np. wprowadzonych łańcuchów znaków). Podczas odtwarzania zarejestrowanych scenariuszy, akcjom użytkownika mogą towarzyszyć różne zestawy danych, co pozwala na dokładne pokrycie systemu testami.

Popularne narzędzia pozwalają także na odtwarzanie wielu instancji tych samych scenariuszy testowych jednocześnie. Pozwala to na przetestowanie odporności systemu na określone obciążenia, czyli przeprowadzenie testów obciążeniowych. Zarówno zestawy danych, jak i parametry akcji użytkowników mogą podczas takich testów podlegać losowym modyfikacjom, dzięki czemu symulacja zachowań wielu użytkowników korzystających z systemu jest pełniejsza. „Sztuczni” użytkownicy charakteryzują

się wtedy np. różnymi opóźnieniami między akcjami jakie wykonują, co powoduje bardziej naturalny rozkład sygnałów stymulujących system.

Bardzo popularną grupą narzędzi wspomagających testowanie są **narzędzia dla testów jednostkowych**. Pozwalają one na generację szkieletów testów, kontrolowane uruchamianie skryptów testowych, analizę wyników działania oraz raportowanie. Testy jednostkowe uruchamiane poprzez odpowiednie oprogramowanie pracują w odpowiednio spreparowanym środowisku. Możliwe jest np. określanie danych testowych oddzielnie dla każdego uruchomienia, a także reprezentowane innych modułów przy pomocy zaślepek. Uruchomienie testów jest kontrolowane – kroki przypadków testowych mogą być realizowane w określonej kolejności. Wyniki testów jednostkowych pozwalają na stwierdzenie, który z warunków zapisanych w testach nie został spełniony oraz jaki był rezultat takiego sztucznie wytworzzonego błędu. Dla przeprowadzonego zestawu testów narzędzia generują raporty zawierające statystykę znalezionych błędów, ich rodzaj, zakres kodu jaki został objęty testami itd. Narzędzia wspierające testy jednostkowe łatwo integrują się z popularnymi zintegrowanymi środowiskami programistycznymi (IDE).

O ile powyżej opisane narzędzia działają na zasadzie testów metodą „czarnej skrzynki”, to istnieje także wsparcie narzędziowe dla testów „przezroczystej skrzynki”. Służą do tego różnego rodzaju **analizatory kodu**. Narzędzia tego rodzaju pozwalają na weryfikację zgodności efektów pracy programistów z przyjętymi w ramach danej organizacji dobrymi praktykami tworzenia kodu. Badanie kodu pozwala także na zlokalizowanie możliwych przeoczeń programistów (np. puste lub nieosiągalne bloki programów, nieużywane zmienne, duplikowany kod). Narzędzia tej kategorii pozwalają na przejrzysty zapis reguł opisujących poprawny kod, a następnie analizę wybranych plików i tworzenie raportów opisujących wyniki analizy. Większość środowisk IDE posiada wbudowane w edytory kodu elementarne mechanizmy analizy kodu. Na przykład, pozwalają one kontrolować styl identyfikatorów, czy sprawdzać ich poprawność ortograficzną.

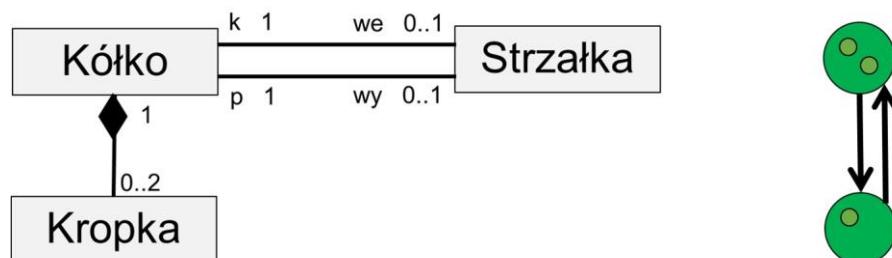
13.3. Metody automatyzacji wytwarzania i eksploatacji

Wszystkie omówione narzędzia można połączyć w cykl pełnej automatyzacji wytwarzania oraz eksploatacji oprogramowania. Na cykl ten składają się dwie podstawowe metody: wytwarzanie oprogramowania sterowane modelami oraz tzw. DevOps.

