

Michał Śmiałek, Wiktor Nowakowski, Tomasz Straszak
Jacek Bojarski, Albert Ambroziewicz

Inżynieria oprogramowania

Politechnika Warszawska

Warszawa, 2010

Spis treści

1.	Wprowadzenie do inżynierii oprogramowania.....	7
1.1.	Czym zajmuje się inżynieria oprogramowania.....	7
1.2.	Złożoność systemów oprogramowania	9
1.3.	Problemy inżynierii oprogramowania.....	9
1.4.	Najlepsze praktyki inżynierii oprogramowania	11
1.5.	Podsumowanie	13
2.	Cykle wytwarzania oprogramowania	14
2.1.	Analiza i synteza, jako podstawa procesu wytwarzania oprogramowania	14
2.2.	Dyscypliny cyklu wytwarzania oprogramowania.....	15
2.2.1.	Dyscyplina analizy.....	16
2.2.2.	Dyscyplina projektowania.....	17
2.2.3.	Dyscyplina implementacji.....	17
2.2.4.	Dyscyplina testowania.....	18
2.2.5.	Dyscyplina wdrożenia	18
2.2.6.	Dyscyplina konserwacji	19
2.3.	Cykle twórcze w inżynierii oprogramowania.....	19
2.3.1.	Cykł wodospadowy	20
2.3.2.	Cykł iteracyjny	21
2.3.3.	Cykł spiralny.....	22
2.4.	Podsumowanie	23
3.	Metodyki wytwarzania oprogramowania	24
3.1.	Co to jest metodyka wytwarzania oprogramowania.....	24
3.1.1.	Notacja.....	25
3.1.2.	Techniki	27
3.1.3.	Proces twórczy	28
3.2.	Rodzaje metodyk wytwarzania oprogramowania.....	29
3.2.1.	Metodyki formalne.....	29
3.2.2.	Metodyki agilne (zwinne).....	30
3.3.	Wytwarzanie oprogramowania sterowane modelami (MDD)	31
3.4.	Podsumowanie	33
4.	Wprowadzenie do obiektowego modelowania oprogramowania	35
4.1.	Czym jest modelowanie?	35
4.2.	Zasady modelowania złożonych systemów.....	37
4.3.	Obiekty jako podstawa modelowania	38
4.4.	Klasy obiektów.....	40
4.5.	Modelowanie systemu jako układu współpracujących ze sobą obiektów.....	44

4.6.	Modele na ścieżce od opisu środowiska do kodu.....	46
4.7.	Podsumowanie	47
5.	Modelowanie struktury systemu w języku UML.....	48
5.1.	Przegląd diagramów opisu struktury w języku UML	48
5.1.1.	Diagram klas	49
5.1.2.	Diagram obiektów	49
5.1.3.	Diagram pakietów	50
5.1.4.	Diagram komponentów	50
5.1.5.	Diagram składowych	51
5.1.6.	Diagram wdrożenia	52
5.2.	Modelowanie struktury logicznej systemu	52
5.2.1.	Model klas.....	52
5.2.2.	Model komponentów.....	59
5.3.	Modelowanie struktury fizycznej.....	62
5.3.1.	Model wdrożenia.....	62
5.4.	Podsumowanie	64
6.	Modelowanie dynamiki systemu w języku UML.....	66
6.1.	Przegląd diagramów opisu dynamiki w języku UML.....	66
6.1.1.	Diagram przypadków użycia	67
6.1.2.	Diagram czynności.....	67
6.1.3.	Diagram sekwencji.....	67
6.1.4.	Diagram komunikacji.....	67
6.1.5.	Diagram opisu interakcji.....	68
6.1.6.	Diagram następstwa	69
6.1.7.	Diagram maszyny stanów	70
6.2.	Model przypadków użycia	71
6.2.1.	Aktorzy i przypadki użycia	71
6.2.2.	Relacje między przypadkami użycia.....	73
6.2.3.	Scenariusze przypadków użycia	74
6.3.	Model czynności	75
6.3.1.	Węzły czynności i przepływ sterowania.....	76
6.3.2.	Decyzje i scalenia	77
6.3.3.	Rozwidlenia i złączenia	78
6.3.4.	Tory	79
6.4.	Modele interakcji.....	80
6.4.1.	Diagram sekwencji.....	81
6.4.2.	Fragmenty w diagramach sekwencji	82
6.5.	Podsumowanie	83
7.	Podstawy inżynierii wymagań	84

7.1.	Co to są wymagania i na czym polega inżynieria wymagań?	84
7.2.	Podział wymagań.....	85
7.2.1.	Wymagania zamawiającego a wymagania oprogramowania.....	86
7.2.2.	Wymagania funkcjonalne a wymagania pozafunkcjonalne	88
7.3.	Specyfikowanie widocznego sposobu zachowania się systemu.....	88
7.4.	Specyfikowanie słownika danych przetwarzanych przez system.....	92
7.5.	Cechy dobrej specyfikacji wymagań.....	94
7.6.	Zachowanie spójności i jednoznaczności specyfikacji wymagań.....	97
7.7.	Podsumowanie	99
8.	Podstawy projektowania	100
8.1.	Projektowanie na różnych poziomach złożoności.....	100
8.1.1.	Projektowanie architektoniczne.....	101
8.1.2.	Projektowanie szczegółowe.....	103
8.2.	Projektowanie struktury systemu.....	104
8.2.1.	Projektowanie struktury modelu architektonicznego	104
8.2.2.	Projektowanie struktury modelu szczegółowego	106
8.3.	Projektowanie dynamiki systemu	111
8.3.1.	Projektowanie dynamiki na poziomie architektury	111
8.3.2.	Projektowanie dynamiki na poziomie projektu szczegółowego	113
8.4.	Warstwowy model architektury oprogramowania.	114
8.5.	Podstawy wzorców projektowych.....	117
8.6.	Zachowanie spójności specyfikacji projektowej i zgodności z wymaganiami. 122	
8.7.	Podsumowanie	124
9.	Metamodelowanie MOF i transformacje	125
9.1.	Metamodelowanie	125
9.1.1.	Czterowarstwowa hierarchia metamodeli	125
9.1.2.	Składnia i semantyka języka	126
9.1.3.	Język MOF	127
9.2.	Fragment metamodelu na przykładzie UMLa	127
9.2.1.	Metamodel przypadków użycia w języku UML	127
9.3.	Podstawy transformacji modeli.....	128
9.4.	Języki specyfikacji transformacji	129
9.4.1.	Przegląd specyfikacji języków transformacji.....	129
9.4.2.	Podstawy języka MOLA	131
9.5.	Transformacja między modelami na różnych poziomach abstrakcji.	135
9.6.	Podsumowanie	138
10.	Implementacja systemu, zarządzanie konfiguracją i zmianami	139
10.1.	Kodowanie systemu na podstawie projektu.....	139
10.2.	Dobre praktyki w zakresie kodowania	142

10.3.	Kontrola wersji	146
10.4.	Zarządzenie zmianami	149
10.5.	Zarządzanie konfiguracją	151
10.6.	„Inżynieria w obie strony” (ang. round-trip engineering) – synchronizacja kodu z projektem	152
10.7.	Podsumowanie	154
11.	Podstawy testowania	155
11.1.	Ustalenie ilości zadań testowych.....	155
11.2.	Różne poziomy testowania systemu: od testów modułowych do testów akceptacyjnych	157
11.3.	Podstawowe metody testowania	158
11.4.	Budowa scenariuszy testowych na podstawie przypadków użycia systemu..	161
11.5.	Testowanie cech pozafunkcjonalnych systemu	163
11.6.	Organizacja procesu testowania systemu	164
12.	Narzędzia Inżynierii oprogramowania	166
12.1.	Omówienie narzędzi wspomagających modelowanie obiektowe	166
12.2.	Narzędzia do zarządzania bazami danych	168
12.3.	Narzędzia wspierające programowanie	169
12.4.	Narzędzia ciągłej integracji	171
12.5.	Zastosowanie narzędzi do zarządzania konfiguracją i zmianami	172
12.6.	Narzędzia wspierające proces testowania	174
12.7.	Podsumowanie	175
13.	Inżynieria procesu wytwórczego oprogramowania	176
13.1.	Inżynierskie aspekty zarządzania konstrukcją oprogramowania	176
13.2.	Wdrażanie metod wytwórczych oprogramowania do praktyki organizacji wytwarzających oprogramowanie	176
13.3.	Organizacja zespołów wytwarzających oprogramowanie	178
13.4.	Szacowanie złożoności oprogramowania	183
13.4.1.	Przegląd metod szacowania oprogramowania	184
13.4.2.	Metoda Punktów Przypadków Użycia (<i>Use Case Points</i>)	184
13.5.	Ponowne wykorzystanie oprogramowania	187
13.5.1.	Modułowe podejście do budowy oprogramowania	187
13.5.2.	Wzorce oprogramowania	188
13.5.3.	Przechowywanie zasobów oprogramowania	189
13.5.4.	Ponowne wykorzystanie oprogramowania sterowane wymaganiami... 190	190
13.5.5.	Proces wytwórczy wzbogacony o ponowne wykorzystanie oprogramowania 191	191
13.6.	Podsumowanie	192

Do użytku wewnętrznego PW

1. Wprowadzenie do inżynierii oprogramowania

1.1. 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 tej dziedziny informatyki.

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, weryfikacji 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, spróbujmy porównać 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 zauważyc, 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. 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 takiej liczby elementów nie jest w stanie opanować nawet najbardziej uzdolniony programista. Konieczne jest zatem zastosowanie metod opanowania tej złożoności.

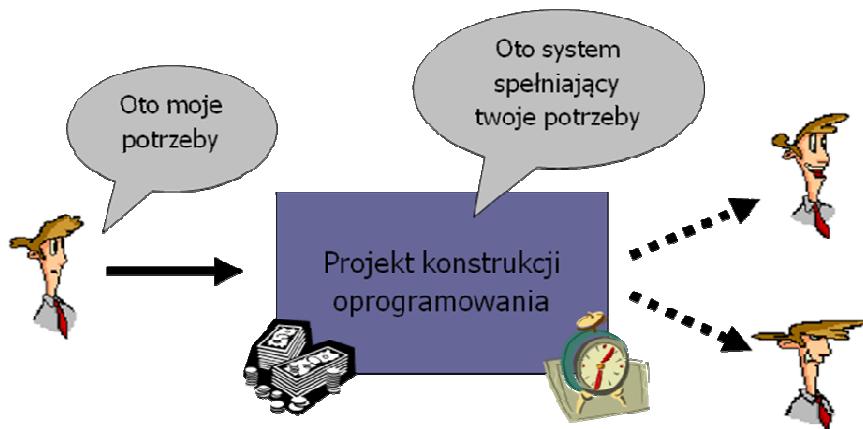
Ta złożoność wynika z zakresu funkcjonalności, jakie współczesne systemy oprogramowania muszą zapewniać ich użytkownikom. Dla przykładu, typowy, zaawansowany procesor tekstu dostarcza dziesiątek opcji menu, wielu operacji sterowanych klawiszami funkcyjnymi, wielu formularzy i okienek do wprowadzania 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, 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.

1.3. Problemy inżynierii oprogramowania

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 (patrz Rysunek 1.1).

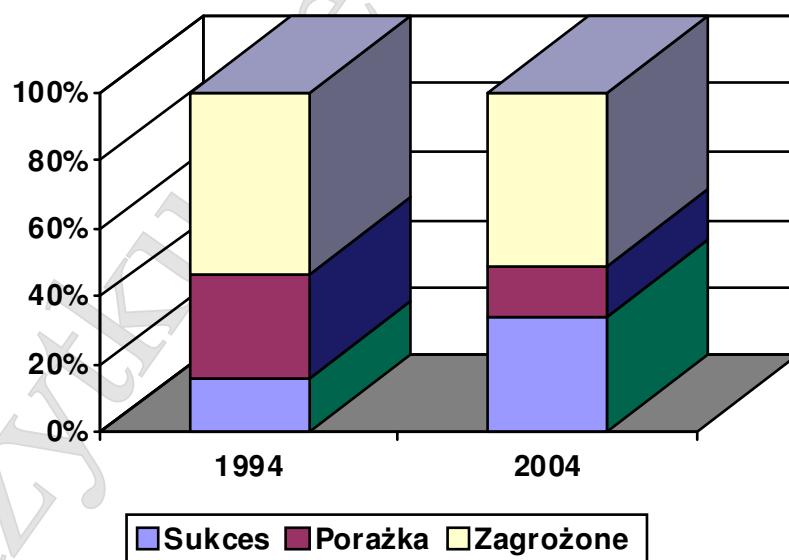


Rysunek 1.1: Kryteria sukcesu projektu konstrukcji oprogramowania

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 dwa wyniki dla lat 1994 i 2004. Statystyki te pokazują bardzo niekorzystną sytuację w przemyśle wytwarzania oprogramowania. Oznaczają one, że znaczący odsetek projektów (66% w roku 2004) 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:

- 34% - 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 naprawowaliśmy i co?”
3. Kłopotnie o zakres
 - a. „chcacie, żebyśmy zbudowali system dwa razy większy niż zapisano w umowie”
 - b. „ale ta funkcjonalność miała być dostarczona dopiero w następnej wersji”
 - c. „nie dostarczyliśmy 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. „myśliliśmy, 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” (do kopania rowów?)
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?)

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”.

1.4. Najlepsze praktyki inżynierii oprogramowania

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 bude 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 systemu. Przyczyna ta wiąże się z projektowaniem systemu. Zarządzanie złożonością polega na umiejętności podziału 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 okupywane 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 pakiecie, 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. Nie używanie 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 wytwórczego jest: „wszystko, czego nam trzeba, to dobry kompilator”

Statystyki raportów chaosu pokazują, że coraz więcej zespołów dostrzega symptomy choroby i dotarło do ich przyczyn. Coraz więcej z nich (choćż zdecydowanie zbyt mało) zaczyna stosować metody inżynierskie w produkcji oprogramowania. Jakie są zatem te metody? Jakie praktyki należy stosować, aby nie doświadczać symptomów choroby?

Potrzebujemy metodyki: notacji, technik i procesu wytwórczego.

Potrzebujemy zastosowania najlepszych praktyk, które sprawdziły się w wielu projektach zakończonych sukcesem. W szczególności, musimy zacząć stosować w sposób systematyczny odpo-

wiedzie metodyki inżynierii oprogramowania. Składniki takich metodyk możemy podzielić na trzy części: notacja, techniki, proces techniczny. Notacja dostarcza jednolitego języka dla porozumiewania się członków zespołu. Techniki przedstawiają sposoby działania, które pokonują zasadnicze przyczyny niepowodzeń. Proces techniczny organizuje te sposoby działania w spójny ciąg czynności prowadzący do celu, jakim jest zbudowanie systemu na czas, w budżecie i spełniającego rzeczywiste potrzeby jego odbiorców. Spróbujmy, zatem, krótko przedstawić te praktyki oraz odnieść się do kolejnych rozdziałów, gdzie elementy tych praktyk są przedstawione bardziej szczegółowo.

1. Proces techniczny

- Praktyka 1: Organizuj czynności cyklu życia systemu – w przeciwnym razie uyskasz chaos organizacyjny. O sposobie organizacji cyklu wytwarzania oprogramowania mówi rozdział 2.

2. Notacja

- Praktyka 2: Dokumentuj wszystkie swoje decyzje – w przeciwnym razie szybko zapomnisz dlaczego zrobiłeś to, co zrobiłeś. Uwaga: dokumentuj najlepiej w trakcie prac, a nie po. Dokumentacja powinna zawierać wiele elementów graficznych, które stanowią „plany” systemu. W rozdziałach 4-6 został zaprezentowany sposób dokumentowania systemów przy pomocy metod obiektowych i jednolitego języka modelowania obiektowego.

3. Techniki

- Praktyka 3: Zarządzaj wymaganiami – w przeciwnym razie zapomnisz o celu projektu.
- Praktyka 4: Zarządzaj jakością – w przeciwnym razie będziesz miał niezadowolonego (często: wściekłego) klienta
- Praktyka 5: Zarządzaj złożonością systemu – w przeciwnym razie wytworzysz coś na kształt talerza ze spaghetti (jak to później rozplatać?).

Dodatkowo, w rozdziale 3 zostały omówione metodyki i ich podstawowe trzy składniki.

1.5. Podsumowanie

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.

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 wytwórczym na realizację wspólnego celu. W przypadku projektu informatycznego wspólnym celem jest wytworzenie 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 jednakże przedstawić jako powtarzalny cykl rozwiązywania poszczególnych problemów, prac technicznych i łączenia rozwiązań. Na podstawie doświadczeń zebranych na przestrzeni lat opracowano wiele modeli procesów wytwórczych oprogramowania, które przedstawiają różne cykle wytwarzania oprogramowania.

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

2.1. Analiza i synteza, jako podstawa procesu wytwarzania oprogramowania

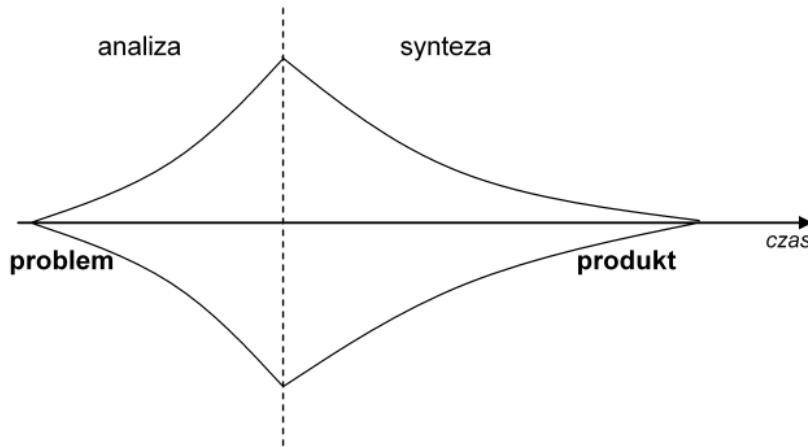
Celem każdego projektu jest wytworzenie pewnego produktu (lub produktów). Głównym produktem wytwórczego przedsięwzięcia informatycznego jest system oprogramowania, który jest uzupełniany innymi produktami, takimi jak: instrukcja użytkownika, specyfikacja wymagań, modele projektowe, itp. Produkty uzupełniające (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 wytwórczym.

Dyscypliny procesu wytwarzania oprogramowania można podzielić dwie grupy: analiza i synteza. 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.



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

Podążając za naturalną dla człowieka ścieżką analizy i syntezy (Rys 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. Rys. 2.2 obrazuje wzajemną zależność czynności analitycznych i syntetycznych. Produkty dyscyplin analitycznych, które prowadzą od ogólnego do szczegółu, są podstawą do rozpoczęcia prac syntetycznych, które poprzez realizację i scalanie mniejszych części prowadzą do wytworzenia spójnego produktu.

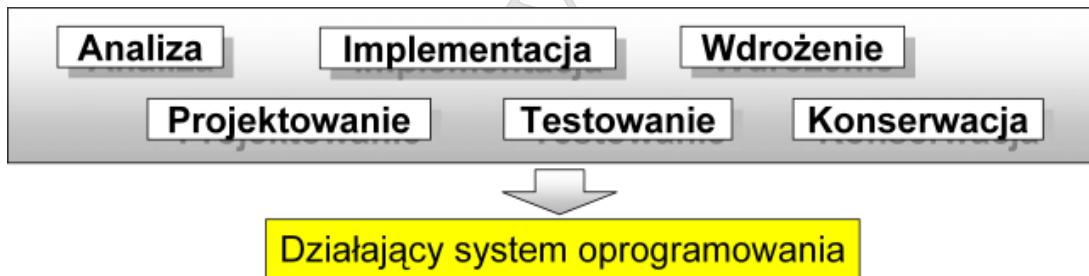


Rys. 2.2: Analiza i syntezę procesie wytwarzania oprogramowania

2.2. Dyscypliny cyklu wytwarzania oprogramowania

Ogólna definicja procesu wytwarzania oprogramowania, wyznaczając jedynie ścieżkę od problemu do produktu, daje dużą swobodę przy jej praktycznym wykorzystaniu. Jest to niezaprzeczalnie jej zaletą, gdyż duża różnorodność projektów informatycznych wymaga dostosowywania kształtu procesu do indywidualnych potrzeb. W sekcji 2.3 zostaną przedstawione główne typy cykli wytwarzających w inżynierii oprogramowania, które mogą być stosowane w różnych procesach wytwarzania oprogramowania.

Niezależnie od wybranego typu cyklu, zgodnie z definicją, każdy proces wytwarzania oprogramowania obejmuje kilka dyscyplin, z których każda dostarcza przynajmniej jeden produkt (pośredni). Produkty pośrednie są punktami startowymi dla czynności objętych pozostałymi dyscyplinami. Na Rys. 2.3 przedstawiono główne dyscypliny procesu wytwarzania oprogramowania w kolejności zgodnej ze ścieżką „analiza – syntezę” [Somm03].



Rys. 2.3: Dyscypliny cyklu wytwarzania oprogramowania

Dyscyplina analizy problemu dostarcza informacji o kształcie budowanego systemu. 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.