Wytwarzanie oprogramowania sterowane modelami (WOSM, ang. Model-Driven Software Development) dotyczy przede wszystkim tych dyscyplin inżynierii oprogramowania, które opierają się na wykonywaniu modeli: dyscyplinie wymagań, a także analizie i projektowaniu. W module III podawaliśmy różne reguły, które pozwalają na przekształcanie modeli na jednym poziomie abstrakcji (np. modele wymagań) w modele na innym poziomie (np. modele architektoniczne). Takie przekształcanie oznacza zwiększenie szczegółowości modeli. Na przykład, modele projektowe uwzględniają szczegóły dotyczące operacji oraz ich sygnatur. Podstawą WOSM jest automatyzacja tego procesu poprzez definiowanie i wykonywanie automatycznych transformacji między modelami. Elementem WOSM jest również automatyczna generacja kodu z modeli, która przenosi nas do dyscypliny implementacji systemu.

WOSM dostarcza technik w dwóch obszarach: definiowania języków modelowania w sposób formalny oraz definiowania przekształceń między modelami zdefiniowanymi w tych językach modelowania. Najczęstszym sposobem formalnego zdefiniowania graficznego języka modelowania jest tzw. **metamodel**. Metamodel to „model modelu”, który wyrażamy również w języku graficznym. Najczęściej jest to wariant modelu klas języka UML. Każda meta-klasa w metamodelu wyraża jeden element składowy języka

modelowania. Przykład bardzo prostego metamodelu widzimy na rysunku 13.6. Po lewej stronie rysunku widzimy diagram klas, który definiuje elementarny język modelowania składający się z kropek, kółek i strzałek. Jak wynika z metamodelu, język dopuszcza tworzenie modeli, w których kółka są połączone strzałkami. Z każdego kółka może wychodzić oraz wychodzić co najwyżej jedna strzałka. W kółkach mogą być zawarte kropki, przy czym może ich być maksymalnie dwie. Metamodel nie definiuje wyglądu elementów języka modelowania, a jedynie ich wzajemne relacje. Mówimy, że metamodel definiuje **składnię abstrakcyjną** języka. Po prawej stronie rysunku 13.6 widzimy przykład składni konkretniej, czyli składni widocznej dla użytkowników języka. Jak można się domyślić, na rysunku widzimy dwa kółka połączone dwoma strzałkami. W kółkach znajdują się odpowiednio – jedna oraz dwie kropki.

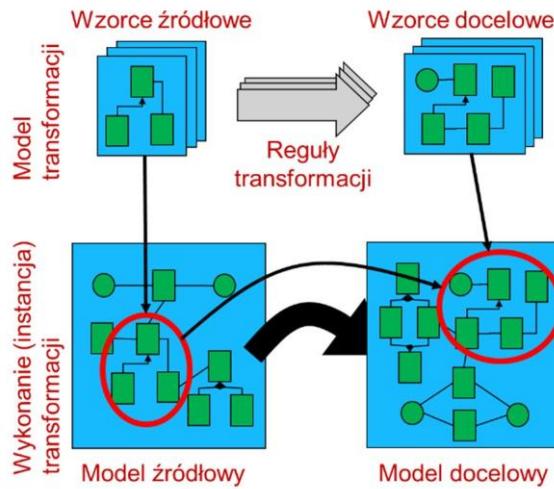


Rysunek 13.6: Przykład metamodelu prostego języka graficznego

Definicję swojej składni wyrażoną w postaci metamodelu posiadają praktycznie wszystkie języki modelowania, takie jak UML, BPMN czy ERD. Pozwala to na automatyczne przetwarzanie modeli. Każdy model jest przechowywany w odpowiednim repozytorium zgodnym z metamodellem. Taki model możemy odczytać przy pomocy odpowiedniego programu, a następnie dokonać automatycznej transformacji w inny model. Przykładem praktycznego zastosowania takiej automatyzacji jest automatyczna generacja struktury relacyjnej bazy danych (język ERD) z modelu dziedzinowego wyrażonego w języku UML.

Aby móc wykonywać transformacje modeli, należy napisać odpowiedni **program transformujący**. Programy takie możemy pisać w standardowych językach programowania, takich jak Java czy C#. Często jednak stosuje się języki dedykowane do transformacji modeli. Wynika to z tego, że języki te posiadają specjalne konstrukcje, które ułatwiają deklarowanie przetwarzania składni języka wyrażonej za pomocą metamodelu. Języki te pozwalają na formułowanie „zapytań” do modeli, na podstawie których można znajdować fragmenty odpowiadające zadanym wzorcom (grafom wzorcowym). Takie fragmenty następnie podlegają przekształceniu poprzez np. transformacje tekstowe (konkatenacje słów, zmiana wielkości znaków itp.), dodawanie lub usuwanie elementów, czy też zmiana relacji między elementami.

Rysunek 13.7 przedstawia podstawowe zasady definiowania **transformacji modeli**. Program transformujący stanowi swego rodzaju „model transformacji” (często jest on wyrażany w graficznym języku transformacji modeli). W takim modelu definiujemy wzorce modeli źródłowych oraz wzorce modeli docelowych. Dla tak zdefiniowanych wzorców określamy szczegółowe reguły transformacji. Wykonanie programu transformującego polega na poszukiwaniu w modelu źródłowym zadanych w programie wzorców. Po znalezieniu fragmentu modelu spełniającego wzorzec dokonywana jest transformacja zgodnie z zadanimi regułami. Podobne zasady obowiązują w sytuacji, kiedy chcemy z modelu wygenerować kod. Różnica polega na tym, że model docelowy wyrażany jest nie w języku graficznym, lecz w języku tekstowym. Model źródłowy przeszukiwany jest poprzez wyszukiwanie wzorców grafowych, natomiast reguły transformacji oparte są na odpowiednim generowaniu tekstu.