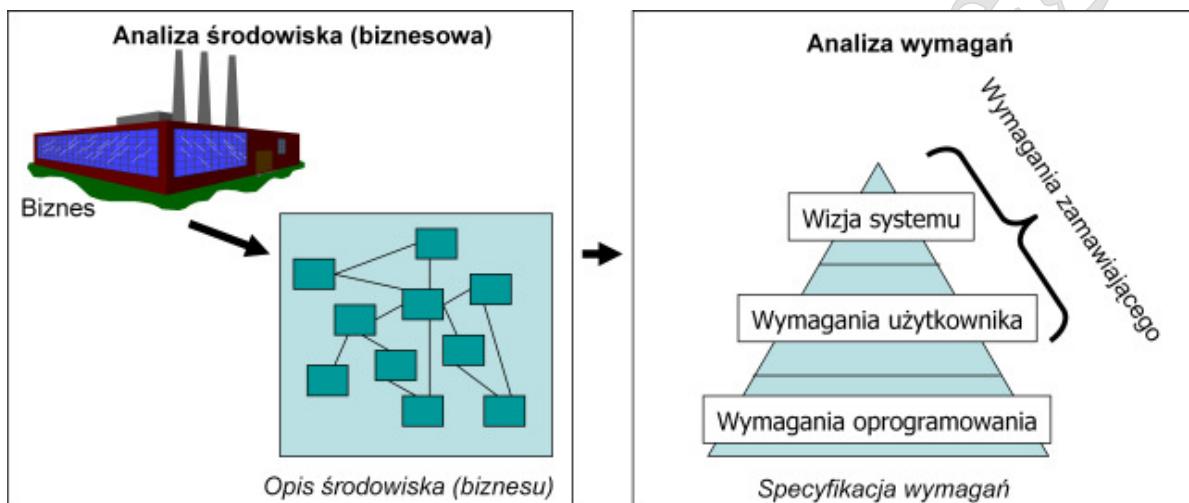
Główne produkty dyscypliny analizy to opis biznesu i specyfikacja wymagań. Na podstawie specyfikacji wymagań, dyscyplina projektowania systemu tworzy zbiór modeli projektowych (architektura, projekt bazy danych, projekty szczegółowe komponentów, wdrożenie). Dyscyplina implementacji realizuje założenia projektowe. Powstaje działający system, który jest gotowy do testowania. Pomyślne przejście testów umożliwia wdrożenie systemu w środowisku produkcyjnym, a instrukcje użytkowania systemu tworzone we wszystkich innych dyscyplinach służą do szkolenia użytkowników. Ostatecznym produktem jest działający system informatyczny, który powinien podlegać ciągłemu utrzymywaniu.

W następnych podsekcjach zostaną omówione najważniejsze cechy poszczególnych dyscyplin. Szczegóły dotyczące każdej z nich znajdują się w pozostałych rozdziałach tego podręcznika.

2.2.1. Dyscyplina analizy

Po określeniu problemu i podjęciu decyzji o rozpoczęciu budowy systemu oprogramowania, który będzie realizował zadany problem, rozpoczyna się analiza. Głównym celem tej dyscypliny jest określenie zakresu i kształtu przyszłego systemu. Rys. 2.4 przedstawia ogólny przebieg i strukturę dyscypliny analizy.

Przy współpracy przedstawicieli zamawiającego i wykonawcy opisany zostaje aktualny stan biznesu. Na tej podstawie definiuje się główne potrzeby użytkownika, 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, opisujący szczegóły działania systemu (m.in. opis interakcji systemu z użytkownikiem, wygląd okien). Głównymi produktami wytwarzanymi przez tą dyscyplinę są opis biznesu i specyfikacja wymagań.



Rys. 2.4: Dyscyplina analizy

Opis biznesu jest 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 setki lub tysiące różnych elementów oddziałujących na siebie w określony, złożony sposób. Podstawowe elementy organizacji biznesowej to współpracownicy, pracownicy, jednostki organizacyjne, produkty, surowce, podzespoły, dokumenty, systemy informatyczne. Budowany system informatyczny będzie jedynie elementem składowym biznesu. Modelowanie może przebiegać na różnym poziomie abstrakcji: możemy modelować całą firmę lub tylko jakiś jej fragment (np. dział czy placówkę). Opis biznesu zawiera zwykle słownik pojęć biznesowych reprezentujący strukturę biznesu i procesy biznesowe wyrażone za pomocą pojęć ze słownika biznesowego. 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 ograniczeń środowiskowych, które są podstawą do sformułowania tzw. wymagań pozafunkcjonalnych na budowany system.

Kompletna specyfikacja wymagań wynika bezpośrednio z opisu biznesu i zwykle przyjmuje kształt piramidy. Na samym szczycie znajduje się wizja systemu, która dostarcza informacje 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 cech, które powinny być podstawą do zawarcia kontraktu. Wizja systemu i wymagania użytkownika tworzą wymagania zamawiającego, które są odzwierciedleniem rzeczywistych potrzeb zdefiniowanych przez biznes. Natomiast wymagania oprogramowania opisują szczegóły funkcjonalne komunikacji między użytkownikami i systemem, wszystkie wymieniane między nimi dane oraz planowany wygląd systemu. Specyfikacja wymagań zawiera, obok ogólnych cech systemu, opis jego dynamiki oraz struktury. Słownik biznesu jest podstawą do stworzenia słownika dziedziny. Słownik

dziedziny jest rozszerzany w trakcie opisu cech systemu, wymagań funkcjonalnych i pozafunkcjonalnych. Słownik przyjmuje ostateczny kształt na koniec tworzenia specyfikacji wymagań. Na podstawie spójnej i kompletnej specyfikacji wymagań można zbudować system informatyczny, będący naturalnym fragmentem rzeczywistości przedstawionej w opisie biznesu i realizujący rzeczywiste potrzeby zamawiającego.

2.2.2. Dyscyplina projektowania

Dyscyplina projektowania jest jedną z dyscyplin, gdzie dokonuje się syntezy opisu problemu. Jako produkt wejściowy przyjmuje się specyfikację wymagań i opis środowiska biznesu. Celem tej działań tej dyscypliny jest stworzenie modeli projektowych na różnych poziomach abstrakcji, według których zostanie zaimplementowany system.

Zwykle, projektowanie zaczyna się od najbardziej abstrakcyjnych modeli i w zależności od złożoności systemu przechodzi się do bardziej szczegółowych poziomów.

Mając do dyspozycji opis środowiska biznesu i specyfikację wymagań można zdefiniować, jakich elementów fizycznych (np. serwer bazy danych czy serwer aplikacyjny) będzie potrzebował budowany system i w jaki sposób te elementy będą się ze sobą porozumiewać. Jest to najbardziej ogólny, fizyczny model projektowy, którego uszczegółowieniem może być model architektoniczny. Na poziomie projektowania architektury logicznej definiuje się moduły, z jakich będzie składał się budowany system i poprzez jakie interfejsy te moduły będą się komunikować. W zależności od przyjętej technologii, może to być model projektowy reprezentujący np. komponenty lub usługi (*ang. service*). Interfejsy, które są specyfikacją funkcji udostępnianych przez element architektury, pozwalają traktować moduły jako czarne skrzynki – nie jest ważne co jest w środku, ważne do czego można to wykorzystać. Istotnym elementem modelu architektonicznego i zależnych od niego modeli bardziej szczegółowych są struktury danych, które służą do komunikacji poprzez interfejsy. Ich podstawą może być słownik dziedziny, zdefiniowany w specyfikacji wymagań. Natomiast moduły mogą być odwzorowaniem grup funkcjonalności. Kolejnym, mniej abstrakcyjnym poziomem projektowania, jest projektowanie szczegółowe modułów architektury. Wchodząc w szczegóły „czarnych skrzynek”, dokładnie definiuje się ich strukturę oraz opracowuje sekwencje wykonywanych operacji, których celem jest realizacja funkcji określonych w poszczególnych interfejsach. W zależności od złożoności systemu, projektowania mogą też wymagać algorytmy, które będą zaimplementowane w systemie (np. algorytmy szyfrujące, obliczające składki bezpieczeństwa).

Produktem projektowania jest zbiór spójnych modeli projektowych, które dostarczą szczegółowych informacji o składnikach systemu, który ma spełniać wymagania opisane w specyfikacji wymagań i funkcjonować w opisanym środowisku.

2.2.3. Dyscyplina implementacji

W wyniku działań objętych dyscypliną implementacji zostaje wytworzony kod realizujący wymagania zdefiniowane podczas analizy i spełniający założenia projektowe, określone w modelach projektowych. Do wytworzenia 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 zasymulować. 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 pozafunkcjonalne i ograniczenia środowiska biznesowego.

Implementacja systemu zwykle podzielona jest na wiele części (ich liczba zależy od stopnia skomplikowania systemu). Każda część realizowana niezależnie powinna poprawnie współpracować z pozostałymi. Jako przykład może posłużyć architektura komponentowa, gdzie zadaniem implementacji komponentu i interfejsów jest wypełnienie założeń projektowych. Każdy zespół realizujący

pewną część systemu odpowiedzialny jest za jego prawidłowe funkcjonowanie. Dlatego nawet najmniejsze „kawałki” kodu powinny być testowane podczas całego procesu implementacji aż do zaimplementowania kompletnego składnika. Wtedy testowaniu podlega cały składnik (np. komponent i jego operacje oferowane przez interfejsy). Wymaga to od środowiska implementacyjnego możliwości komplikacji i uruchamiania systemu w trakcie jego tworzenia. Środowisko tworzą również narzędzia wspomagające pracę programistów. Narzędzia te wspomagają pracę grupową, kontrolę wersji kodu, kodowanie, testowanie, komplikację, jak również kontrolę nad pracą programistów.

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, obszerna dokumentacja i gwarancja poprawności działania potwierdzona dokumentacją testów jednostkowych.

Po implementacji sprawnie działających części systemu i ich połączeniu, całości powinna przejść testy integracji. Poprawność działania zostaje sprawdzona globalnie, dla całego systemu. Tak sprawdzony system jest głównym produktem dyscypliny implementacji. Pozostałe produkty to dokumentacja testów (jednostkowych i integracji), uzupełniony podręcznik użytkownika i podręcznik instalacji systemu.

2.2.4. 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, gdzie zbudowany system funkcjonuje w sporeparowanym na potrzeby testów środowisku produkcyjnym. W odróżnieniu od testów przeprowadzonych w czasie implementacji, głównymi testującymi są przedstawiciele zamawiającego. Zwykle są to przyszli użytkownicy systemu, wytypowani i specjalnie przygotowani do 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 oczekiwaniemi do działania systemu (funkcjonalność, ergonomia).

Przeprowadzenie testów akceptacyjnych systemu (ang. *user acceptance test*) wymaga planu testów. Zazwyczaj plan testów jest dokumentem powstającym na podstawie specyfikacji wymagań, gdzie ułożone w odpowiedniej kolejności wykonania funkcjonalności systemu dają spodziewane wyniki, zgodnie z regułami biznesowymi. Testerzy mając do dyspozycji instrukcje użytkowania używają system realizując scenariusze testowe, potwierdzają, bądź nie potwierdzają poprawność działania systemu. W przypadku niepomyślnego wyniku testów, może nastąpić powrót do jednej z dyscyplin twórczych (projektowanie lub implementacja) i ponowienie testów po zaimplementowaniu poprawek.

Wynikiem testowania jest dokumentacja testów odbiorczych, która jest podstawą do przejścia do czynności objętych dyscypliną wdrożenia.

2.2.5. Dyscyplina wdrożenia

Po pomyślnym przejściu testów i zatwierdzeniu włączenia zbudowanego systemu oprogramowania do biznesu, rozpoczynają się czynności dyscypliny wdrożenia. Dyscyplina ta obejmuje czynności związane z instalacją systemu i szkoleniem przyszłych użytkowników.

Można wyróżnić kilka najbardziej popularnych strategii wdrożenia nowego systemu do środowiska produkcyjnego [Szej02]:

- 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.

Instalacja Systemu obejmuje zarówno przygotowanie środowiska produkcyjnego do wdrożenia nowego systemu, jak i samą fizyczną instalację systemu w środowisku docelowym. Forma szkolenia przyszłych użytkowników jest uzależniona od wybranej strategii wdrożenia. Niezmennym celem szkolenia pozostaje jednak nauka zasad działania i operowania systemem.

Wdrożenie powinno dostarczyć ocenę zbudowanego i wdrożonego systemu.

2.2.6. Dyscyplina konserwacji

Czynności konserwacji, zwana także czynnościami utrzymania, są ostatnimi czynnościami podejmowanymi 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, funkcjonalność lub cechy systemu oprogramowania, które zostały zdefiniowane podczas analizy, mogą ulec zmianie, może zmienić się także samo środowisko i procesy biznesowe (np. zmiana innego systemu, zmiana prawa). Wszystko te sytuacje wymuszają podjęcie pewnych działań, które można podzielić na cztery grupy [Szej02]:

- 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 analizy 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.

2.3. Cykle wytwórcze w inżynierii oprogramowania

Wytworzenie oprogramowania wymaga podjęcia wielu, skoordynowanych ze sobą czynności, które prowadzą od postawienia problemu do dostarczenia produktu. W inżynierii oprogramowania ciąg takich czynności nazywany jest procesem wytworzenia oprogramowania. Abstrakcyjną reprezentacją takiego procesu wytwarzania oprogramowania jest model cyklu wytwórczego. Pozwala on

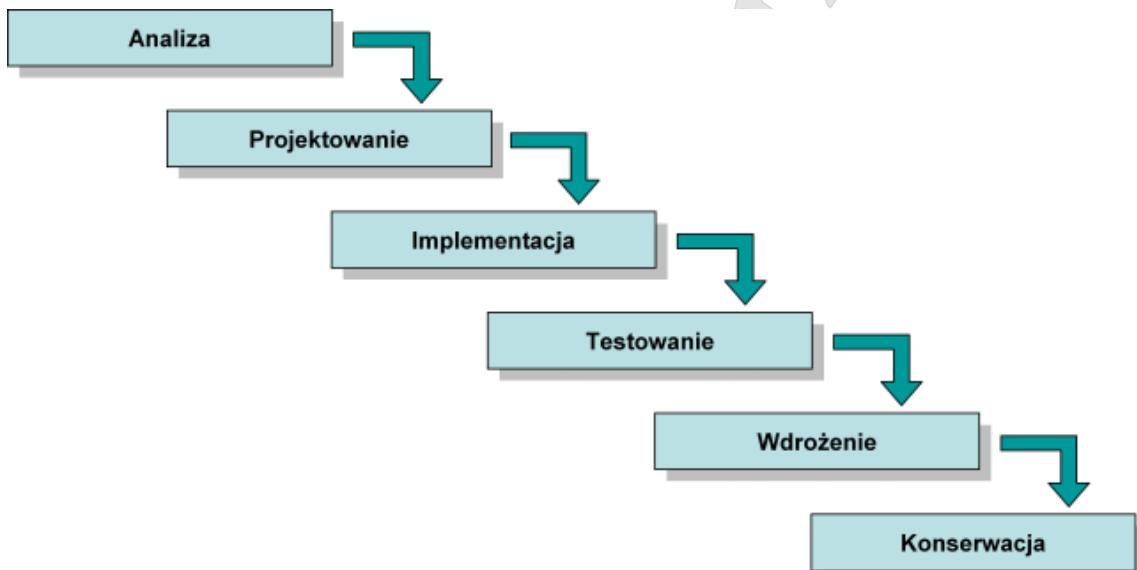
przedstawić skomplikowane cykle wytwarzania oprogramowania w zrozumiały sposób na różnym poziomie szczegółowości.

Z uwagi na indywidualne cechy każdego procesu wytwarzania oprogramowania, nie udaje się zastosować ściśle do reguł przyjętych w wybranym cyklu wytwarzającym. Często eksponuje się pewne czynniki kosztem innych, które są pomijane. Cykl wytwórczy jest rodzajem przewodnika, który wskazuje kierunek, który należy obrać, aby osiągnąć zamierzony cel, ale nie pokazuje drogi, którą należy pójść.

Na podstawie doświadczeń zebranych podczas wytwarzania oprogramowania oraz działań badawczo – rozwojowych zdefiniowano wiele typów cyklu wytwarzania oprogramowania. Poniżej zostaną przedstawione trzy najbardziej charakterystyczne.

2.3.1. Cykl wodospadowy

Definicja wodospadowego cyklu życia oprogramowania, zwanego także 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. Na Rys. 2.5 znajduje się blokowy schemat, który obrazuje, w jaki sposób następują po sobie kolejne dyscypliny cyklu wytwarzania oprogramowania. Z racji podobieństwa tego schematu do wodospadu, cykl ten nazywa się wodospadowym lub czasem kaskadowym.



Rys. 2.5: Wodospadowy cykl wytwarzania oprogramowania

Poszczególne kroki odpowiadają dyscyplinom cyklu życia oprogramowania przedstawionym w poprzedniej sekcji. Każdy krok dostarcza co najmniej jeden produkt, który podlega weryfikacji i akceptacji. Następny krok może się rozpocząć tylko po zakończeniu poprzedniego i wykorzystaniu jego produktów. W praktyce jednak, sąsiadujące ze sobą etapy zazębiają się i przekazują nawzajem informacje. Można to nazwać „mini-iteracjami”. Taka sekwencja jest odzwierciedleniem naturalnego sposobu rozwiązywania złożonych problemów przez ludzi (najpierw analiza, a później synteza), gdzie wszystkie podejmowane czynności prowadzą od potrzeb użytkownika do gotowego systemu w jednym przejściu. Umożliwia to łatwe zarządzanie zasobami (zespoły specjalistów, sprzęt). Wymaga jednak zamrożenia wyniku prac w poszczególnych fazach, aby można było przejść do następnych, aby została zachowana zasada spójności pomiędzy produktami poszczególnych dyscyplin.

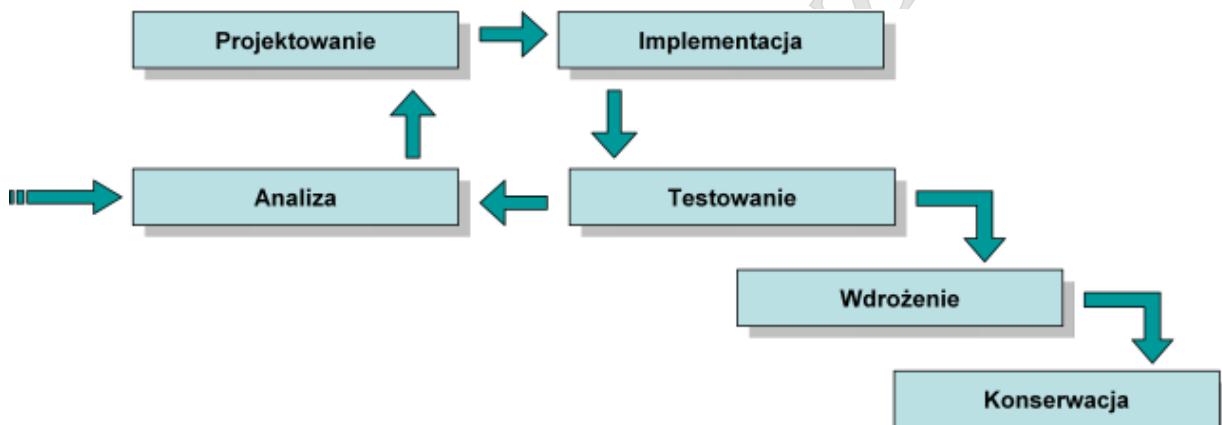
Pomimo iż cykl wodospadowy zapewnia przejrzystą ścieżkę wytwarzania oprogramowania, wiele problemów może być odkrytych dopiero na koniec cyklu. Naraża to projekt na duże ryzyko przekroczenia terminu i/lub budżetu, ponieważ każda zmiana wymaga przejścia przez kilka dyscyplin i często wprowadza chaos po końcowej walidacji systemu. Jest to niewątpliwie wadą tego cyklu,

podobnie jak rozłączne prowadzenie prac przez poszczególne zespoły. Znacznie wydłuża to długość cyklu, ponieważ następna faza może zacząć się tylko po zakończeniu poprzedniej. Kolejnym wadą cyklu wodospadowego jest przyjęcie za ostateczną specyfikację wymagań stworzoną na początku cyklu. Bardzo trudno jest stworzyć poprawną i kompletną specyfikację już na początku projektu, ponieważ wymagania często są odkrywane lub zmieniane w trakcie trwania budowy oprogramowania, a każda modyfikacja wymaga cofnięcia się do faz poprzednich.

Wodospadowy cykl wytwarzania oprogramowania, mimo iż jest najstarszy i posiada wiele wad, wciąż jest popularny. Powodzenie jego wykorzystania mocno zależy od stabilności i poprawności specyfikacji wymagań. Dlatego jest polecaný do krótkich projektów, w których wymagania są dobrze określone i zrozumiałe, a ich specyfikacja jest w pełni przygotowana.

2.3.2. Cykl iteracyjny

Duże ryzyko związane z „zamrażaniem” produktów poszczególnych faz cyklu wytwórczego oprogramowania (cykl wodospadowy), może być wyeliminowane przez wprowadzenie wielokrotnego przechodzenia przez powiązane ze sobą dyscypliny analizy i syntezy. To podejście jest główną cechą iteracyjnego cyklu wytwarzania oprogramowania. Na **Błąd! Nie można odnaleźć źródła odwołania.** znajduje się schemat blokowy, przedstawiający uproszczoną sekwencję dyscyplin cyklu oraz ich iteracje.



Rys. 2.6: Iteracyjny cykl wytwarzania oprogramowania

Choć wytwarzanie oprogramowania w cyklu iteracyjnym nie daje gwarancji na końcowy sukces procesu wytwórczego, to zwiększa szanse na jego powodzenie. Przyczynia się do tego włączenie zamawiających i przyszłych użytkowników systemu w budowę oprogramowania. Poprzez powtarzające się dyscypliny analizy, projektowania, implementacji, tworzony jest system odpowiadający ciągle odkrywanym i zmieniającym się rzeczywistym potrzebom. Testowanie (często połączone z wdrożeniem), które również jest częścią każdej iteracji, pozwala ocenić poprawność działania systemu, ale przede wszystkim zgodność z założeniami. Wszelkie problemy odkrywane są po każdej iteracji. Jeżeli aktualny stan oprogramowania nie odpowiada rzeczywistym potrzebom, lub funkcjonalność nie została w pełni zrealizowana, następuje przejście do fazy analizy, a następnie do fazy projektowania, implementacji aż do fazy testowania. Po zatwierdzeniu poprawności systemu przez zamawiających i przyszłych użytkowników następuje faza wdrożenia i konserwacji.

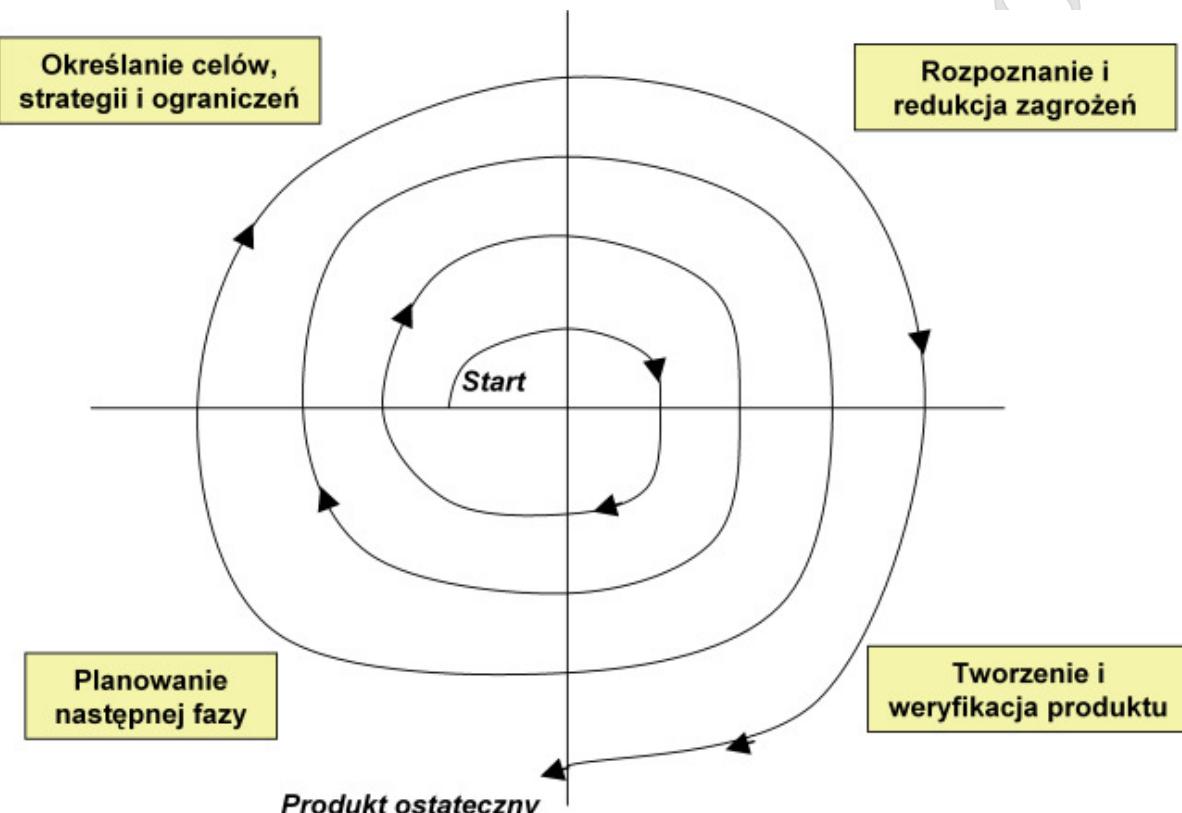
Cykł iteracyjny pozwala na przyrostowe tworzenie systemu oprogramowania. Daje to możliwość ustalenia priorytetów części systemu oraz kolejności ich wykonania i wdrożenia. Uproszczeniu ulega także zarządzanie wprowadzaniem zmian w tworzonym oprogramowaniu, które odzwierciedlają zmieniające się ograniczenia 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 wszelkimi pracami. Zwykle niesie to ze sobą zwiększenie kosztów wykonania projektu i wydłuża czas od rozpoczęcia prac do ostatecznego ich zakończenia.

Iteracyjny cykl wytwarzania oprogramowania sprawdza się w budowie dużych systemów, gdzie nie wszystkie wymagania są dobrze zdefiniowane, a nawet rozpoznane.

2.3.3. Cykl spiralny

Kolejnym szeroko wykorzystywanym cyklem wytwarzania oprogramowania jest cykl spiralny. Główną przyczyną jego powstania była próba powiązania najlepszych cech wodospadowego i iteracyjnego cyklu wytwarzania oprogramowania oraz wzbogacenie go rozbudowaną analizę ryzyka. Rys. 2.7 obrazuje ścieżkę procesu wytwarzania oprogramowania prowadzącą od postawienia problemu (punkt na wykresie na lewo od początku układu współrzędnych) do dostarczenia końcowego produktu poprzez spiralę zgodną z ruchem wskazówek zegara.



Rys. 2.7: Spiralny model wytwarzania oprogramowania

Każda pętla spirali podzielona jest na cztery części odpowiadające kolejnym ćwiartkom układu współrzędnych i może reprezentować dowolną fazę procesu wytwarzania. 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 wytwarzania, można wykorzystać do wytworzenia spodziewanego produktu podejście kaskadowe, a w innej pętli – podejście迭代性的. Na początku każdej pętli spirali wyznacza się cele, strategie i ograniczenia (pierwsza ćwiart-

ka), 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 następnego 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 jawnie 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 wybór spiralnego cyklu największym zagrożeniem dla projektu.

2.4. Podsumowanie

Celem tego rozdziału było ogólne przedstawienie wszystkich czynności związanych z wytwarzaniem oprogramowania. Łatwo zauważać, że wytwarzanie oprogramowania opiera się na naturalnym dla ludzi mechanizmie analizy i syntezy. To, co sprawdza się w naszym codziennym życiu, sprawdza się także w przypadku skomplikowanych problemów informatycznych, poprzez podział procesu twórczego na fazy analizy, projektowania, implementacyjną, testowania, wdrożenia i konserwacji. Aby zachować spójność pomiędzy poszczególnymi fazami, powinna następować pomiędzy nimi wymiana informacji poprzez odpowiednie produkty. Są to zwykle dokumentacje służące do opisu rzeczywistości i systemu informatycznego na różnym poziomie abstrakcji.

Aby zapanować nad złożonością procesu, kieruje się nim wg wybranego 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.

Celem stosowania się do zasad, określonych przez wybrany typ cyklu twórczego, jest minimalizacja ryzyka zakończenia się procesu twórczego niepowodzeniem. Należy pamiętać, iż pomimo konieczności odstępstw od reguł zawartych w definicji wybranego cyklu, to należy się kierować jego główną ideą. Da to na pewno lepsze rezultaty niż wytwarzanie oprogramowania w chaotyczny i niezorganizowany sposób.

3. Metodyki wytwarzania oprogramowania

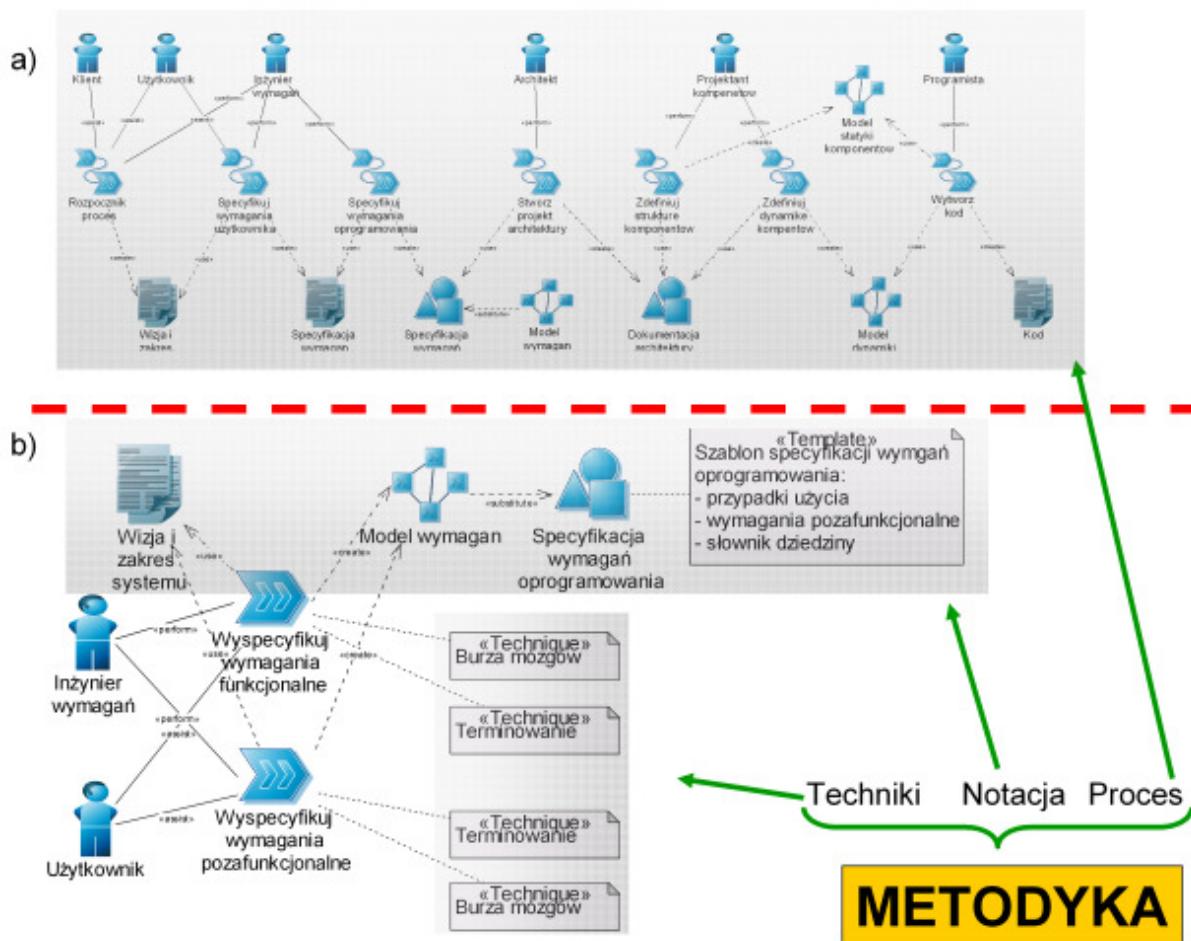
Wytwarzanie oprogramowania zgodnie z wybranym cyklem twórczym, mimo iż zmniejsza ryzyko niepowodzenia przedsięwzięcia, to nie eliminuje go całkowicie. Aby wyeliminować przyczyny niepowodzeń przedsięwzięć budowy oprogramowania, takich jak np.: nieprecyzyjna specyfikacja, zła komunikacja, brak właściwej architektury, brak zarządzania złożonością systemu, bardzo późne wykrywanie niespójności, niewystarczające testy, brak zarządzania zmianami, zbyt późna weryfikacja potrzeb klienta, oprogramowanie należy wytwarzać zgodnie z pewną metodyką.

Poniższy rozdział przedstawia metodyki wytwarzania oprogramowania, jako usystematyzowany i zorganizowany zbiór czynności (techniki metodyk) objętych procesem twórczym, którego produkty wyrażone są za pomocą odpowiedniej notacji. Aby skutecznie wykorzystać metodykę wytwarzania oprogramowania, należy ją wdrożyć zarówno w obrębie organizacji, jak również w obrębie projektu (te zagadnienie zostaną omówione w rozdziale 13). Pozwoli to na organizację i przeprowadzenie procesu twórczego według określonych standardów i z wykorzystaniem przygotowanych narzędzi (notacja, techniki).

Wśród wielu zdefiniowanych metodyk, można wyróżnić dwie główne grupy: metodyki formalne i agilne (*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 metodyk, umożliwia zrozumienie idei stosowania metodyk agilnych i formalnych. Pozwoli to także poznać cechy wspólne i rozwijające obu grup.

3.1. Co to jest metodyka wytwarzania oprogramowania

Sposobem opisu metody realizacji pewnych zadań, w oderwaniu od merytorycznego kontekstu danego obszaru, jest zastosowanie standardu w postaci metodyki. W skrócie – metodyka udziela odpowiedzi na pytanie „*Jak należy to zrobić?*”. Definicję tą można przenieść na metodykę wytwarzania oprogramowania, jako przepis postępowania, którego efektem jest system oprogramowania. Przepis taki zawiera podział całego procesu wytwarzania oprogramowania na fazy projektu, scenariusze postępowania w każdej z faz, reguły przejścia od jednej fazy do fazy następnej, produkty tworzone w każdej fazie (modele i dokumentacje), notacje używane do wyrażania produktów, role uczestników procesu i ich obowiązki.



Rys. 3.1.: Metodyka wytwarzania oprogramowania; a) proces wytwórczy, b) podprocesy związane ze specyfikacją wymagań oprogramowania oraz związane z nimi notacje i techniki

Metodykę wytwarzania oprogramowania definiuje się za pomocą trzech składowych, które przedstawione zostały na Rys. 3.1. Są to:

- notacja – inaczej język lub zbiór języków i stylów, formularzy i szablonów, w którym można wyrazić produkty powstające we wszystkich fazach procesu wytwórczego,
- techniki – opis czynności przypisanych do konkretnych ról, które prowadzą do powstania produktów (wyrażonych w odpowiedniej notacji) każdej fazy procesu wytwórczego,
- proces wytwórczy – uporządkowanie czynności zgodnych z technikami w spójny, zarządzalny przebieg pracy przez wszystkie fazy procesu wytwórczego, prowadzący do powstania końcowego systemu oprogramowania.

Te trzy składniki, za pomocą których wyrażone powinny być wszystkie metodyki wytwarzania oprogramowania zostaną omówione szerzej w dalszej części tego rozdziału.

3.1.1. Notacja

Zgodnie z definicją, każda metodyka zawiera pewną notację. Główną rolą notacji jest umożliwienie zapisu wyniku poszczególnych faz procesu wytwarzania oprogramowania (patrz Rys. 3.1, część b). Nie jest to jednak jedyne zadanie notacji. Jest to także doskonały środek, który poprzez wspomaganie ludzkiej pamięci i wyobraźni, pozwala na spójną reprezentację złożonych pomysłów, problemów biznesowych i projektowych oraz ułatwia ich rozwiązanie. Produkty stworzone z wykorzystaniem przyjętej notacji stanowią doskonałe medium komunikacji pomiędzy członkami zespołu realizacyjnego oraz do komunikacji z klientem. Dzięki przyjęciu notacji, odpowiadającym warunkom trwania procesu wytwórczego, możliwe jest także zachowanie spójności wszystkich produktów oraz zależności między nimi. Aby przyjęta notacja w pełni realizowała przedstawione powyżej funkcje, powinna posiadać kilka głównych cech:

- powszechność – notacja powinna być znana wszystkim osobom biorącym udział w procesie budowy oprogramowania. Poszczególne części notacji mogą być przeznaczone dla konkretnych ról, musi istnieć jednak część wspólna.
- łatwość używania – sposób wykorzystywania notacji nie powinien znaczaco zwiększać nakładu pracy na tworzenie produktów procesu wytwarzanego.
- łatwość nauki – notacja powinna być łatwa do nauki dla osób, które dołączają do zespołu realizacyjnego. Potrzebna jest odpowiednia dokumentacja, która maksymalnie ułatwia naukę.
- przejrzystość i precyza – produkty stworzone z wykorzystaniem notacji powinny posiadać strukturę oraz zapewnić precyzyjne przedstawienie treści.
- kompletność – notacja powinna umożliwiać spójne wyrażanie wszystkich produktów procesu wytwarzanego.
- nawigowalność – możliwe jest wiązanie ze sobą produktów, a także ich wewnętrznych części.

Najczęściej spotykane sposoby opisu produktów cyklu wytwarzania oprogramowania, to komentarz w kodzie, dokumenty tekstowe, modele. W Tabl. 3-1 przedstawione są indywidualne cechy poszczególnych sposobów opisu.

Tabl. 3-1. Różne typy notacji metryk oprogramowania

Komentarz w kodzie	Dokumenty tekstowe	Modele
Opis	Ustrukturyzowany zapis tekstowy i numeryczny	Notacje graficzne
<ul style="list-style-type: none"> Bardzo dobra praktyka – pozwala programistom na utrzymanie kodu w przyszłości 	<ul style="list-style-type: none"> Mogą być tworzone zgodnie ze standardami (szablony) Nie potrzeba specjalnych narzędzi (wystarczy edytor tekstu) 	<ul style="list-style-type: none"> Wizualna notacja ma przewagę nad tekstem Kod może wygenerowany pół-automatycznie z modeli Zalety notacji graficznych potwierdzają badania psychologiczne
<ul style="list-style-type: none"> Ale: opisuje tylko lokalne decyzje w obrębie procedur i modułów 	<ul style="list-style-type: none"> Ale: ma tendencje do dezaktualizacji (ludzie nie lubią tworzenia dokumentów i ich aktualniania) Ale: ciężko zarządzać zależnościami między dokumentami 	<ul style="list-style-type: none"> Ale: potrzebne jest specjalne narzędzie CASE¹

Jak widać z powyższego zestawienia, sposoby opisu wykorzystywane w inżynierii oprogramowania uzupełniają się nawzajem. Dlatego notacja konkretnej metodyki może być definiowana jako zbiór odpowiednio dobranych sposobów opisu, języków i ich podzbiorów, który posiada pożądane cechy. Na potrzeby inżynierii oprogramowania powstało wiele języków i wzorców dokumentów, opisujących produkty procesu wytwarzania oprogramowania. Można za ich pomocą tworzyć opisy tekstowe, graficzne, jak również samo oprogramowanie.

Modele

¹ CASE – Komputerowo Wspomagana Inżynieria Oprogramowania (ang. Computer Aided Software Engineering)

Obecnie popularność zyskały języki wspierające modelowanie obiektowe – najważniejszym z nich jest UML (*Unified Modelling Language*) opracowany i rozwijany przez OMG (*Object Modelling Group*). Jest to język formalny służący do opisu świata obiektów w analizie obiektowej. Pozwala on na przedstawienie dziedziny problemu, jego statyki oraz dynamiki. Wykorzystuje się go również do zobrazowania architektury i projektów szczegółowych, algorytmów i struktur danych. Więcej informacji o modelowaniu obiektowym i języku UML znajduje się w następnych rozdziałach tej książki.

Kolejną notacją umożliwiającą tworzenie modeli oprogramowania jest opracowana przez OMG, BPMN (*Business Process Modelling Notation*). Pozwala ona w graficzny sposób opisać procesy biznesowe w postaci przepływu. Choć notacja ta nie będzie omawiana w tej książce, to warto o niej wspomnieć, ze względu na szybkie zdobywanie przez nią coraz większej popularności.

Zarówno UML, jak i BPMN zaliczają się do grupy języków ogólnego przeznaczenia (*General-Purpose Modeling GPM*). Przeciwieństwem tej grupy jest grupa języków dla specyficznych dziedzin (*Domain-Specific Language DSL*). Języki te wspierają wyższy poziom abstrakcji, niż języki klasy GPM, przez co pozwalają na łatwiejsze modelowanie. Przykładem języka DSL może być opracowany przez OMG SysML wspierający specyfikację, analizę, projektowanie, weryfikację i walidację systemów oprogramowania.

Modele w języku UML, BPMN, SysML (lub innych) można tworzyć na kartce papieru. Jednak aby tworzyć je i zarządzać nimi w sprawny sposób potrzebne są specjalne narzędzie, które wspomagają projektowanie – narzędzia CASE.

Dokumenty tekstowe

Czasem metodyka wymusza stosowanie dokumentów tekstowych. Zwykle są to różnego rodzaju dokumentacje i raporty, a także tekstowe opisy modeli. Ich tworzenie polega na wypełnianiu szablonów i wzorców lub na ścisłym przestrzeganiu zasad tworzenia dokumentu. Przykładem może być szablon specyfikacji wymagań, listy, okólniki, harmonogramy, ankiety.

Wspomniane szablony i wzorce tworzenia dokumentów tekstowych, traktowane są jako notacja wybranej metodyki. Warto zauważyć, że każda metodyka może definiować własne standardy tworzenia tego typu dokumentów.