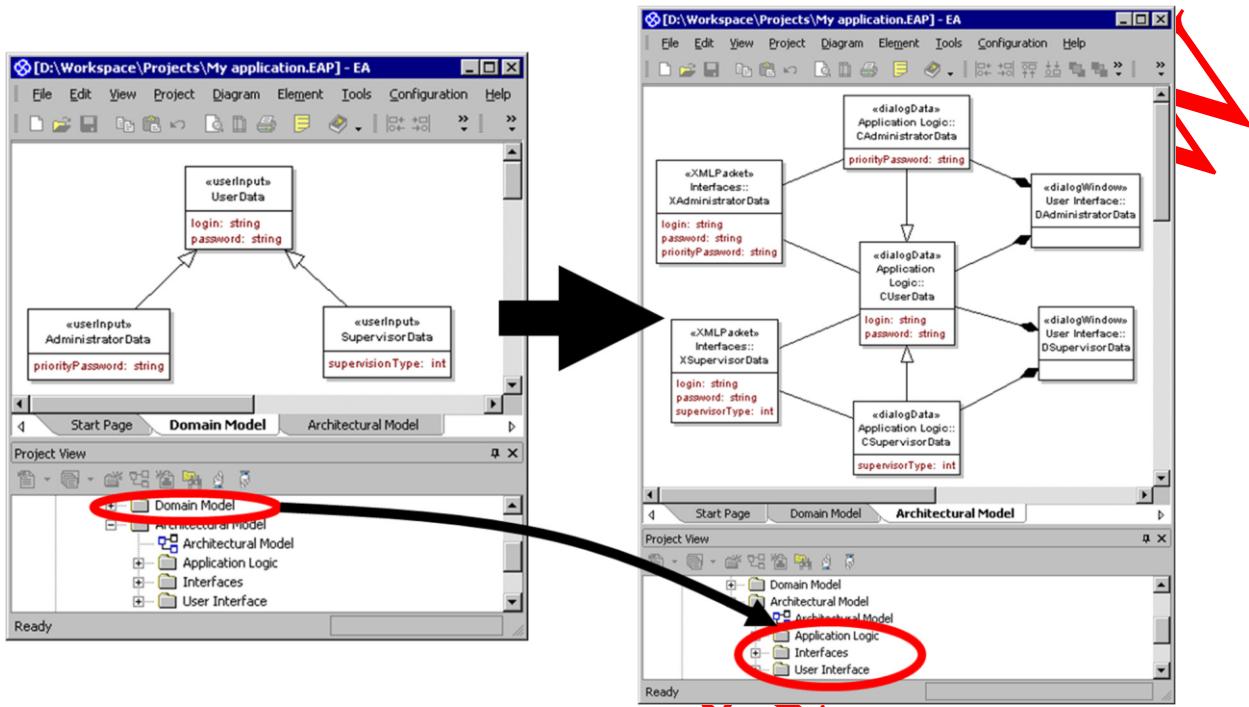


Rysunek 13.7: Zasada definiowania transformacji modeli

Szczegółowe omówienie zasad transformacji modeli jest poza zakresem niniejszych materiałów. Czytelnika odsyłamy zatem do odpowiedniej literatury z obszaru WOSM. Z punktu widzenia typowego projektu software'owego ważne jest to, że techniki transformacji modeli oraz generacji kodu pozwalają na znaczące przyspieszenie rutynowych czynności związanych z modelowaniem. W wielu przypadkach można wykorzystać gotowe programy transformujące, które pozwalają przekształcać konkretny model (np. model dziedziny w postaci diagramów klas) w inny – bardziej szczegółowy lub dotyczący innego paradymatu (np. model bazy danych w postaci diagramów ERD).

Przykład zastosowania transformacji modeli widzimy na rysunku 13.8. Widzimy tutaj fragment modelu dziedziny oraz wygenerowany na jego podstawie fragment modelu architektonicznego. Deweloper umieszcza model źródłowy w odpowiednim pakiecie w narzędziu CASE do modelowania oraz uruchamia program transformujący. W rezultacie tworzony jest nowy pakiet z wygenerowanym modelem docelowym. Taki model podlega potem dalszej modyfikacji w sposób tradycyjny. Ważne jest to, że określone podstawowe zasady translacji między jednym modelem a drugim zostały „zaszyte” w programie transformującym.

Do użytku WOZNICKIEGO PW



Rysunek 13.8: Przykład definiowania transformacji modeli

Drugą metodą w cyklu automatyzacji inżynierii oprogramowania jest metoda nazywana **DevOps** (ang. Development and Operations), czyli – Rozwój i Obsługa. O ile metoda WOSM dotyczyła dyscyplin związanych z modelowaniem, o tyle DevOps dotyczy przede wszystkim dyscyplin związanych z kodowaniem (implementacja, testowanie, wdrożenie i utrzymanie). Podstawową zasadą metody DevOps jest stworzenie spójnego zestawu narzędzi, które w jak największym stopniu zautomatyzują powtarzalne i rutynowe czynności w cyklu życia kodu. Co jest szczególnie istotne – nacisk jest położony na zintegrowanie czynności związanych z wytworzeniem oprogramowania (rozwój) z czynnościami związanymi z eksploatacją (obsługa). Zależność ta jest zilustrowana na rysunku 13.9.



Rysunek 13.9: Schemat cyklu automatyzacji metodą DevOps

Cykł DevOps rozpoczyna się od etapu **planowania**. Etap ten jest wspierany przez narzędzia do zarządzania projektami dedykowane projektom software'owym. W szczególności, narzędzia te wspierają planowanie oraz kontrolę realizacji czynności w procesie implementacji systemu. Istotna jest również integracja tych narzędzi z narzędziami do śledzenia błędów i problemów. Każde zgłoszenie problemu może być dzięki temu w łatwy sposób zamienione na odpowiednie cechy systemu (np. historie użytkownika lub przypadki użycia) w planie projektu oraz czynności w planie odpowiednich iteracji.

Następny etap to **kodowanie**. Na pierwszy plan wchodzą tutaj oczywiście narzędzia IDE wraz z centralnymi repozytoriami kodu oraz narzędziami do kontroli wersji. Jest to typowe środowisko pracy programistów. Bardzo ważna jest jednak integracja tego środowiska z innymi narzędziami. W pierwszej kolejności, integracja dotyczy narzędzi z etapu planowania. Dzięki integracji, programista wie na bieżąco, jakie czynności są konieczne do wykonania oraz bardzo szybko może zgłosić podjęcie czynności i jej wykonanie. W niektórych rozwiązaniach dzieje się to bezpośrednio podczas wykonywania typowych czynności związanych z zarządzaniem kodem i jego wersjonowaniem. Nie wymaga zatem dodatkowej pracy, a znacznie usprawnia komunikację w zespole.