Kod

Jednym z produktów procesu wytwarzania oprogramowania jest kod stworzony w wybranym języku oprogramowania. Wybór jednego lub więcej spośród szerokiego zbioru języków obiektowych i strukturalnych, podyktowany jest cechami wybranej metodyki, jak również preferencjami zamawiającego i wykonawców. Ważnym składnikiem notacji, obok kodu, są także komentarze. Istnieją mechanizmy generowania dokumentacji kodu na podstawie komentarzy. Przykładem może być JavaDoc dla języka Java.

Produkty procesu wytwarzczego, w zależności od przyjętej metodyki, można opisywać na wiele sposobów. Nie należy jednak utożsamiać notacji z metodyką, ponieważ jedna notacja (lub jej części) mogą być składową wielu metodyk. Głównym celem wprowadzania odpowiedniej notacji jest ułatwienie komunikacji między członkami zespołu, systematyzacja sposobu wyrażania produktów.

3.1.2. Techniki

Oprócz notacji, metodykę stanowią także techniki. Pod pojęciem „technika” definiujemy opis czynności, których wykonanie prowadzi do powstania produktów poszczególnych faz procesu wytwarzania oprogramowania wyrażonych za pomocą odpowiedniej notacji (patrz Rys. 3.1, część b). Często są to instrukcje „krok po kroku”, rady, wskazówki, zalecenia, sposoby postępowania, najlepsze praktyki. Poszczególne metodyki znacznie różnią się pod względem ilości i rodzajów technik. Przyjmując pewne uogólnienie, można stwierdzić, że metodyki formalne (OPEN, UP), zawierają dużo większą liczbę technik niż metodyki agilne (XP, SCRUM).

Główną cechą technik oferowanych przez metodyki formalne, jest praca oparta na dokumentach, które są produktami poszczególnych faz procesu wytwarzczego. Dotyczą one wytwarzania jak najbardziej poprawnych dokumentów w ścisłe określony sposób i wspomagają ich przepływ. Do-

kumenty te następnie są podstawą do komunikacji i interakcji. Przykładami takich technik mogą być:

- metody tworzenia poprawnego modelu przypadków użycia,
- wytyczne dla tworzenia otwartych architektur komponentowych,
- instrukcje śledzenia zmian,
- kontrola jakości w poszczególnych fazach procesu twórczego,
- formalne przekształcenia modeli.

Cechą charakterystyczną technik oferowanych przez metodyki agilne jest orientacja na bezpośrednią interakcję między ludźmi i bazujący na niej udział całego zespołu we wszystkich fazach procesu twórczego. Techniki te, w odróżnieniu od metodyk formalnych, mniejszą uwagę skupiają na konieczności tworzenia dokumentów, a większy nacisk kładą na wytwarzanie oprogramowania, które ewoluje do końca procesu twórczego. Przykładami takich technik mogą być:

- programowanie w parach,
- wspólna własność kodu,
- sesje wspólnego projektowania systemu,
- tworzenie prostych modeli wymagań i systemu,
- wspólne sesje planowania w oparciu o wymagania.

Technikom zawartym w wybranej metodyce przyporządkowani są członkowie zespołu realizacyjnego, którzy pełnią określone role projektowe. Techniki metodyk agilnych, ze względu na rodzaj technik, nie wymagają dużej liczby ról projektowych. W skrajnym przypadku będą to po prostu użytkownicy i wykonawcy, którzy dopasowują się do wykonywania określonych czynności. Często jednak występuje jasny podział na role, które jednak różnią się od podziału na role w metodykach formalnych. Tam poszczególnym technikom ściśle przypisane są poszczególne role, które mają dokładnie sprecyzowany zakres odpowiedzialności i zadania do wykonania. Liczba możliwych ról, ze względu na ilość technik, jest duża. Przykłady ról w metodykach formalnych to np.: analitycy (biznesowy, systemowy, pisarz wymagań, projektant interfejsu użytkownika), programista, architekt, projektant bazy danych, tester, kierownicy (kierownik projektu, kierownik testów).

Zbiór i kształt technik wykorzystywanych w procesie wytwarzania oprogramowania zależy od przyjętej metodyki, ale także od ich doboru i modyfikacji. Uwarunkowane jest to zwykle specyfiką projektu oraz organizacji pracy i preferencji danej organizacji.

3.1.3. Proces twórczy

Oprócz notacji i technik, kształt metodyki określany jest również przez proces twórczy. Proces ten uporządkowuje wszystkie podejmowane działania w czasie, organizuje i zapewnia spójny przebieg pracy, który prowadzi do powstania końcowego systemu oprogramowania (patrz Rys. 3.1, część a).

Aby zminimalizować ryzyko niepowodzenia procesu wytwarzania oprogramowania, wszystkie objęte nim czynności dzieli się na fazy realizacji. Podział na fazy daje większe możliwości zarządzania i nadzoru. Wynikiem działalności w każdej fazie zwykle jest pewien produkt, który powinien być podstawą prac w kolejnej fazie. Proces twórczy określa działania, jakie będą podejmowane na początku każdej fazy i co będzie wynikiem działań danej fazy.

Dla procesu twórczego ważna jest prawidłowa organizacja zespołu. Poszczególnym grupom lub osobom przyporządkowuje się role i odpowiedzialności w poszczególnych fazach procesu twórczego. Zależnie od wybranej metodyki, do obsługi procesu twórczego, stosuje się różne modele struktury organizacyjnej zespołu. Więcej informacji o organizacji zespołu znajduje się w rozdziale 13.

Oprócz zapewnienia odpowiedniej infrastruktury i zasobów, w celu sprawnej i efektywnej realizacji zadań przez poszczególnych członków i grupy zespołu realizacyjnego, należy zarządzać całym

procesem twórczym. Ponieważ istnieją silne powiązania między działaniami podejmowanymi w różnych fazach procesu, a produkty tych faz pozostają w różnych relacjach, to zmiana jednego parametru procesu twórczego wymagać może korekty pozostałych. Przez to podążanie jedną, wyznaczoną wcześniej, linią wytwarzania systemu oprogramowania jest niemalże niemożliwe. Zarządzanie w procesie wytwarzania oprogramowania polega na aktywnym zarządzaniu związkami między poszczególnymi fazami, produktami, zespołami, osobami i kierowaniu pracą zgodnie ze zmianami zachodzącymi w trakcie trwania procesu twórczego.

Mimo iż procesy twórcze różnych metodyk zawierają pewne cechy wspólne, to każda metodyka dostarcza własny proces twórczy. Jego kształt uwarunkowany jest tym, na które działania kładziony jest akcent.

Jeden z pierwszych procesów twórczych został zdefiniowany w Departamencie Obrony USA. Jest to proces oparty o kaskadowy cykl życia oprogramowania, gdzie wszystkie zadania podzielone na fazy układają się w kaskadę prowadzącą od zainicjowania połączonego z analizą do wdrożenia i utrzymania systemu. Tego typu proces twórczy jest mało skomplikowany i łatwy w zarządzaniu, ponieważ występuje jeden przebieg przez wszystkie fazy.

Inaczej wyglądają procesy twórcze oparte o cykl iteracyjny i spiralny. Kolejne fazy prowadzące dotworzenia oprogramowania, które występują w procesach opartych o cykl kaskadowy, powtarzane są cyklicznie. Każdy cykl dostarcza coraz bardziej rozbudowanych i szczegółowych produktów, które w ostatnim cyklu stanowią produkty pośrednie i produkt końcowy w wersjach ostatecznych. Ilość iteracji zależy od złożoności projektu oraz od innych szczegółowych uwarunkowań realizacyjnych. Poszczególne fazy mogą się nakładać. Znaczco przyspiesza to realizację i pozwala zaangażować maksymalnie dostępne zasoby, zwiększając tym samym ilość zadań zarządczych.

Z uwagi na różną specyfikę problemów i warunków, w jakich mają zostać zbudowane systemy oprogramowania, które będą te problemy rozwiązywać, stosuje się różne procesy twórcze. Do krótkich przedsięwzięć, gdzie nie wszystkie wymagania są dokładnie zdefiniowane, lepiej będą sprawdzały się procesy oparte o cykl iteracyjny i spiralny. Procesy takie zawarte są w metodykach agilnych. W przypadku dużych przedsięwzięć, gdzie specyfikacja wymagań jest kompletna, lepiej będą się sprawdzać procesy oparte o cykl iteracyjny i wodoszadowy (w zależności od organizacji pracy i oczekiwania klienta). Procesy takie są cechą charakterystyczną metodyk formalnych. Procesy twórcze dostarczone przez wybraną metodykę zwykle się modyfikują. Jest to podyktowane koniecznością dostosowania procesu do indywidualnych potrzeb danego przedsięwzięcia.

3.2. Rodzaje metodyk wytwarzania oprogramowania

Przegląd zdefiniowanych metodyk wytwarzania oprogramowania pozwala podzielić je na dwie główne grupy. Są to metodyki formalne i metodyki agile (zwinne). Choć, podział jest jednoznaczny i występuje wiele cech rozłącznych, to oba rodzaje metodyk posiadają wiele cech wspólnych. Dzięki temu dana metodyka może być wzbogacona elementami innej metodyki.

Żeby wybrać odpowiednią metodykę i efektywnie ją wykorzystać w procesie wytwarzania oprogramowania, najpierw należy poznać główne cechy metodyk formalnych i agilnych oraz specyfikę poszczególnych metodyk. W poniższych sekcjach zostaną przedstawione dwa rodzaje metodyk, a krótki przegląd wybranych metodyk, pozwoli lepiej zrozumieć zasady ich wykorzystania.

3.2.1. Metodyki formalne

Metodyki formalne są zorientowane na proces, a właściwie na wiele procesów. W każdej fazie procesu wytwarzania oprogramowania zachodzą różne procesy, których celem jest dostarczenie produktów potrzebnych do rozpoczęcia następnej fazy. Sprawia to wrażenie, jakby metodyki formalne były sterowane przede wszystkim formalnymi dokumentami produkowanymi podczas cyklu twórczego. Zwykle u podstaw metodyk formalnych stoją pozytywne i negatywne doświadczenia zebrane w wielu projektach. Poza faktycznym wytworzeniem systemu oprogramowania, metodyki formalne kładą duży nacisk na czynności związane z zarządzaniem integralnością projektu, zakre-

sem, czasem, kosztami, jakością, zasobami ludzkimi, komunikacją, ryzykiem, zaopatrzeniem. Procesy przyporządkowane czynnościom zarządczym można podzielić na 4 grupy. Są to:

- procesy rozpoczęcia - procesy, które służą zdefiniowaniu i zatwierdzeniu projektu w organizacji;
- procesy realizacji - grupują i koordynują wykorzystanie zasobów i ludzi w projekcie w celu wykonania założonego planu,
- procesy kontroli - monitorują postępy prac w projekcie, badają ewentualne odchylenia, aby w razie konieczności uruchomić odpowiednie działania zapobiegawcze lub/i korygujące,
- procesy zakończenia – przygotowanie formalnej akceptacji produktu finalnego projektu lub jego faz.

Procesy wytwarzające metodyk formalnych dostarczają szablon organizacyjny procesów pochodzących z wyżej wymienionych grup. Zwykle procesy nachodzą na siebie. Wykonanie konkretnych procesów uzależnione jest od wybranej metodyki i od specyfiki danego projektu. Najbardziej popularne metodyki formalne, które są szeroko wykorzystywane przez organizacje wytwarzające oprogramowanie to:

- IBM: Rational Unified Process (RUP) lub Unified Process (UP)
- Borland: Application Lifecycle Management (ALM)
- Microsoft: Microsoft Solution Framework (MSF)
- Eiffel Software: Eiffel Development Framework
- OPEN Consortium: O-O Process Environ. and Notation (OPEN)

3.2.2. Metodyki agilne (zwinne)

Metodyki agilne (zwinne) realizują zasady, które zawarte są w Manifeście Zwinnego Wytwarzania Oprogramowania (*Agile Manifesto, Manifest for Agile Development*). Został on opracowany w 2001 roku, a jego celem było zdefiniowanie zasad alternatywnego do tradycyjnego sposobu wytwarzania oprogramowania. Główny nacisk położony został na bezpośrednią interakcję między ludźmi (klienci, wykonawcy) i opartą o nią poprawną implementację i ciągłą validację systemu oprogramowania.

Treść manifestu:

Poprzez wytwarzanie oprogramowania oraz pomaganie innym w tym zakresie odkrywamy lepsze sposoby realizowania tej pracy. W wyniku tych doświadczeń zaczeliśmy przedkładać:

- Jednostki i współdziałania między nimi nad procesy i narzędzia.
- Działające oprogramowanie nad dokładną dokumentację.
- Współpracę z klientem nad negocjacją umów.
- Reagowanie na zmiany nad realizowanie planu.

Autorzy manifestu, mimo iż wprowadzają i promują nowe zasady wytwarzania oprogramowania, to wcale nie odrzucają starych. Kolejną główną cechą metodyk agilnych, obok zwiększonego nacisku na integrację między ludźmi, jest adaptacyjność procesu wytwarzycza i technik do możliwych zmian. Informacji o konieczności uaktualnienia dostarczają częste kontrole polegające na częstym produkowaniu działającej wersji systemu oprogramowania i ocenę jej przez przedstawicieli klienta. Dzięki takiemu podejściu, ciągłe testowanie kolejnych wersji oprogramowania nie tylko sprawdza poprawność wytworzonego kodu, ale także pozwala odkrywać nowe wymagania lub ewentualnie modyfikować stare. Stały udział klienta w procesie wytwarzycza pozwala sterować kolejnymi pracami poprzez określanie priorytetów realizacji poszczególnych funkcjonalności. Wytwarzanie oprogramowania w ciągłym cyklu kodowania, testowania i integracji zmniejsza ryzyko błędnej integracji całego systemu, ponieważ kolejne jego elementy są intensywnie testowane pod kątem działania, ale również pod kątem integracji z resztą systemu.

Poniżej wymieniono najbardziej popularne metodyki agilne:

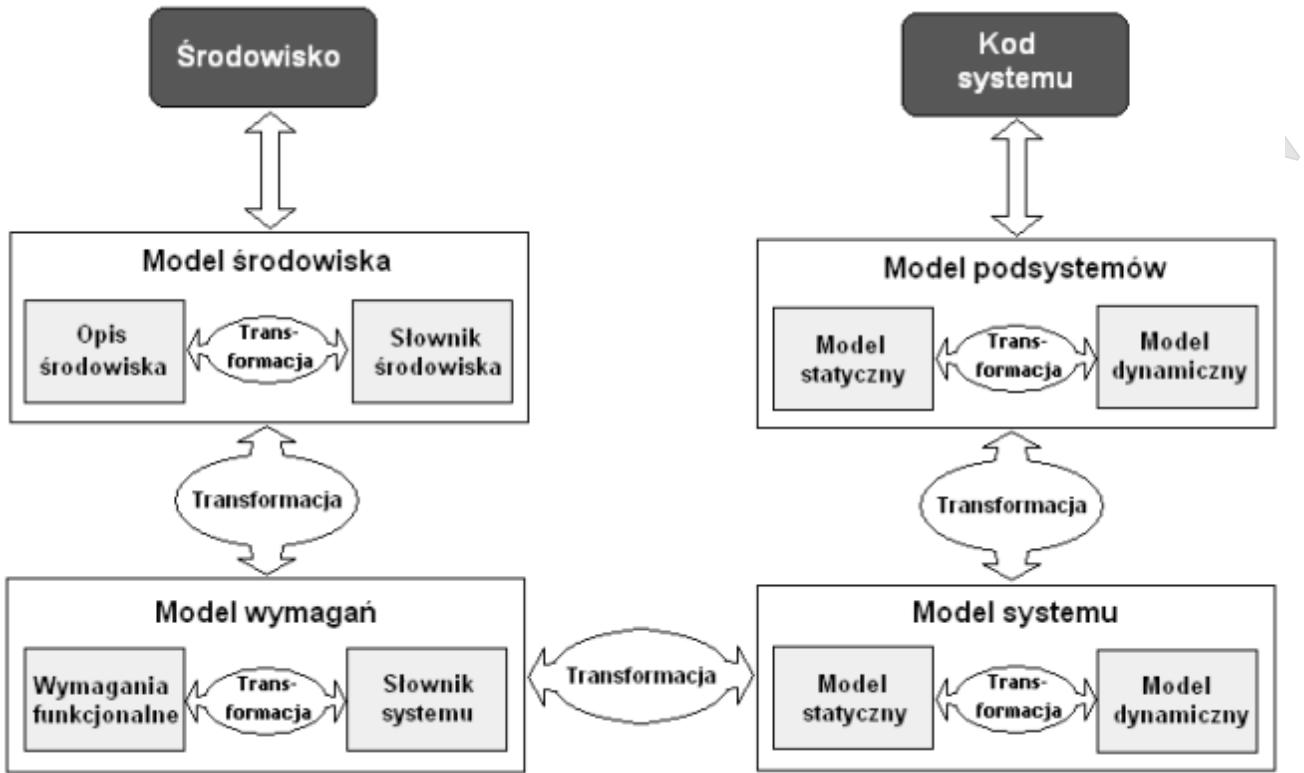
- SCRUM
- FDD – Feature Driven Development
- Pragmatic Programming
- Agile ICONIX
- Kent Beck: eXtreme Programming (XP)
- DSDM Consortium: Dynamic Systems Dev. Method
- Alistair Cockburn: Crystal
- Jim Highsmith: Agile Project Management

3.3. Wytwarzanie oprogramowania sterowane modelami (MDD)

Kolejnym sposobem na zmniejszenie ryzyka niepowodzenia procesu budowy oprogramowania jest zapewnienie ścisłej spójności między produktami kolejnych faz i całościowe spojrzenie na budowany system oprogramowania w różnych fazach. Można to osiągnąć poprzez budowanie modeli na różnych poziomach abstrakcji, co pozwala zrozumieć różne aspekty funkcjonowania środowiska oraz systemu. Zgodność kodu systemu z definicją środowiska może być zapewniona poprzez odpowiednie transformacje poprzez modele pośrednie (wymagania, architektura, itd).

Zamawiający system i jego przyszli użytkownicy oraz programista znajdują się na dwóch skrajnych biegunach w całym procesie powstawania oprogramowania. Zamawiający oraz użytkownicy funkcjonują w określonym środowisku biznesowym (przedsiębiorstwie), które rządzi się pewnymi regulamini i w którym zachodzą określone procesy. Posługują się oni pojęciami występującymi w tym środowisku. Zadaniem programisty natomiast, jest stworzenie takiego kodu systemu, który będzie dobrze odwzorowywał te pojęcia - będzie zgodny ze środowiskiem oraz oczekiwaniemi użytkowników. Jest to zadanie tym bardziej trudne, że częste zmiany w funkcjonowaniu środowisk biznesowych, wymuszają zmiany wymagań w stosunku do systemów informatycznych. Zmiany te powinny znajdować szybkie odzwierciedlenie w kodzie systemów.

Aby sprostać temu zadaniu, potrzebny jest sposób transformacji modelu środowiska, sporządzanego w wyniku współpracy analityków środowiska i zamawiającego, na model kodu, który posłuży programistom do stworzenia działającego systemu. Transformacja taka (Rys. 3.2) powinna się składać z szeregu drobniejszych przekształceń - tzw. transformacji pionowych oraz poziomych. W wyniku transformacji pionowych otrzymujemy modele odpowiadające kolejnym etapom ścieżki prowadzącej od środowiska do kodu. Na każdym etapie tej ścieżki wykorzystuje się zarówno modele wyrażające dynamikę oraz modele statyczne. Przekształcenia tych modeli między sobą to transformacje poziome.



Rys. 3.2.: Przejście od opisu środowiska do kodu według koncepcji MDA

Sposób wytwarzania oprogramowania, oparty na budowie modeli oraz ich transformacjach, został zaproponowany przez organizację OMG i określany jest jako sterowane modelami wytwarzanie oprogramowania (*ang. Model Driven Development - MDD*) i jest oparte o architekturę sterowaną modelami (*ang. Model Driver Architecture – MDA*). Koncepcja MDD mówi o tworzeniu systemu informatycznego jako ciągu przekształceń jednego modelu w inny, według określonych reguł, który prowadzi od wymagań środowiska do działającego kodu systemu, spełniającego te wymagania. MDA zakłada wykorzystanie przede wszystkim języka UML do wyrażania modeli.

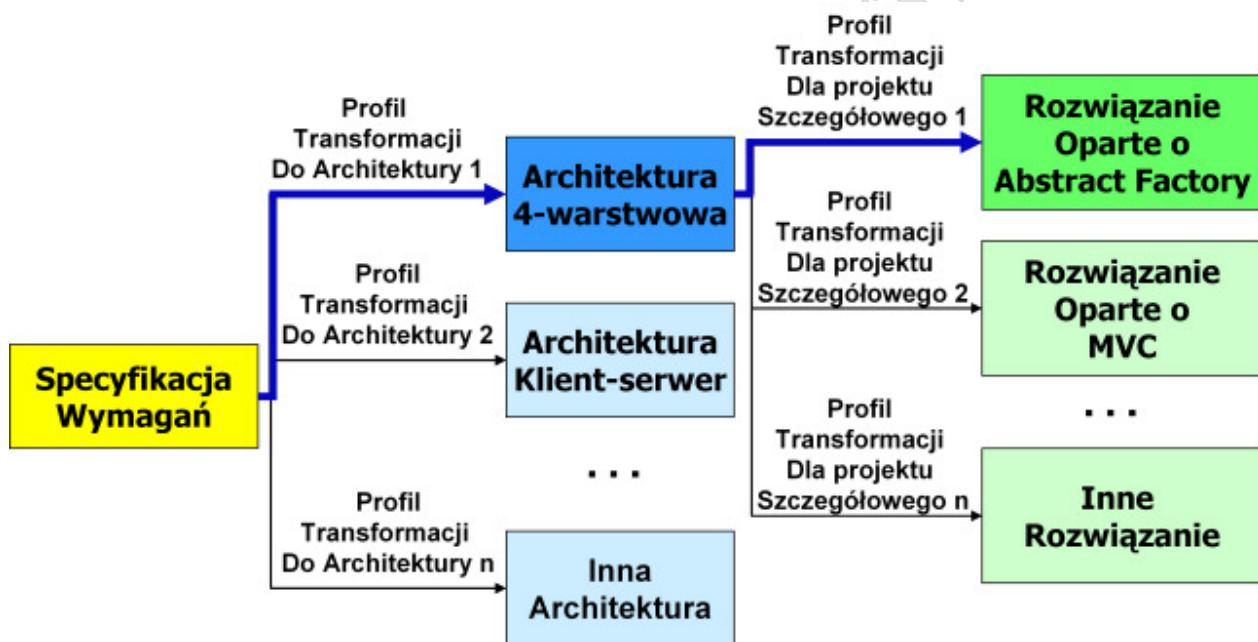
Zasadniczą kwestią w podejściu MDA jest odseparowanie specyfikacji funkcjonalnej systemu od sposobu jego realizacji. Dzięki takiemu podejściu, model wymagań oraz szczegóły realizacyjne stanowią rozłączne elementy mogące ewoluować niezależnie od siebie. Gdy zmienią się wymagania, nowy ich model można szybko przekształcić w model niższego poziomu. MDA definiuje trzy główne poziomy modelowania, pozwalające przedstawić system z różnych perspektyw. Perspektywy pozwalają ukazać różne aspekty systemu, określają poziom szczegółowości modelu oraz obecność w nim elementów specyficznych dla konkretnej implementacji.

- **Computation Independent Model (CIM).** Na tym poziomie modelować można zarówno środowisko, jak też wymagania funkcjonalne dla systemu. Do specyfikowania modeli na tym poziomie wykorzystuje się słownictwo charakterystyczne dla danego środowiska biznesowego. Model CIM odgrywa zatem bardzo ważną rolę w komunikacji pomiędzy analitykami środowiska a zamawiającym oraz analitykami wymagań a użytkownikami. Model ten powinien być pozbawiony elementów związanych z realizacją systemu.
- **Platform Independent Model (PIM).** Model PIM ukazuje sposób działania systemu, ukrywając szczególne dla konkretnej platformy. Pokazuje on tę część systemu, która pozostaje niezmienna niezależnie od implementacji. Przykładem może być model architektury komponentowej, która zbudowana jest według zadanego schematu (np. architektura 4-warstwowa).
- **Platform Specific Model (PSM).** Jest to model systemu wraz ze wszystkimi szczegółami implementacyjnymi, specyficznymi dla konkretnej platformy i konkretnych rozwiązań technologicznych. Przykładem mogą być projekty szczegółowe komponentów, które oparte są o

wybrany wzorzec projektowy (np. MVC) lub wykorzystują specjalistyczne biblioteki (np. szycfrowanie metodą RSA w Javie).

Aby tworzony system był zgodny ze środowiskiem, należy zapewnić, że modele powstające w kolejnych etapach - począwszy od modelu środowiska, a kończąc na modelu kodu - będą spójne. Również sam kod systemu powinien być stworzony zgodnie z jego modelem. Można to osiągnąć stosując jednoznaczne transformacje - zarówno pionowe, jak i poziome. Uzyskanie jednoznaczności jest jednak trudne i kosztowne bez odpowiednich narzędzi wspomagających, potrafiących przekształcać modele w sposób automatyczny lub półautomatyczny.

Na Rys. 3.3 przedstawiony jest sposób wykorzystania koncepcji MDD w przejściu od specyfikacji wymagań (CIM) oznaczonej kolorem żółtym, przez model architektury (PIM) oznaczony kolorem niebieskim, do modelu projektowego specyficzego dla technologii (PSM) oznaczonego kolorem zielonym. Zastosowanie zestawu profili transformacji, zawierających reguły przekształceń i mapowań, pozwala w łatwy sposób przejść ścieżkę konstruowania oprogramowania, które może być wytworzzone jako kombinacja różnych typów architektur i rozwiązań technologicznych. Takie podejście znacznie przyspiesza proces wytwarzania oprogramowania. Wskazania pomiędzy elementami modeli różnych poziomów abstrakcji stworzone są dzięki transformacjom i mapowaniom, pozwalały na łatwe zlokalizowanie części systemu, które są odpowiedzialne za wybraną funkcjonalność.



Rys. 3.3. Przykład wykorzystania koncepcji MDD w praktyce

Opisana w tej sekcji koncepcja nie jest pełną metodyką. Jednakże zawarte w koncepcji notacja (UML), techniki (transformacje) i elementy procesu twórczego mogą wzbogacić prawie każdą metodykę.

3.4. Podsumowanie

Metodyka jest kolejną próbą zmniejszenia ryzyka niepowodzenia przedsięwzięcia budowy oprogramowania. Głównymi jej elementami są notacja, techniki i proces twórczy. Wdrożenie zasad i narzędzi do organizacji wytwarzających oprogramowanie wyznacza kierunek prac i zapewnia powtarzalność działań podejmowanych na każdym etapie procesu budowy oprogramowania.

Większość współczesnych metodyk, oprócz notacji tekstowych (formularze, szablony, dokumenty) i kodu, wykorzystuje także modele. Obecnie najbardziej powszechnym językiem do modelowania jest UML. Proces twórczy organizuje etapy budowy oprogramowania w ciąg następujących po sobie czynności. Czynności te operując odpowiednimi technikami kierują tworzeniem i przepływem różnych produktów.

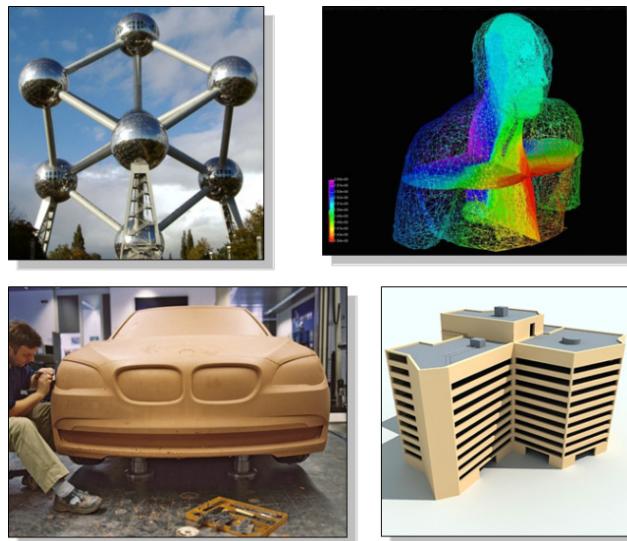
W zależności od specyfiki postawionych problemów i warunków, w jakich mają zostać zbudowane systemy oprogramowania, wykorzystuje się różne metodyki. Można wyróżnić dwie główne grupy metodyk: formalne i agilne. Metodyki formalne główny nacisk kładą na sam proces i jego produkty, które są podstawą komunikacji. Natomiast metodyki agilne główny nacisk kładą na bezpośrednią interakcję między ludźmi i adaptacyjność procesu twórczego w reakcji na bieżące potrzeby całego przedsięwzięcia.

Żadna metodyka, w swojej czystej formie, nie gwarantuje całkowitego sukcesu przedsięwzięcia wytwarzania oprogramowania. Dlatego często tworzy się kombinację elementów procesu, różnych technik i notacji, która jest w danym momencie najbardziej przydatna. Kluczem do sukcesu jest zespół doświadczonych, zaangażowanych i kompetentnych osób, których praca będzie wspomagana przez metodykę.

4. Wprowadzenie do obiektowego modelowania oprogramowania

4.1. Czym jest modelowanie?

Gdy przeglądamy słowniki w poszukiwaniu słowa „model”, napotykamy kilka definicji. Oto niektóre z nich: „wzór, według którego coś jest lub ma być wykonane”, „mężczyzna prezentujący ubiory na pokazach mody”, „konstrukcja, schemat lub opis ukazujący działanie, budowę, cechy, zależności jakiegoś zjawiska lub obiektu”, „przedmiot będący kopią czegoś, wykonany zwykle w odpowiedniej skali”. 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.



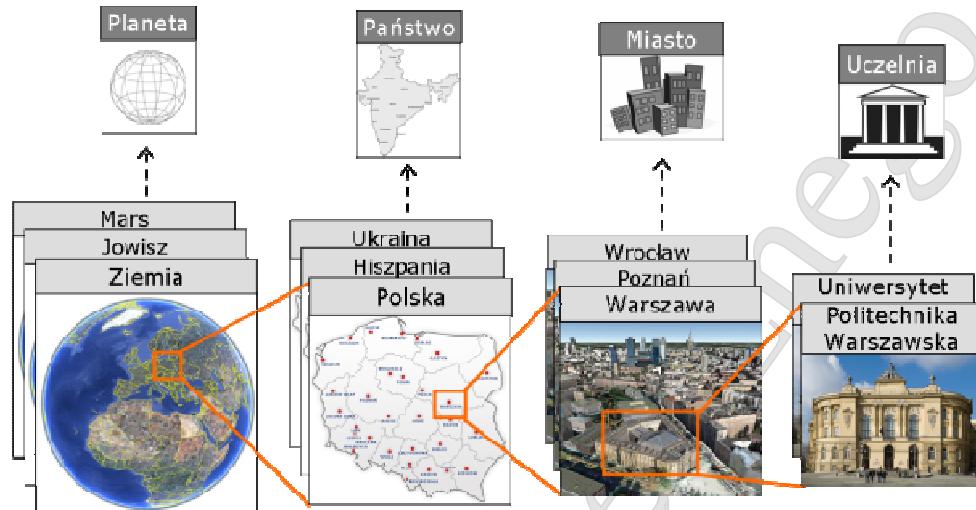
Rysunek 4.1: Przykłady różnych modeli

Budowanie modeli ma różne cele:

- 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 jeden z najważniejszych przyczyn, dla których wykonywane są modele. Rzeczywistość składa się z bardzo wielu elementów, między którymi występuje szereg powiązań i wzajemnych oddziaływań. Ludzkiego mózg, natomiast, ma ograniczoną zdolność jednoczesnego przetwarzania wielu elementów. Według badań psychologicznych, optymalna liczbą 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.

Wspomniany sposób radzenia sobie ze złożonością rzeczywistego świata jest jedną z podstawowych technik stosowanych przez ludzki umysł. Technika ta nazywa się abstrakcją. Abstrakcja jest realizowana poprzez generalizację oraz klasyfikację.

Klasyfikacja polega na grupowaniu elementów świata rzeczywistego w kategorie, zgodnie z przyjętymi kryteriami. 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 (Rysunek 4.2).

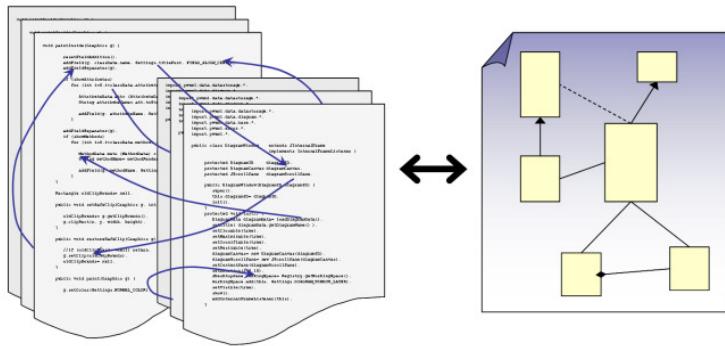


Rysunek 4.2: Przykład zastosowania zasady abstrakcji

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 nientechniczne. Innym sposobem podziału byłby podział na uczelnie państwowie i prywatne.

Budując abstrakcyjne modele, zamiast szukać prawdy absolutnej o otaczającej nas rzeczywistości, możemy skupić się na wybranym jej fragmencie, który jest najbardziej adekwatny dla celu, jaki przyświeca nam podczas budowy danego modelu. Może, zatem, istnieć wiele różnych modeli, które ukazują różne aspekty modelowanego obiektu czy zjawiska.

Oprogramowanie również jest modelem określonego fragmentu świata rzeczywistego. 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 odzwierciedlać ogromne ilości obiektów i mechanizmów zachodzących w danym środowisku, które system ma realizować. Gotowy system składa się zazwyczaj z setek tysięcy linii kodu.



Rysunek 4.3: Model, jako wizualna mapa kodu systemu

Często 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. Niewyobrażalne jest przystąpienie do pisania kodu dużego systemu jedynie na podstawie przekazanego przez zamawiającego, nieformalnego opisu środowiska. 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 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 (patrz Rysunek 4.3).

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.

Kod oprogramowania można zapisać w wielu różnych językach programowania. Obecnie, najpowszechniej wykorzystywane są obiektowe języki programowania, takie jak Java, C# czy C++. Do zapisania modeli również niezbędny jest odpowiedni język. Najpowszechniej stosowanym od pewnego czasu językiem jest UML, który można uznać obecnie za standard notacji modelowania obiektowego. UML jest graficznym językiem modelowania, który powstał w wyniku połączenia trzech różnych notacji stosowanych wcześniej w inżynierii oprogramowania. W efekcie otrzymaliśmy uniwersalny język, za pomocą którego można tworzyć modele niezbędne w całym procesie budowy systemu informatycznego. W dalszej części książki będziemy sukcesywnie przedstawiać składnię i semantykę języka UML w wersji 2.0. Pełną specyfikację języka znaleźć można na stronach internetowych organizacji OMG: <http://www.omg.org>.

4.2. Zasady modelowania złożonych systemów

Booch, Rumbaugh i Jacobson – główni twórcy języka UML – przedstawili cztery podstawowe zasady modelowania [Booc01].

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

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.*”

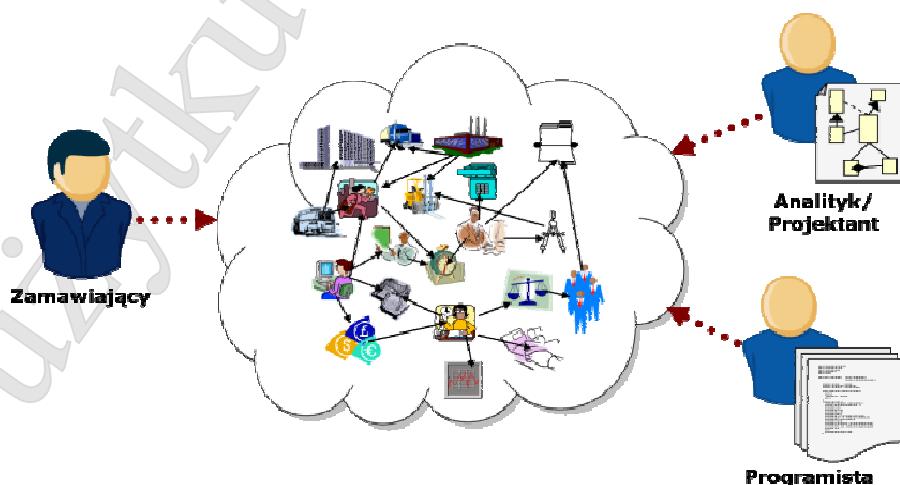
Model jest tym lepszy, im lepiej oddaje modelowaną rzeczywistość. Oczywiście, jak wspominaliśmy wcześniej, modele upraszczają skomplikowaną rzeczywistość, aby łatwiej można było ją zrozumieć. Istotne jest, zatem, aby to uproszczenie nie dotyczyło elementów istotnych. Model powinien pomijać tylko te szczegóły, 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: pojęciowa, 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 odrębnaniu od pozostałych.

4.3. Obiekty jako podstawa modelowania

System oprogramowania powinien odwzorowywać określone środowisko w taki sposób, aby spełnić wymagania zamawiającego. Problem polega na tym, że z jednej strony mamy zamawiających, którzy posiadają wiedzę na temat dziedziny, którą ma realizować system, ale brakuje im wiedzy technicznej na temat sposobów i możliwości realizacji oprogramowania. Z drugiej strony mamy programistów, którzy wiedzą jak zrealizować system od strony technicznej, jednak ich wiedza na temat dziedziny problemu jest zwykle pobiczna. Jedni i drudzy posługują się innym językiem w swojej pracy. Sposobem porozumienia się obu stron jest modelowanie obiektowe. Podstawą modelowania obiektowego są obiekty. Umożliwiają one m.in. realizację zasady abstrakcji.



Rysunek 4.4: Obiekty – wspólny język dla tworzenia modeli

Z punktu widzenia zamawiającego system, obiekty odpowiadają rzeczywistym elementom modelowanego środowiska (Rysunek 4.4). Mogą zatem być różnego rodzaju przedmiotami, osobami, zdarzeniami, procesami czy innymi tworami niematerialnymi występującymi w danym środowisku.

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 służą 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: Tożsamość, stan i zachowanie obiektów

Rysunek 4.5 przedstawia dwa samochody. Oba auta są bardzo podobne do siebie – ich marka, model, parametry techniczne i właściwości jezdne są identyczne. Różnią się jednak 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.

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 specjalną, ukrytą właściwością 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 obiektami.

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 obiektowi, 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.

Rysunek 4.6 przedstawia diagram prezentujący notację dla obiektów w języku UML.

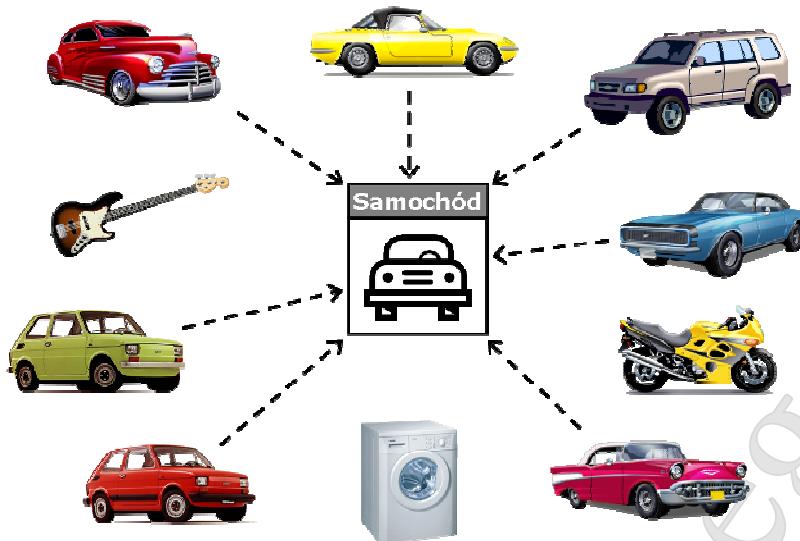


Rysunek 4.6: Notacja dla obiektów w języku UML

Podstawową reprezentacją obiektu jest prostokąt zawierający podkreśloną nazwę obiektu oraz oddzieloną dwukropkiem nazwę klasy obiektu (o klasach obiektów będzie mowa w następnym podrozdziale), czyli jego typ (ang. *type*). W taki podstawowy sposób reprezentowane są na powyższym diagramie obiekty Zenek i Bogdan, obydwa klasy Kierowca. Możliwa jest również prezentacja stanu obiektu. Listę właściwości obiektu umieszcza się poniżej jego nazwy. Aktualne wartości poszczególnych właściwości mogą być umieszczone po ich nazwie, oddzielone znakiem „=”. Obiekty na diagramie mogą być połączone łącznikami (ang. *link*), które odzwierciedlają powiązania pomiędzy obiektami. Powiązane obiekty mogą komunikować się między sobą poprzez wywoływanie swoich usług. Powiązania mogą być ukierunkowane, wskazując kierunek komunikacji pomiędzy obiektami. Na rysunku (Rysunek 4.6) widzimy, że obiekt „Bogdan” jest powiązany z obiektem „samochód Bogdana”. Kierunek powiązania wskazuje na to, że obiekt „Bogdan” może poprosić obiekt „samochód Bogdana” o wykonanie jakieś usługi. Odwrotna komunikacja nie jest możliwa. To, jakie usługi „samochód Bogdana” może wykonać, określa jego zachowanie. Zachowania obiektów nie przedstawia się na diagramach obiektów. Zachowanie jest takie samo dla wszystkich obiektów danej klasy i jest definiowane w klasie, co zostanie przedstawione w kolejnym podrozdziale.

4.4. Klasy obiektów

Modelowanie obiektowe pozwala odwzorowywać obiekty występujące w świecie rzeczywistym na obiekty w systemie oprogramowania, przez co łączy świat zamawiający ze światem programistów. Jednakże, w każdej dziedzinie problemu, dla którego tworzony jest system oprogramowania, występuje zwykle tysiące czy miliony obiektów. Wystarczy chociażby wyobrazić sobie system do obsługi ubezpieczeń komunikacyjnych, w którym występują, jako obiekty, wszystkie ubezpieczone samochody oraz ich właściciele. Jak wiemy, jednym ze sposobów radzenia sobie ze złożonością jest stosowanie zasady abstrakcji poprzez klasyfikację. Dlatego też, w modelowaniu obiektowym podstawowym elementem nie jest obiekt, lecz jego uogólnienie, czyli klasa. Klasa jest to opis grupy podobnych obiektów.