Drugi obszar integracji narzędzi do kodowania jest związany z etapem **budowy**. Na tym etapie wykorzystywane są narzędzia do ciągłej integracji systemu. Integracja polega głównie na bezpośrednim uruchamianiu narzędzi integrujących z poziomu środowiska programistycznego IDE. Programista jednym przyciskiem jest w stanie skompilować i zintegrować cały system uzyskując bardzo szybko i automatycznie – gotowe do zainstalowania wydanie systemu.

Kolejny obszar integracji dotyczy etapu **testowania**. Wykorzystując odpowiednie narzędzia możemy uruchamiać testy jednostkowe, automatycznie wykonywane podczas integracji systemu. Taka integracja jest szczególnie istotna dla testów regresyjnych. Programista ma możliwość szybkiego i automatycznego zweryfikowania, czy nie pojawiły się błędy dotyczące kodu już wcześniej sprawdzonego. Podobnie, możliwa jest automatyzacja wykonania testów akceptacyjnych. Po zintegrowaniu systemu uruchamiane są wtedy odpowiednie skrypty testowe sprawdzające działanie systemu z punktu widzenia jego użytkownika końcowego.

Na etapie testowania kończy się obszar rozwoju (Dev) i przechodzimy do obszaru obsługi (Ops). Ważne jest to, że przejście to jest automatyczne i ściśle zintegrowane. Obsługa rozpoczyna się od etapów **konfiguracji** oraz **wdrożenia**. Wykorzystywane są tu narzędzia do zarządzania konfiguracją oraz do ciągłego wdrażania. Wykorzystują one produkty narzędzi do ciągłej integracji. Często wykorzystywane są tutaj środowiska do konteneryzacji. Konfiguracja polega wtedy na stworzeniu odpowiedniego opisu kontenerów, które zawierają wszystkie moduły niezbędne do ich uruchomienia (kod wykonywalny, system bazy danych, system kolejkowania itd.). Kontenery te są budowane na etapie integracji. Na etapie konfiguracji z odpowiednich wersji kontenerów tworzony pakiet instalacyjny dla całego systemu. Następnie, odpowiednie wersje kontenerów są instalowane przy pomocy narzędzi do ciągłej instalacji. Instalacja odbywa się poprzez umieszczenie kontenerów w odpowiednich środowiskach kontenerowych.

Po zainstalowaniu systemu w środowisku wykonywalnym przechodzimy do etapu **obsługi**. Obsługa polega na uruchamianiu zainstalowanego systemu oraz utrzymywaniu uruchomionego systemu w ruchu. Tak jak już wspomnieliśmy, najlepszymi środowiskami wspierającymi te działania – w kontekście metody DevOps – są środowiska kontenerowe. Środowiska te pozwalają na ciągłą kontrolę działania uruchomionych kontenerów. Jednocześnie, bardzo łatwe jest dokonywanie serwisowania systemu poprzez zatrzymywanie kontenerów oraz szybkie uruchamianie kolejnych ich wersji. W łatwy sposób można również zarządzać obciążeniem systemu poprzez automatyzację uruchamiania wielu instancji kontenerów.

Ostatni etap cyklu DevOps to **monitorowanie**. Na tym etapie zbierane są różne informacje na temat działającego systemu. Informacje te są następnie przekazywane do zespołu deweloperskiego (Dev) w celu podjęcia dalszych działań rozwojowych. Zbierane informacje mogą dotyczyć zauważonych niedogodności w działaniu systemu, niewystarczającej wydajności czy też postulowanych rozszerzeń lub zmian funkcjonalności. Informacje te zbierane są przez odpowiednie narzędzia analityczne i przekazywane do systemów planowania projektu. W ten sposób zamyka się cykl DevOps.