Rysunek 4.7: Klasa Samochód grupuje podobne do siebie obiekty

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 więc do innych klas. Możemy zatem sformułować następującą definicję:

Klasa (ang. *class*) to opis grupy obiektów, które mają taki sam zestaw właściwości oraz sposób zachowania.

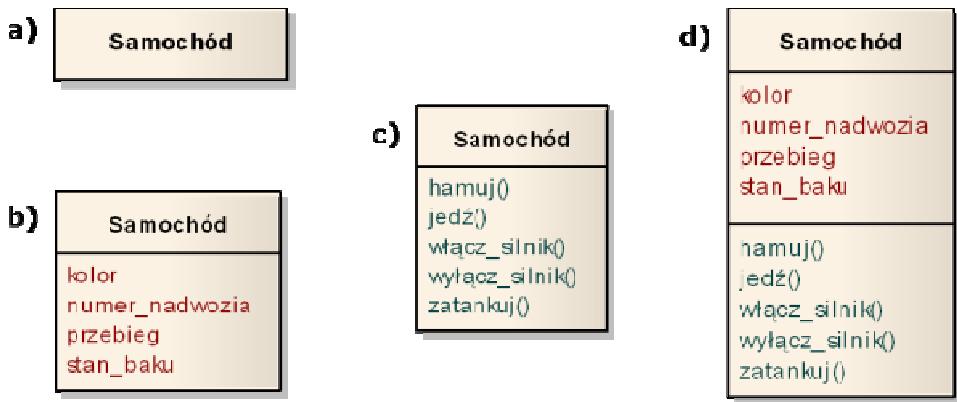
Każda klasa ma przypisaną nazwę, która wyróżnia ją spośród innych klas. Właściwości obiektów reprezentowanych przez klasę zwane są atrybutami. Klasa może mieć dowolną ilość atrybutów (ang. *attribute*), a nawet nie mieć ich wcale. Zachowanie, czyli usługi, które mogą wykonywać obiekty danej klasy nazywane są operacjami (ang. *operation*). Klasa może mieć dowolną ilość operacji, a nawet nie mieć ich wcale.

Rysunek 4.8 przedstawia notację dla klasy w języku UML. Ta sama klasa może być reprezentowana w różny sposób – na różnym poziomie szczegółowości. W najprostszym wariantie, klasa jest symbolizowana przez prostokąt z nazwą klasy (Rysunek 4.8a). Na ogół nazwę podaje się w formie rzeczownika lub wyrażenia rzeczownikowego, pochodzącego z modelowanej dziedziny. Każdy wyraz w nazwie zaczyna się zwykle wielką literą.

Możliwe jest także pokazanie atrybutów klasy. Umieszcza się je w sekcji oddzielonej poziomą kreską od jej nazwy klasy (Rysunek 4.8b). Można podać jedynie nazwy atrybutów. Można też podać typ wartości, jakie atrybut może przyjmować. Nazwę atrybutu podaje się zazwyczaj w formie rzeczownika lub wyrażenia opisującego właściwości obiektów danej klasy.

Możliwe jest pokazanie tylko operacji klasy, poprzez umieszczenie ich w sekcji symbolu klasy poniżej jej nazwy (Rysunek 4.8c). Można podać jedynie nazwy operacji. Można też dokładnie określić operacje poprzez podanie w nawiasach typu przekazywanych parametrów oraz typu wartości zwracanej.

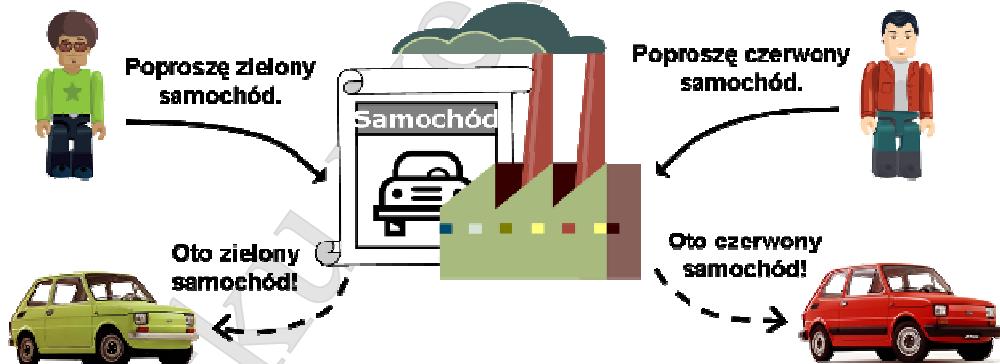
Jeśli chcemy pokazać jednocześnie atrybuty i operacje klasy, to atrybuty umieszczamy w sekcji poniżej nazwy klasy, natomiast operacje w sekcji poniżej atrybutów (Rysunek 4.8d). Symbol klasy nie musi zawierać wszystkich atrybutów i operacji.



Rysunek 4.8: Notacja dla klasy w języku UML

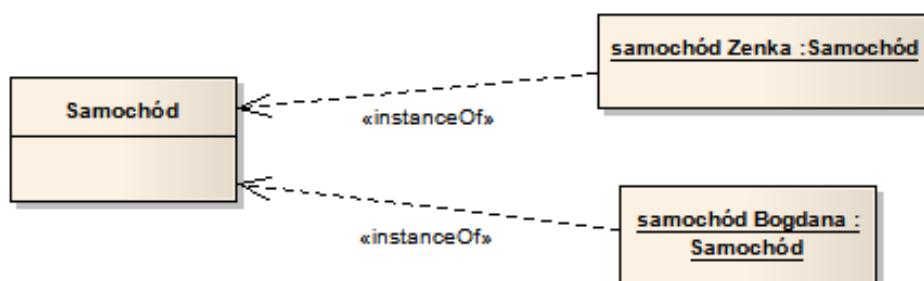
Sposób prezentacji klasy na diagramie zależy od celu, w jakim sporządzamy diagram oraz od osoby, która będzie ten diagram czytać. Jeśli diagram ma na celu poglądowe przedstawienie struktury pojęć modelowanej dziedziny, wystarczy, że klasy będą przedstawione w najprostszej postaci. W przypadku diagramów, które mają posłużyć programistom do implementacji systemu, przedstawione na nich klasy powinny zawierać wszystkie szczegóły. Diagramy klas będą dokładniej omówione w kolejnym rozdziale.

Klasę można obrazowo przedstawić jako fabrykę obiektów (Rysunek 4.9). Fabryka posiada plany konstrukcyjne, które szczegółowo określają cechy produkowanych w niej 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 określane są przez fabrykę, np. przebieg czy stan baku. Stan początkowy samochodu będzie się oczywiście zmieniał w miarę jego użytkowania przez właściciela.



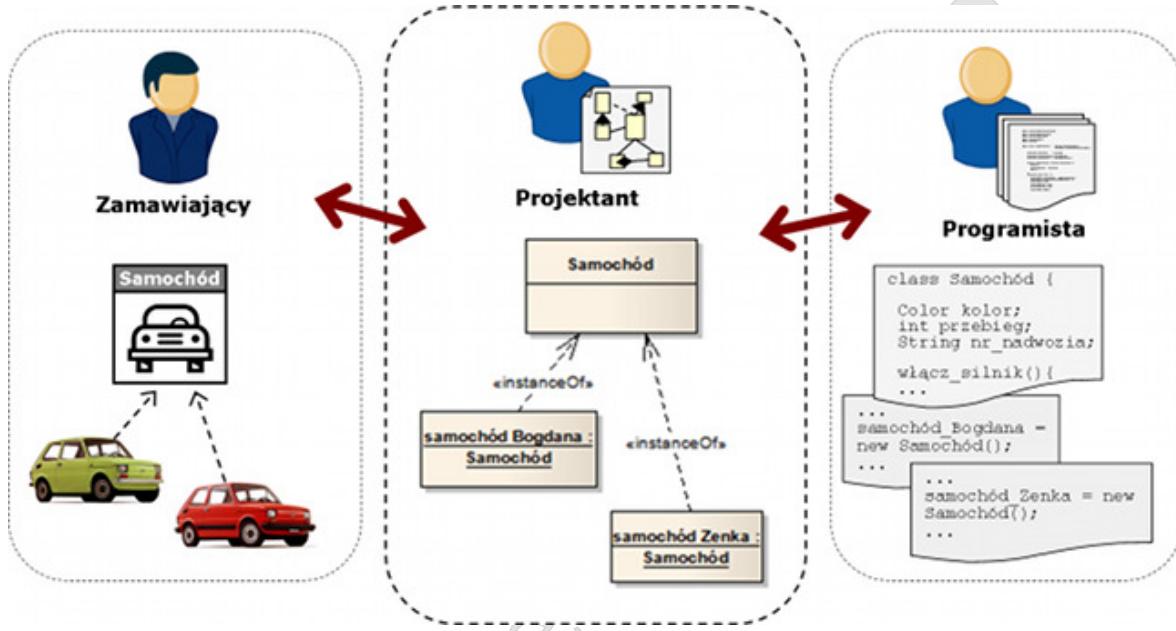
Rysunek 4.9: Klasa jako „fabryka obiektów”

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 i relacje «*instanceOf*» łączące obiekty z klasą.



Rysunek 4.10: Obiekty jako instancje klas

Powiedzieliśmy wcześniej, że obiekty są postrzegane przez różnych ludzi z różnych perspektyw zależnych od tego, jaka rolę dana osoba pełni w procesie tworzenia systemu oprogramowania. Mamy też możliwość prezentowania klas obiektów na różnych poziomach szczegółowości mając na uwadze cel 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).



Rysunek 4.11: Modele oparte o klasy obiektów stanowią wspólny język dla zamawiających projektantów i programistów

Warto zwrócić uwagę na pewien istotny aspekt tworzenia modeli obiektowych, który ma wpływ na ich interpretację. Możemy wyróżnić trzy perspektywy, których można użyć przy tworzeniu modeli, zwłaszcza modeli klas [Fowl02].

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.

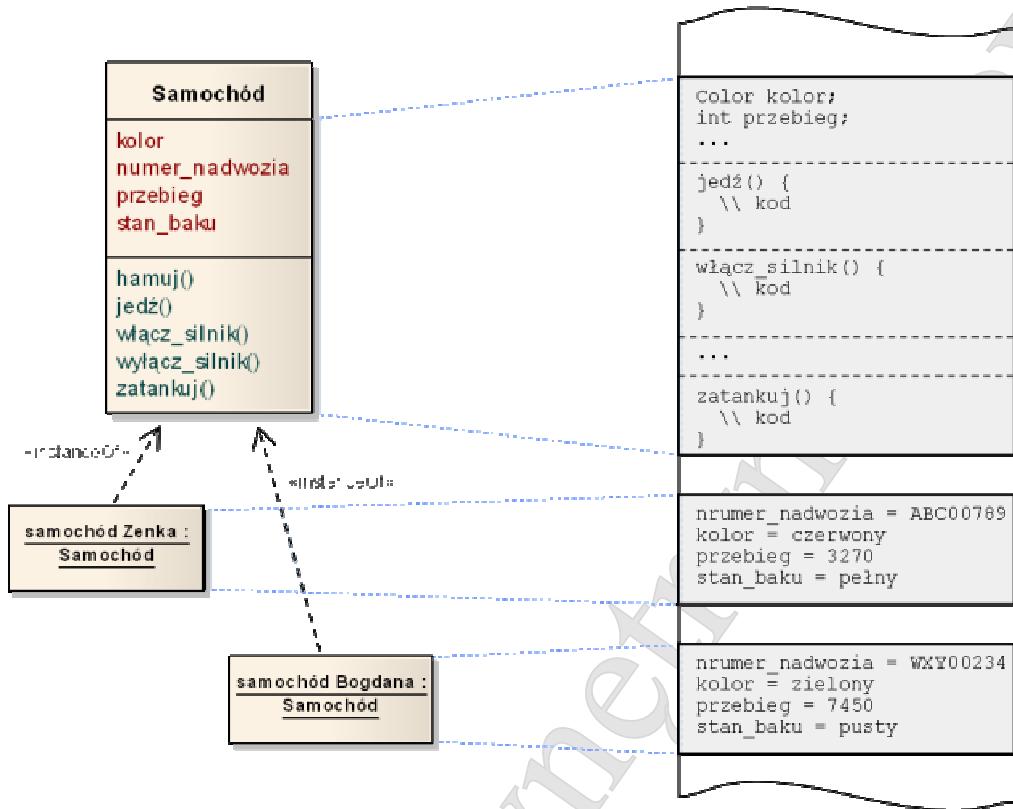
Perspektywa specyfikacyjna. Perspektywa ta dotyczy bezpośrednio oprogramowania, ale raczej w sensie definiowania typów obiektów niż szczegółów implementacji. Typ może być implementowany przez wiele klas, a klasa może implementować wiele typów. W perspektywie tej unika się zazwyczaj szczegółów związanych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

W działającym oprogramowaniu, klasa oraz jej obiekty przechowywane są w pamięci komputera (Rysunek 4.12). Jak już wiemy, klasa definiuje atrybuty opisujące właściwości wszystkich jej obiektów oraz operacje opisujące sposób zachowania się obiektów. Ponieważ kod operacji jest wspólny dla wszystkich obiektów danej klasy, jest on umieszczony w jednym miejscu w pamięci. Konkretnie obiekty klasy odwołują się do tego kodu podczas wywoływania na nich operacji przez inne obiekty. Każdy obiekt ma ponadto swoje własne miejsce w pamięci operacyjnej, dzięki czemu ma on swoją

indywidualna tożsamość oraz stan. W pamięci przeznaczonej dla obiektu przechowywane są aktualne wartości każdego atrybutu. Typ wartości atrybutów jest zdefiniowany w klasie obiektów. Rozmiar pamięci zajmowanej przez każdy z obiektów danej klasy jest jednakowy.



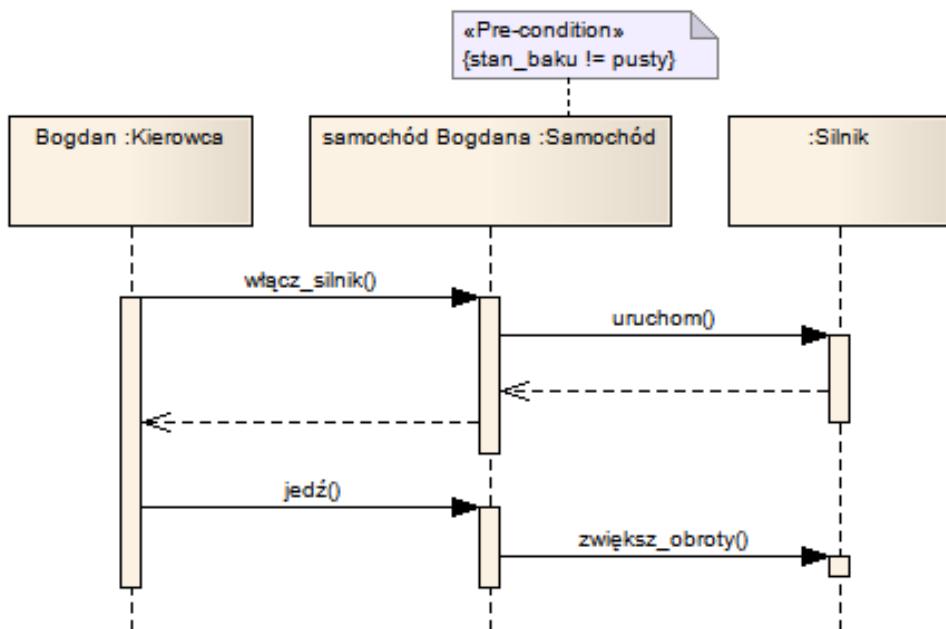
Rysunek 4.12: Operacje klasy i atrybuty obiektów klasy umieszczone są w pamięci

4.5. Modelowanie systemu jako układu współpracujących ze sobą obiektów

Przy pomocy klas i obiektów modelujemy statyczną strukturę systemu. Obiekty w systemie nie są jednak bezczynne, lecz wzajemnie na siebie oddziałują. Działanie każdego systemu oprogramowania polega na wzajemnym przesyłaniu komunikatów w ramach pewnego zbioru obiektów w ścisłe określonym celu. Jak wiemy, obiekty posiadają zachowanie określone przez klasę obiektów w postaci zbioru usług. Usługi te mogą być wykonywane na prośbę innych obiektów. Prośba o wykonanie usługi nazywa się komunikatem (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. Po wykonaniu usługi, obiekt może poinformować nadawcę komunikatu o zakończeniu całej operacji. Ewentualnie, może zwrócić nadawcy komunikatu jakąś wartość lub obiekt będące rezultatem wykonania usługi. Wpływ na to, jak zostanie wykonana określona usługa mają trzy rzeczy.

- 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.
- Stanu innych obiektów, z usług których korzysta obiekt w celu wykonania swojej usługi. W celu wykonania usługi „`włącz_silnik`”, obiekt klasy „`Samochód`” wysyła do obiektu klasy „`Silnik`” komunikat „`uruchom`”. Gdy stan silnika nie pozwala na wykonanie tej usługi, również usługa „`włącz_silnik`” nie przyniesie spodziewanego efektu.

Komunikację pomiędzy obiektami możemy w języku UML modelować w postaci tzw. Diagramów sekwencji (ang. *sequence diagram*). Na rysunku (Rysunek 4.13) widzimy diagram sekwencji pokazujący wymianę komunikatów pomiędzy obiektami w celu wykonania usługi „*włącz_silnik*” oraz „*jedź*”. Pionowe linie umieszczone pod obiektami są to tzw. linie życia (ang. *lifeline*), które pozwalają ukazać wymianę komunikatów w czasie. Komunikaty wykonywane są w kolejności od góry do dołu. Komunikat wywołujący usługę symbolizowany jest przez poziomą strzałkę od nadawcy do wykonawcy komunikatu. Na diagramie widzimy, że obiekt klasy „*Kierowca*” prosi obiekt klasy „*Samochód*” o włączenie silnika. W tym momencie zaczyna się wykonywanie usługi, które zaznaczone jest jako wąski prostokąt umieszczony na linii życia obiektu wykonującego usługę. Aby usługa przyniosła spodziewany efekt, stan baku nie może być pusty, stąd przyczepiony do obiektu warunek wstępny (ang. *pre-condition*) wykonania tej usługi. Aby zrealizować usługę, samochód przesyła do obiektu klasy „*Silnik*” komunikat „*uruchom*”. Gdy silnik wykona swoją usługę, tzn. uruchomi się, wysyła komunikat zwrotny do nadawcy, oznaczony na diagramie przerywaną linią. Samochód, wiedząc, że silnik jest już uruchomiony, może np. dokonać zmiany swojego stanu dla zasygnalizowania gotowości do jazdy. Po zakończeniu wszelkiego przetwarzania w ramach usługi, samochód informuje o tym kierowcę wysyłając mu komunikat zwrotny. Kierowca może wtedy poprosić samochód o wykonanie kolejnej usługi, np. „*jedź*”.



Rysunek 4.13: Prezentacja współpracy pomiędzy obiektami w postaci diagramu sekwencji

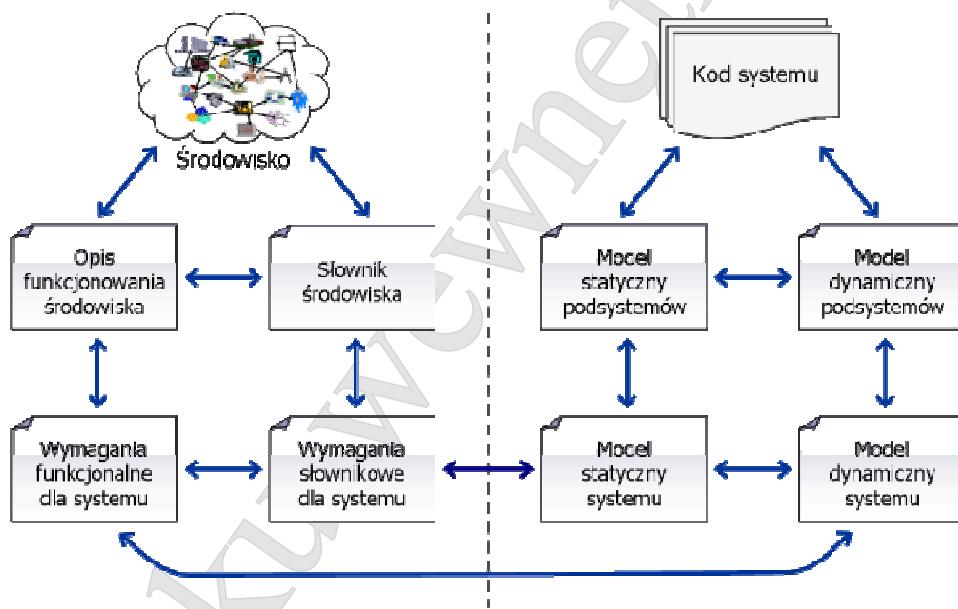
Zauważmy, że kierowca chcąc uruchomić silnik, wysyła jedynie odpowiedni komunikat samochodowi i oczekuje na rezultat nie wiedząc, w jaki sposób samochód realizuje usługę i jakie inne obiekty są w to zaangażowane. Samochód jest dla niego czarną skrzynką z przyciskami do wywoływania określonych usług (Rysunek 4.14). Dobrze zaprojektowany obiekt udostępnia innym obiektom tylko te usługi lub właściwości, które są im potrzebne. Ważne jest, 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 4.14: Obiekt jako czarna skrzynka z przyciskami

4.6. Modele na ścieżce od opisu środowiska do kodu

Jak już wspominaliśmy w tym rozdziale, aby połączyć świat zamawiających oraz programistów, musimy zbudować szereg modeli obiektowych, które opisują różne aspekty budowanego systemu oraz jego środowiska, na różnym poziomie szczegółowości. Rysunek 4.15 przedstawia taki szereg modeli.