Na koniec warto podkreślić, że zakres narzędzi stosowanych w projekcie należy koniecznie dostosować do uwarunkowań danego projektu oraz organizacji wytwarzającej oprogramowanie. Zastosowanie pełnego cyklu DevOps może być kosztowne. Dotyczy to kosztu zakupu narzędzi, ich konfiguracji oraz utrzymania. Jednocześnie jednak, cykl DevOps pozwala zwiększyć efektywność pracy, co zmniejsza koszty wytwarzania systemu. Konieczne jest jednak dokonanie analizy korzyści i wybór tych narzędzi, które rzeczywiście usprawnią pracę w danych warunkach projektowych. Ważne kryteria selekcji narzędzi CASE to m.in. wielkość zespołów pracujących nad produktami, stopień komplikacji rozwijanych systemów, charakterystyka docelowych platform uruchomieniowych, ograniczenia środowiskowe projektu, plany dotyczące utrzymania powstałych aplikacji, a także metodyka, procedury i zasady stosowane w procesie twórczym.

Warto pamiętać, że warsztat pracy inżynierów oprogramowania nie musi być wyposażony we wszystkie możliwe narzędzia od pierwszego dnia swojego funkcjonowania – kolejne elementy mogą być dodawane stopniowo, wraz z pojawianiem się nowych potrzeb oraz rosnącym doświadczeniem zespołów managerów i programistów. Wprowadzenie zbyt wielu narzędzi zbyt wcześnie może spowodować paraliż organizacyjny, utrudnienia pracy wynikające problemów technicznych ze świeżo zaaplikowanymi narzędziami oraz poświęcenie nadmiernych zasobów na naukę narzędzi. Z drugiej strony inwestycje infrastrukturalne, finansowe oraz czas poświęcony na poznanie aplikacji CASE może przynieść sukces nie tylko w obrębie danego projektu, ale także procentować w przyszłości przez wyższą jakość produktów powstających przy mniejszym nakładzie pracy.

Zadania

Zadanie 1

Narysuj metamodel dla zadanego języka. Język powinien pozwalać na rysowanie akcji oraz przepływów sterowania lub przepływów obiektów między akcjami. Akcje mogą być trzech rodzajów (proszę założyć odpowiednie typy akcji).

Zadanie 2

Narysuj przykładowe modele zgodne z metamodelem z rozwiązania zadania 1. Zaproponuj składnię konkretną.

Zadanie 3

Poszukaj języków do transformacji modeli oraz powiązanych z nimi narzędzi. Napisz prostą transformację w wybranym języku transformacji. Transformacja powinna zamieniać klasę z atrybutami w języku UML w tabelę z kolumnami w języku ERD. Spróbuj wykonać tą transformację w wybranym narzędziu.

Zadanie 4

Poszukaj narzędzi dla poszczególnych etapów metody DevOps. Opisz, w jaki sposób narzędzia te można ze sobą zintegrować.

Słownik pojęć

DevOps

Metoda automatyzacji inżynierii oprogramowania łącząca ze sobą fazy rozwoju (Dev) i obsługi (Ops). Podstawową zasadą metody DevOps jest stworzenie spójnego zestawu narzędzi, które w jak największym stopniu zautomatyzują powtarzalne i rutynowe czynności w całym cyklu życia kodu.

Metamodel

Model służący do zdefiniowania składni modeli w określonym języku modelowania („model modelu”). Metamodel najczęściej jest wariantem modelu klas języka UML. Każda meta-klasa w metamodelu wyraża jeden element składni języka modelowania.

Narzędzia komputerowego wsparcia inżynierii oprogramowania (CASE)

Narzędzia pozwalające usprawnić czynności związane z wieloma aspektami rozwoju oprogramowania: od zarządzania projektami informatycznymi, przez integrację powstających modułów i zarządzanie zmianami, po wsparcie dla modelowania systemów na różnych poziomach i ułatwianie implementacji.

Szybkie prototypowanie aplikacji

Podejście wykorzystujące kombinację generatorów kodu i edytorów GUI (interfejsu użytkownika), pozwalające na wcześnie utworzenie aplikacji uruchamialnych i posiadających zbliżony do docelowego interfejs użytkownika.