Rysunek 4.15: Modele na ścieżce od opisu środowiska do kodu

Po lewej stronie znajdują się modele będące produktami analizy środowiska. Po prawej znajdują się modele prowadzące do powstania kodu, czyli produkty syntezy. 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 system.

Dwa modele wynikające bezpośrednio z analizy środowiska opisujące zarówno jego dynamikę jak i strukturę, możemy zaliczyć do perspektywy pojęciowej. Mogą one być przekształcone w modele wymagań, które są podstawowymi składnikami specyfikacji wymagań, która określa, co budowany system ma robić. Na tej podstawie, budowane są modele systemu, stanowiące jego projekt architektoniczny. Projekt taki określa, z jakich elementów zbudować system, aby najlepiej realizował postawione wymagania. 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.

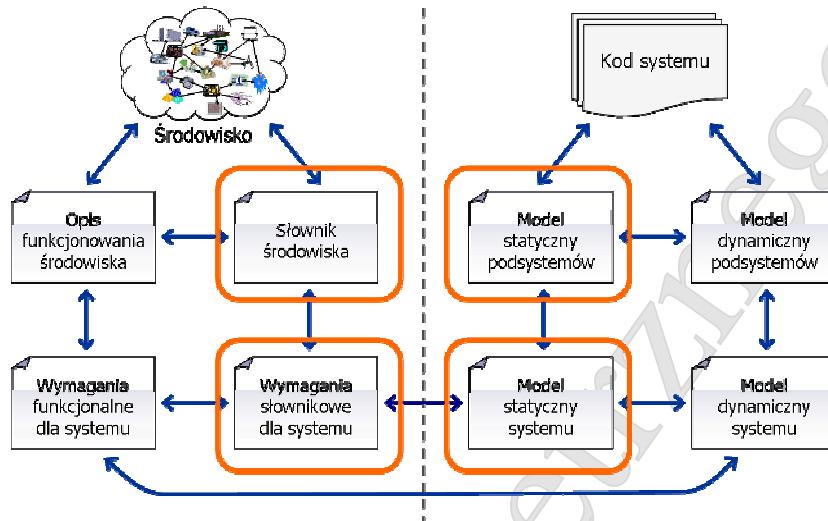
W kolejnych dwóch rozdziałach opiszemy jak modelować aspekty statyczne i dynamiczne systemu, na ścieżce od środowiska do kodu.

4.7. Podsumowanie

Modelowanie obiektowe wspomaga nas w zrozumieniu złożonych systemów oprogramowania. Dzięki modelom jesteśmy w stanie lepiej zrozumieć strukturę i działanie systemu. Modele wizualne pozwalają na stworzenie swego rodzaju mapy ułatwiającej poruszanie się po wymaganiach na system i jego implementacji.

5. Modelowanie struktury systemu w języku UML

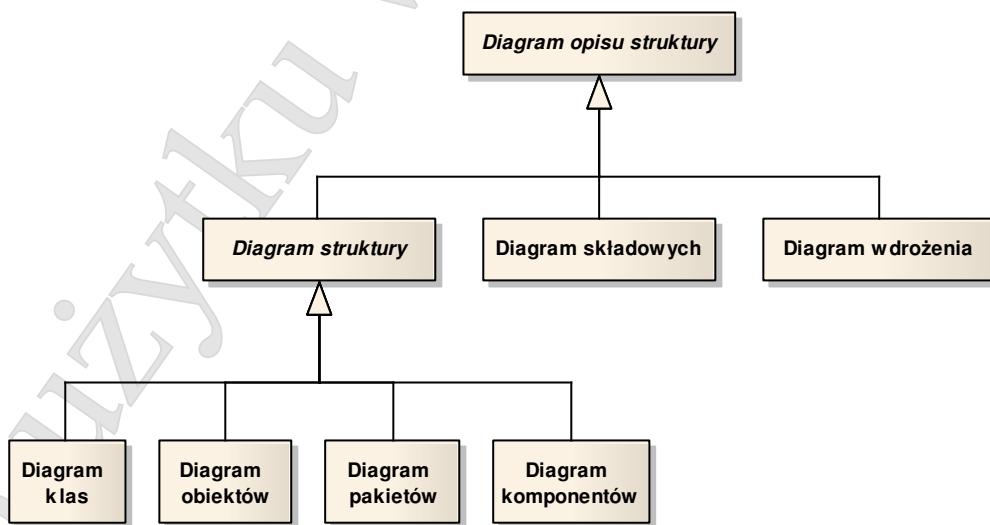
W niniejszym rozdziale przedstawimy diagramy, jakie oferuje język UML do opisywania struktury systemu oprogramowania oraz środowiska. Modele na ścieżce od środowiska do kodu, które możemy tworzyć przy pomocy diagramów opisu struktury zostały wyróżnione na rysunku (Rysunek 5.1).



Rysunek 5.1: Modele strukturalne na ścieżce od środowiska do kodu

5.1. Przegląd diagramów opisu struktury w języku UML

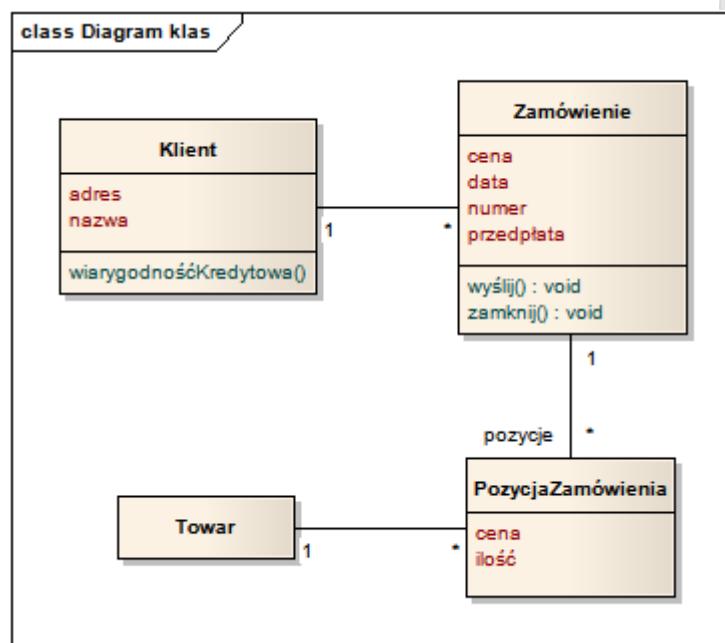
Rysunek 5.2 przedstawia klasyfikację diagramów języka UML, służących do opisu struktury. Istnieją trzy podstawowe rodzaje diagramów opisu struktury: diagram struktury, diagram składowych oraz diagram wdrożenia. Zauważmy, że diagram struktury nie jest żadnym konkretnym typem diagramu. Jest on generalizacją czterech konkretnych diagramów: diagramu klas, diagramu obiektów, diagramu pakietów oraz diagramu komponentów.



Rysunek 5.2: Klasyfikacja diagramów opisu struktury w języku UML

5.1.1. Diagram klas

Rysunek 5.3 przedstawia przykładowy diagram klas. Na diagramach klas umieszczamy przed wszystkim klasy oraz związki (ang. *relationship*) między klasami. Najczęszszym rodzajem związków są asocjacje (ang. *association*). Asocjacje mogą oznaczać np. powiązanie znaczeniowe pomiędzy pojęciami reprezentowanymi przez klasy. Mogą też oznaczać, że istnieje komunikacja pomiędzy obiektami powiązanych klas. Asocjacje przedstawia się na diagramie, jako linie łączące dwie klasy. Asocjacje mogą mieć krotności (ang. *multiplicity*) oraz role (ang. *role*), które wskazują ile obiektów danej klasy i w jakiej roli, może brać udział w danym powiązaniu. Inne typy związków między klasami, jakie możemy prezentować na diagramach klas, to: generalizacja (ang. *generalization*), agregacja (ang. *aggregation*), kompozycja (ang. *composition*). Same klasy mogą być prezentowane na diagramach na różnym poziomie szczegółowości – atrybuty i operacje mogą być widoczne bądź ukryte.

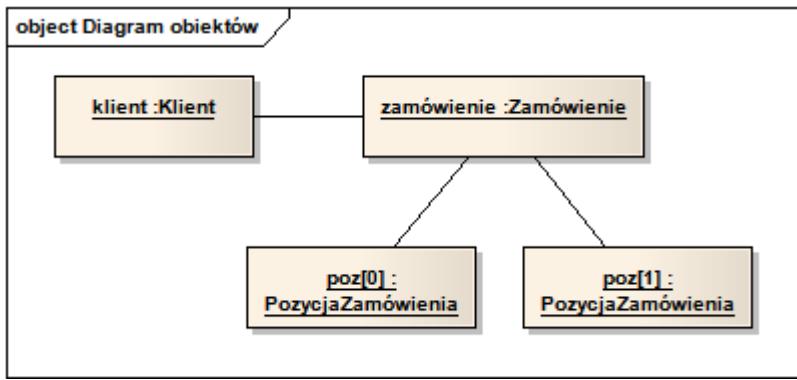


Rysunek 5.3: Przykładowy diagram klas

Bardziej szczegółowy opis modelu klas znajduje się w dalszej części tego rozdziału.

5.1.2. Diagram obiektów

Klasy obiektów wraz z powiązaniemi i krotnościami prezentują ogólnie wszystkie możliwe konfiguracje swoich instancji, czyli obiektów, w działającym systemie. Czasem zachodzi jednak potrzeba pokazania tylko wybranej sytuacji, w której obiekty i ich powiązania przyjmują konkretną konfigurację. Taką sytuację możemy zobrazować na diagramie obiektów, który można porównać do zdjęcia systemu wykonanego w określonej chwili. Na diagramie obiektów pokazujemy pojedyncze obiekty będące instancjami określonych klas oraz łączniki, będące instancjami asocjacji pomiędzy tymi klasami. Na diagramie obiektów możemy umieścić wiele instancji tej samej klasy a asocjacja między dwoma klasami może się przekładać na wiele łączników między instancjami tych klas.

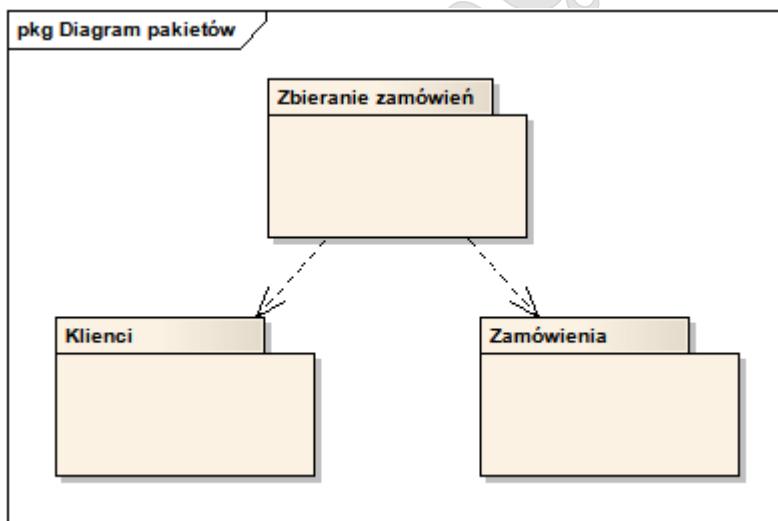


Rysunek 5.4: Przykładowy diagram obiektów

Obiekt na diagramie zawiera podkreślona nazwę oraz, oddzieloną dwukropkiem nazwę klasy, której jest instancją. Łączniki, podobnie jak asocjacje, zaznaczane są jako linia łącząca dwa obiekty. Diagram obiektów na powyższym rysunku (Rysunek 5.4) odpowiada diagramowi klas na rysunku poprzednim (Rysunek 5.3).

5.1.3. Diagram pakietów

Pakietы (ang. package) służą do hierarchicznego organizowania elementów modelu w celu zapewnienia przejrzystości i czytelności. Pakiet stanowi zbiór klas, asocjacji lub innych elementów modelu; może też zawierać inne pakiety. Pakietы stanowią także unikalne przestrzenie nazw, dzięki czemu nie jest konieczna unikalność nazw klas czy innych elementów w ramach całego modelu, a jedynie w ramach pakietów.



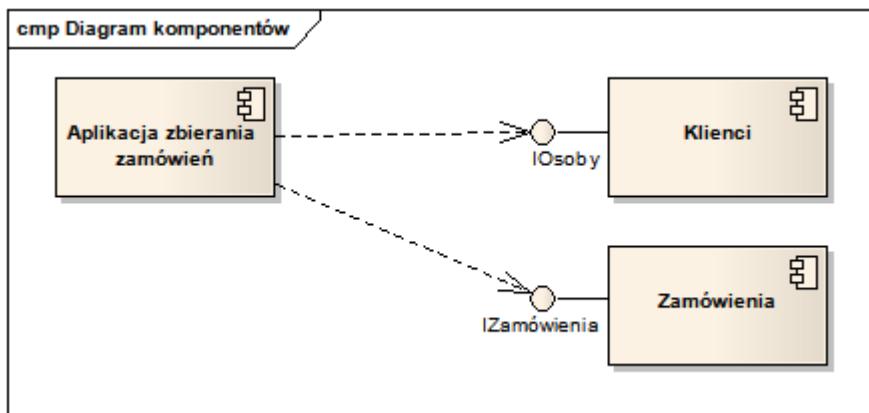
Rysunek 5.5: Przykładowy diagram pakietów

Diagramy pakietów pokazują sposób grupowania elementów modelu w pakiety (Rysunek 5.5). Diagramy te ukazują pakiety oraz zależności między nimi. Sposób prezentacji pakietów widać na przykładowym diagramie. Zależności między pakietami zaznaczamy strzałkami z przerywaną linią. Strzałka skierowana jest do pakietu zależnego w stosunku do pakietu, z którego ta strzałka wychodzi. Możliwe jest nadawanie zależnościom dodatkowego znaczenia poprzez stosowanie stereotypów. W przypadku zależności pakietów, najczęściej stosowanym stereotypem jest «import». Oznacza to, że pakiet jest importowany do innego pakietu, czyli że można korzystać z elementów importowanego pakietu w pakiecie zależnym.

5.1.4. Diagram komponentów

Komponent (ang. component) jest elementem, który oprócz grupowania elementów modelu, dostarcza również pewnej funkcjonalności innym elementom. Komponent stanowi podsystem całego systemu. Komunikacja pomiędzy komponentami odbywa się poprzez interfejsy (ang. interface).

Interfejs jest elementem modelu, który dostarcza zestawu operacji. Interfejs jest zwykle powiązany z innymi elementami (klasami, komponentami) i określa zewnętrzny sposób zachowania się tych elementów. Każda klasa lub komponent mogą realizować jeden lub więcej interfejsów. Jeden interfejs, z kolei, może być realizowany przez wiele klas czy komponentów.



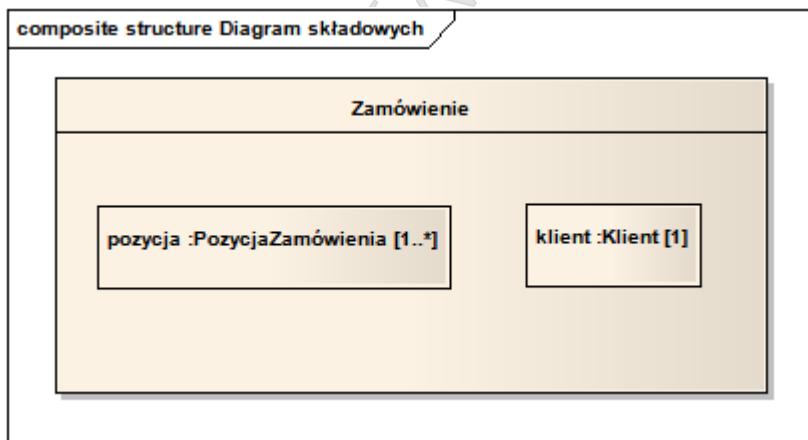
Rysunek 5.6: Przykładowy diagram komponentów

Komponenty, ich interfejsy oraz zależności pomiędzy nimi umieszczamy na diagramach komponentów. Przykład takiego diagramu przedstawia rysunek (Rysunek 5.6).

Bardziej szczegółowy opis modelu komponentów znajduje się w dalszej części tego rozdziału.

5.1.5. Diagram składowych

Żaden z diagramów struktury nie dawał możliwości pokazania zawartości pojedynczych elementów modelu bezpośrednio na diagramie. Możliwość umieszczania elementów wewnętrznych elementów, dają diagramy składowych oraz diagramy wdrożenia. Umożliwiają one pokazanie elementów składowych klas, komponentów lub fizycznych elementów systemu komputerowego.

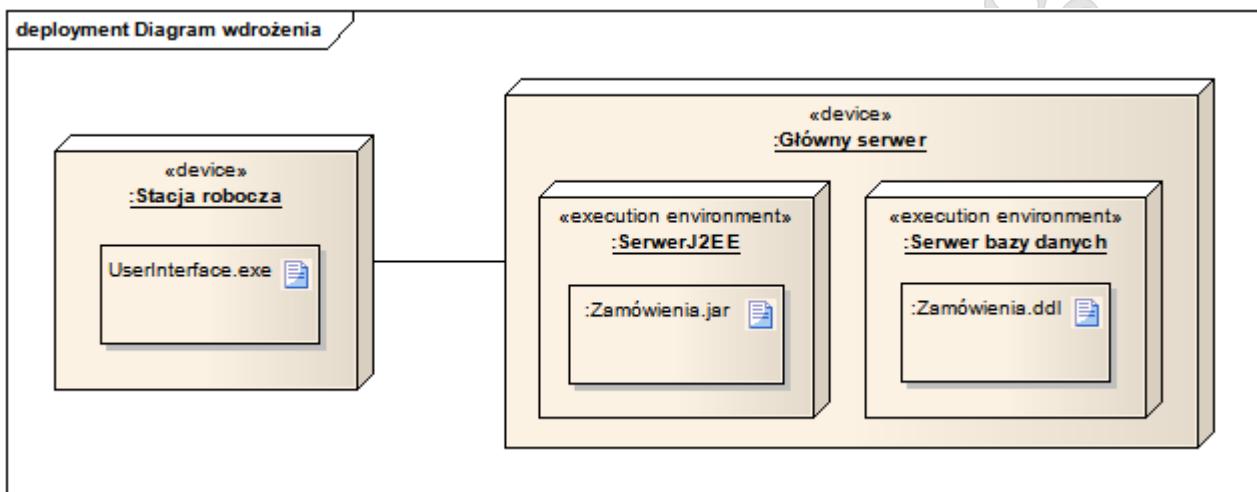


Rysunek 5.7: Przykładowy diagram składowych

Rysunek 5.7 przedstawia przykład diagramu składowych. Na diagramie takim, można zaprezentować klasę lub komponent wraz z obiektami, które tworzą ich wewnętrzną strukturę. Obiekty tworzące wewnętrzną strukturę są na diagramie składowych reprezentowane przez składniki (ang. *part*). Są one własnościami klasy lub komponentu i połączone są powiązaniemi (ang. *connector*). Składniki mogą odpowiadać atrybutom, ale dodatkowo możliwe jest pokazanie ich wzajemnych zależności w ramach klasy lub komponentu. Składniki wraz z powiązaniemi rysujemy wewnątrz piktogramu klasy bądź komponentu. Składniki zawierają nazwę oraz oddzielony dwukropkiem typ składnika. Składnikom można przypisać krotność, która pokazuje ile obiektów danej klasy zawartych jest w całości. Krotności umieszczamy w nawiasach kwadratowych po nazwie składnika.

5.1.6. Diagram wdrożenia

Diagramy wdrożenia służą przede wszystkim do pokazania fizycznej struktury systemu. Przykład takiego diagramu znajduje się na rysunku (Rysunek 5.8). Na diagramie wdrożenia umieszczamy głównie węzły (ang. *node*), połączenia między nimi a także artefakty (ang. *artifact*). Węzły odpowiadają fizycznym elementom systemu komputerowego. Węzeł może reprezentować np. serwer fizyczny, serwer w postaci oprogramowania oferującego usługi innym programom bądź środowisko uruchomieniowe dla programów. Wewnątrz węzłów umieszczane są odpowiednie artefakty. Artefakty, w kontekście języka UML, są to fizyczne elementy informacyjne używane lub wyprodukowane w procesie wytwarzania oprogramowania. Artefaktami są np. pliki z kodem źródłowym, pliki wykonywalne, schematy baz danych, modele czy też dokumenty tekstowe bądź inne. Artefakty umieszczamy na diagramie w tych węzłach, w których są lub będą fizycznie zainstalowane w działającym systemie.