Zintegrowane środowisko programistyczne (IDE)

Klasa narzędzi CASE wspierających tworzenie kodu, posiadających wiele funkcji ułatwiających życie programistom. Podstawowym zadaniem takich narzędzi jest zwiększenie produktywności przez zintegrowanie wszystkich funkcjonalności używanych w typowym cyklu życia kodu w jedną spójną całość.

Co trzeba zapamiętać

Stosowanie narzędzi

Aby praca informatyków była nie tylko wydajna i zakończona sukcesem, ale w ogóle możliwa, niezbędne jest powstanie odpowiedniego środowiska, w którym te tworzone artefakty są właściwie uporządkowane. Takie środowisko pracy powinno pozwalać nie tylko na łatwą wymianę informacji między uczestnikami projektu, ale również na automatyzację możliwie wielu czynności na wszystkich etapach procesu twórczego. W celu usprawnienia pracy zespołów konstruujących systemy oprogramowania powstały różnego rodzaju narzędzia, które nazywamy narzędziami komputerowego wsparcia inżynierii oprogramowania.

Narzędzia analizy i projektowania

Bardzo istotną rolę w warsztacie twórców oprogramowania spełniają narzędzia modelowania obiektowego. Pozwalają one m.in. na kontrolę poprawności notacji modeli, generowanie szkieletowego kodu na podstawie modeli czy też generowanie dokumentacji modeli. Projektowanie baz danych ułatwiają narzędzia do zarządzania bazami danych. Samo modelowanie struktury bazy danych odbywać się może w sposób podobny do modelowania obiektowego – przy wykorzystaniu notacji ERD lub UML. Narzędzia

CASE dedykowane bazom danych dostarczają wielu funkcji ułatwiających budowę systemów bazodanowych. Często używaną grupą narzędzi są narzędzia do modelowania interfejsu użytkownika. Narzędzia takie pozwalają na tworzenie szkiców i makiet interfejsu użytkownika

Narzędzia implementacji i testowania

Praktycznie we wszystkich projektach używane są zintegrowane środowiska programistyczne (IDE). Podstawowymi elementami takich środowisk są inteligentny edytor kodu zintegrowany z kompilatorem oraz odpluskwiacz. Wiele nowoczesnych narzędzi klasy IDE posiada wsparcie dla prototypowania graficznych interfejsów użytkownika. Narzędzia IDE współpracują z narzędziami ciągłej integracji i wdrażania. Centrum takich systemów są mechanizmy automatycznego tworzenia i instalowania w środowisku wykonawczym, działających wersji rozwijanego produktu. Warsztat implementatora uzupełniają narzędzia do śledzenia błędów i problemów. Osobną grupę narzędzi stanowią systemy wspomagające testowanie. Wiele takich systemów pozwala na wspomaganie testów jednostkowych w ramach środowisk IDE. Inną grupę stanowią systemy wspomagające wykonywanie testów akceptacyjnych poprzez automatyzację nawigacji przez interfejs użytkownika.

Automatyzacja wytwarzania i eksploatacji

Kompletny cykl automatyzacji wytwarzania i eksploatacji oprogramowania obejmuje dwie podstawowe koncepcje: wytwarzanie oprogramowania sterowane modelami (WOSM) oraz tzw. DevOps. Podstawą WOSM jest automatyzacja procesu przekształcania modeli (analitycznych, projektowych) w inne, bardziej szczegółowe modele, oraz kod. Główną zasadą jest ujednolicenie definiowania składni języków za pomocą metamodeli oraz automatyzacja przetwarzania modeli jako grafów. Podejście DevOps koncentruje się przede wszystkim na dyscyplinach związanych z kodowaniem (implementacja, testowanie, wdrożenie i utrzymanie). Podstawową zasadą metody DevOps jest stworzenie spójnego zestawu narzędzi, które w jak największym stopniu zautomatyzują powtarzalne i rutynowe czynności w cyklu życia kodu. Co jest szczególnie istotne – nacisk jest położony na zintegrowanie czynności związanych z wytwarzaniem oprogramowania (rozwój) z czynnościami związanymi z eksploatacją (obsługa).