Rysunek 5.8: Przykładowy diagram wdrożenia

Bardziej szczegółowy opis modelu wdrożenia znajduje się w dalszej części tego rozdziału.

5.2. Modelowanie struktury logicznej systemu

W modelu struktury systemu oprogramowania możemy wyodrębnić dwa poziomy specyfikacji: strukturę logiczną oraz strukturę fizyczną. Podczas modelowania struktury logicznej wyodrębniamy logiczne jednostki systemu, takie jak klasy, pakiety, komponenty. Elementy te są abstrakcją dziedziny systemu na różnym poziomie. Współpracując ze sobą realizują one wymaganą od systemu funkcjonalność. W niniejszym podrozdziale opisujemy dwa najważniejsze modele oferowane przez język UML, służące do specyfikowania struktury logicznej systemu: model klas oraz model komponentów.

5.2.1. Model klas

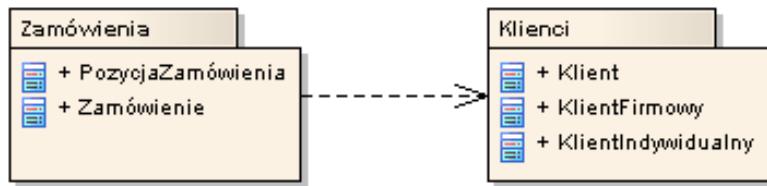
Wraz z rozwojem technik obiektowych, klasy stały się podstawowymi elementami konstrukcyjnymi systemów oprogramowania. Modele klas tworzone są na każdym etapie procesu wytwarzania oprogramowania – począwszy od etapu analizy, kiedy modelowane są pojęcia związane z dziedziną problemu, poprzez etapy projektowania oraz implementacji, podczas których tworzone są bardziej szczegółowe modele klas, uwzględniające aspekty konstrukcyjne i implementacyjne oprogramowania.

Poniżej prezentujemy elementy języka UML służące do tworzenia modeli klas. Podstawowa notacja dla klasy została omówiona w poprzednim rozdziale. W tym rozdziale przedstawiamy pozostałe elementy modelu klas. Opisujemy zarówno ich semantykę, jak również ich graficzną reprezentację służącą do rysowania diagramów klas. Należy w tym miejscu wyjaśnić różnicę pomiędzy modelem a diagramem. Model klas jest to kompletny opis wszystkich typów obiektów w systemie lub pojęć analizowanej dziedziny, wraz z wszystkimi statycznymi powiązaniemi między nimi. Diagram klas,

natomast, jest graficzną prezentacją wybranego fragmentu modelu, sporządzanym w celu łatwiejszego zrozumienia wybranych aspektów modelu. Może istnieć dowolna ilość diagramów dla danego modelu, na różnych poziomach szczegółowości. Każdy element modelu może występować na wielu diagramach w różnym kontekście. W przypadku bardzo prostego modelu, może być on zobrazowany na jednym diagramie.

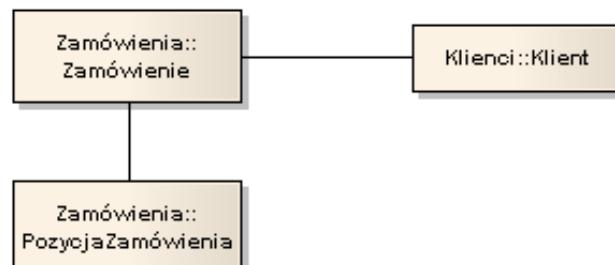
Przestrzenie nazw klas

Jak już wiemy, pakiety służą do grupowania elementów modelu w hierarchiczne struktury. W przypadku modelu klas, gdzie może występować bardzo duża ilość elementów, stworzenie odpowiedniej struktury pakietów może okazać się niezbędne. Modelując dziedzinę problemu, za pomocą pakietów grupujemy logicznie powiązane ze sobą pojęcia (Rysunek 5.9). Możemy też oddzielić klasy modelujące aspekty techniczne oprogramowania (np. okna interfejsu użytkownika) od klas modelujących pojęcia dziedziny.



Rysunek 5.9: Pakiety jako elementy grupujące klasy

Pakiety stanowią przestrzeń nazw dla klas. W modelu nie mogą istnieć dwie klasy o takiej samej nazwie, chyba, że należą do innych przestrzeni nazw. Na diagramie, nazwa klasy może być poprzedzona nazwą pakietu, z którego klasa pochodzi. Nazwę klasy oddziela się znakiem „::” od nazwy pakietu (Rysunek 5.10).



Rysunek 5.10: Przestrzenie nazw klas

W przypadku pakietów zagnieżdżonych, nazwę pakietu podlegającego poprzedza się nazwą pakietu nadzawanego, oddzielając znakiem „::”, np. pakiet1::pakiet2::pakiet3::NazwaKlasy. Takie nazwy klas określa się mianem nazw kwalifikowanych. Ich stosowanie ma szczególne znaczenie wtedy, gdy na diagramie umieszcza się klasy o takiej samej nazwie, lecz pochodzące z innych przestrzeni nazw.

Nazwa samej klasy może być tekstem zawierającym dowolną ilość liter, cyfr lub innych znaków (oprócz dwukropka). W praktyce jednak, najpowszechniejsza jest konwencja nazewnicza, w której nazwę klasy stanowi rzeczownik lub wyrażenie rzeczownikowe, w którym każdy wyraz zaczyna się wielką literą a wyrazy nie są oddzielone spacią, np. „Zamówienie” lub „PozycjaZamówienia”.

Atrybuty i operacje

Z poprzedniego rozdziału wiemy, że klasę można prezentować na różnym poziomie szczegółowości. Czasami wystarczy zaprezentować klasę w postaci prostokąta z nazwą, w większości przypadków jednak, umieszcza się także atrybuty i operacje klasy. Nie trzeba prezentować wszystkich atrybutów i operacji, jakie klasa posiada. Wystarczy pokazać tylko te, które niosą istotną informację na danym diagramie.

W zależności od szczegółowości diagramu notacja dla atrybutów i operacji może uwzględniać lub pomijać pewne elementy. Tworząc model klas na wysokim poziomie abstrakcji, zazwyczaj wystarczy podać jedynie nazwy atrybutów i operacji. Nazwa atrybutu lub operacji może być dowolnym tekstem. Najpowszechniejszą praktyką jest jednak podawanie nazwy w postaci rzeczownika lub wyrażenia opisującego określoną właściwość lub określone zachowanie danej klasy.

Jedną z najważniejszych cech – poza nazwą – jaką można określić dla składników klasy jest widoczność. Widoczność określa czy dany atrybut lub operacja jest dostępna dla pozostałych klas. UML definiuje cztery poziomy widoczności składników klas.

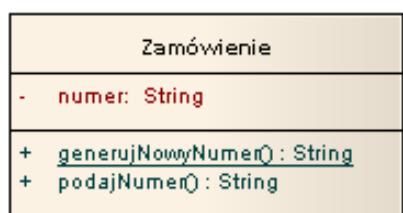
- *Public* (publiczny). Każda zewnętrzna klasa, która ma dostęp do klasy zawierającej składniki publiczne, ma również dostęp do tych składników. Na diagramie składowe publiczne oznacz się znakiem „+”. Jeśli składnik nie ma określonej widoczności, domyślnie przyjmuje się, że jest to składnik publiczny.
- *Private* (prywatny). Dostęp do składników prywatnych ma jedynie klasa zawierająca te składniki. Na diagramie składowe prywatne oznacza się znakiem „-”.
- *Protected* (zabezpieczony). Dostęp do składników chronionych ma klasa zawierająca te składniki oraz wszystkie klasy dziedziczące z niej (podklasy). Na diagramie składniki chronione oznaczamy znakiem „#”.
- *Package* (pakietowy). Dostęp do składników o takiej widoczności ma każda klasa znajdująca się w tym samym pakiecie, co klasa zawierająca te składniki. Na diagramie oznaczamy takie składniki znakiem „~”.

Rysunek 5.11 przedstawia przykład klasy zawierającej atrybuty i operacje o różnej widoczności. Poziomy widoczności zdefiniowane w języku UML odpowiadają poziomom widoczności zdefiniowanym w większości obiektowych języków programowania, jak np. Java czy C++.



Rysunek 5.11: Widoczność atrybutów i operacji klasy

Kolejną cechą atrybutów i operacji jest ich zasięg. Normalnie, atrybuty i operacje klasy występują w każdym obiekcie danej klasy określając ich właściwości oraz zachowanie. O takich atrybutach i operacjach mówimy, że mają zasięg instancji. Dla niektórych atrybutów i operacji klasy, możemy określić, że ich zasięg ogranicza się jedynie do tej klasy. Składniki o takim zasięgu określamy mianem statycznych. Oznacza to, że istnieje jeden egzemplarz takiego składnika, wspólny dla wszystkich instancji klasy. Na diagramie atrybuty i operacje statyczne oznacza się poprzez ich podkreślenie. Rysunek 5.12 przedstawia klasę zawierającą operację statyczną. Operacja statyczna generujNowyNumer() zawsze zwraca unikalny numer zamówienia, który może być przypisany jako wartość atrybutu numer w nowo tworzonych obiektach klasy Zamówienie. Numer ten może być odczytany poprzez wywołanie na obiekcie operacji podajNumer(), która ma zasięg instancji.



Rysunek 5.12: Operacja statyczna

Oprócz widoczności oraz zasięgu atrybutu, można także określić jego liczebność, typ oraz wartość początkową.

Liczebność atrybutu określa ilość egzemplarzy danego atrybutu, jaką może zawierać każdy obiekt klasy posiadającej ten atrybut. Liczebność definiuje się podając w nawiasach kwadratowych liczbę lub zakres.

Typ atrybutu może być jednym z typów pierwotnych, jak np. int (liczby całkowite) lub boolean (wartość logiczna). Typem atrybutu może być inna klasa, co oznacza, że jego wartością jest obiekt danej klasy.

Wartość początkowa określa wartość, jaką atrybut przyjmuje zaraz po utworzeniu obiektu danej klasy. Wartość ta musi być zgodna z typem atrybutu.

Atrybuty możemy więc określać w następujący sposób:

```
[widoczność] nazwa [[liczebność]] [: typ] [= wartość początkowa]
```

Elementy w nawiasach ostrokatnych możemy umieszczać na diagramach opcjonalne. Rysunek 5.11 zawiera przykłady różnych atrybutów.

Oprócz widoczności oraz zasięgu metody, można także określić jej listę parametrów oraz typ wyniku. Nazwa operacji wraz z parametrami oraz typem wyniku nazywamy sygnaturą. Dana klasa może zawierać operacje o takiej samej nazwie, jednak ich sygnatury muszą być różne.

Operacje klasy możemy deklarować w następujący sposób:

```
[widoczność] nazwa [(lista parametrów)] [: typ wyniku]
```

Parametry są wartościami określonych typów, przekazywanymi operacji podczas jej wywoływania. Mogą one np. stanowić dane wejściowe dla operacji, sterować sposobem jej wykonania, przekazywać wartości potrzebne do zmiany stanu obiektu, przekazywać wyniki wykonania operacji do obiektu wywołującego, itp. Lista parametrów może zawierać dowolną ilość parametrów oddzielonych przecinkami. Deklaracja każdego z parametrów jest następująca:

```
[tryb] nazwa : typ [= wartość domyślna]
```

Tryb określa, w jaki sposób operacja wykorzystuje dany parametr i może przyjmować jedną z trzech wartości:

- in – parametr jest parametrem wejściowym i nie może być modyfikowany,
- out – parametr jest parametrem wyjściowym i może być zmodyfikowany w celu przekazania informacji obiekowi wywołującemu,
- inout – parametr jest parametrem wejściowym, ale może być zmodyfikowany.

Jeżeli tryb nie jest określony dla parametru, to domyślnie przyjmuje się tryb „in”.

Typ parametrów określa się w taki sam sposób jak typ atrybutów. Wartość domyślna, natomiast, określa wartość, jaką przyjmuje parametr w przypadku, kiedy wartość parametru nie zostanie podana podczas wywołania operacji.

Jeśli operacja w wyniku wykonania zwraca jakąś wartość, to typ zwracanej wartości jest określany poprzez typ wyniku. Jeśli operacja nie zwraca żadnej wartości, typ wyniku możemy określić jako „void”. Rysunek 5.11 przedstawia przykładowe deklaracje operacji.

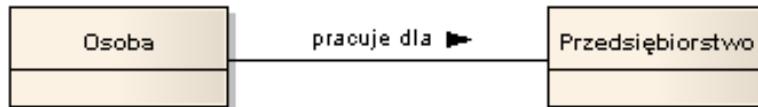
Należy tutaj wprowadzić rozróżnienie pomiędzy operacją a metodą. Operacja to specyfikacja usługi, która może być wykonana przez obiekt danej klasy. Metoda, natomiast, to implementacja operacji, czyli określenie sposobu jej wykonania.

Asocjacje

Podstawowymi blokami konstrukcyjnymi w języku UML, służącymi do łączenia elementów modelu są związki. Związki są bardzo istotnym elementem modelu klas. Najczęstszym rodzajem związków w modelach klas są asocjacje.

Asocjacje reprezentują związki strukturalne pomiędzy instancjami klas. Najogólniej, asocjacja pomiędzy dwiema klasami oznacza, że możliwe jest przejście od obiektu jednej klasy do obiektu drugiej klasy. Z punktu widzenia perspektywy pojęciowej asocjacje odzwierciedlają związki znaczeniowe pomiędzy pojęciami modelowanej dziedziny. W przypadku modeli na niższym poziomie abstrakcji asocjacja oznacza najczęściej, że obiekt jednej klasy ma stały, bezpośredni dostęp do obiektu drugiej klasy.

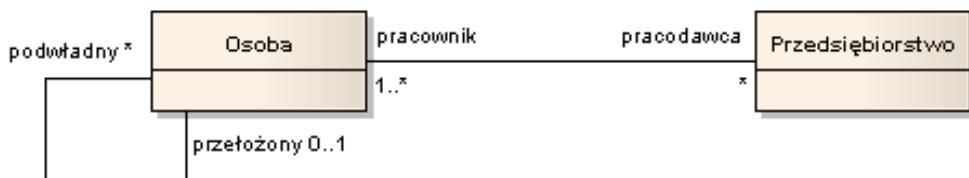
Asocjacja jest przedstawiana na diagramie jako linia ciągła łącząca powiązane klasy. Asocjacja może być uzupełniona o dodatkowe informacje, takie jak nazwa asocjacji, nazwy ról, krotności oraz kierunek asocjacji.



Rysunek 5.13: Asocjacja z nazwą

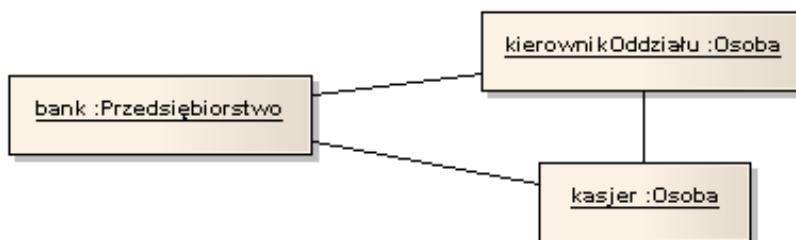
Rysunek 5.13 przedstawia przykład asocjacji z nazwą. Nazwa określa znaczenie. Strzałka obok nazwy wskazuje kierunek, w jakim należy odczytywać asocjację. Asocjacja z powyższego przykładu mówi nam: „osoba pracuje dla przedsiębiorstwa”. Możliwe jest zamodelowanie tej asocjacji w drugą stronę: „przedsiębiorstwo zatrudnia osobę”. Nazywanie każdej asocjacji w modelu nie jest konieczne. Nazwy nadaje się zwykle wtedy, kiedy pozwala to uniknąć niejednoznaczności.

Nazwy asocjacji nie trzeba podawać zwłaszcza w przypadku, kiedy wyraźnie określi się nazwy ról. Każda z klas będąca końcem asocjacji może być opisana nazwą roli, jaką dana klasa pełni w związku. W kolejnym przykładzie (Rysunek 5.14) Osoba pełni rolę pracownika względem Przedsiębiorstwa, natomiast Przedsiębiorstwo występuje w roli pracodawcy względem Osoby.



Rysunek 5.14: Role i krotności asocjacji

Rysunek 5.14 zawiera również przykład asocjacji, której oba końce stanowią ta sama klasa. Taka asocjacja oznacza, że obiekty danej klasy mogą być połączone z innymi obiektami tej samej klasy. W takim przypadku również możemy określić role końców asocjacji.



Rysunek 5.15: Instancje asocjacji

Rysunek 5.15 przedstawia diagram obiektów pokazujący instancje asocjacji z powyższego diagramu klas. Mamy tutaj dwie instancje klasy Osoba: kierownikOddziału oraz kasjer. Połączenie między nimi jest natomiast instancją asocjacji, której oba końce stanowią klasa Osoba występująca w różnych rolach.

Na tym samym diagramie obiektów widzimy, że obiekt klasy Przedsiębiorstwo łączy się z dwoma obiektami klasy Osoba – mamy więc dwie instancje asocjacji łączącej Osobę z Przedsiębiorstwem. O tym, ile obiektów klasy będącej jednym końcem asocjacji może być połączonych z obiektem klasy z drugiego końca asocjacji decyduje krotność. Ogólnie rzecz biorąc, krotność wskazuje dolne i górne granice dla liczby obiektów mogących brać udział w asocjacji. Krotności umieszczaemy na

końcach asocjacji, przy klasach, których te krotności dotyczą. Na omawianym diagramie klas (Rysunek 5.14) widzimy, że Przedsiębiorstwo może mieć co najmniej jednego pracownika. Jedna Osoba, natomiast, może mieć dowolną ilość pracodawców (przynajmniej w teorii) lub nie mieć żadnego. W praktyce najczęściej używa się krotności: jeden (1), zero lub jeden (0..1), dowolna ilość (*), co najmniej jeden (1..*). Można też określić dowolną kombinację liczb i zakresów, jak np. 2, 10..*, 5..7.

Jak zostało to wyjaśnione na początku tego podrozdziału, asocjacja pomiędzy dwiema klasami oznacza możliwość przejścia od obiektów jednej klasy do obiektów drugiej klasy w obu kierunkach. Czasami jednak zachodzi potrzeba ograniczenia możliwości nawigowania pomiędzy klasami tylko do jednego kierunku. Stosuje się wtedy asocjację jednokierunkową, która oznacza, że mając dostęp do obiektu na jednym końcu można bezpośrednio przejść do obiektów na drugim końcu. W modelu implementacyjnym jest to zazwyczaj możliwe dzięki przechowywaniu przez obiekt źródłowy odniesień do obiektów będących celem asocjacji.

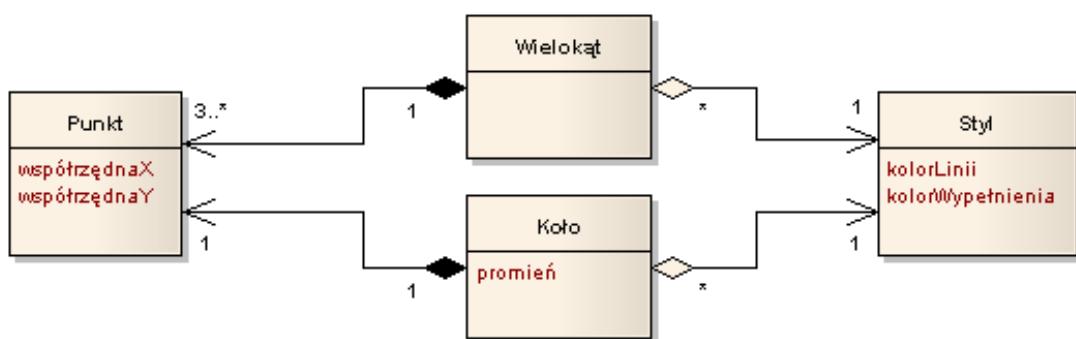


Rysunek 5.16: Asocjacja jednokierunkowa

Rysunek 5.16 przedstawia przykład asocjacji jednokierunkowej. Dla konkretnej Osoby, mamy bezpośredni dostęp do jej adresów, ale na podstawie danego Adresu nie jesteśmy w stanie bezpośrednio zidentyfikować powiązanych z nimi Osób. Istnienie asocjacji jednokierunkowej nie wyklucza jednak zupełnie możliwości wskazania Osób na podstawie Adresu – możliwe jest przejście pośrednie poprzez inne klasy.

Asocjacje odzwierciedlają zwykle związek strukturalny pomiędzy równorzędnymi klasami, znajdującymi się na tym samym poziomie pojęciowym. Czasem jednak klasy powiązane są relacją „całość-część”, w której klasa-agregat (całość) zawiera klasę-składnik (część). Innymi słowy, oznacza to zawieranie się obiektów jednej klasy w obiektach innej klasy. Taki rodzaj asocjacji nazywa się agregacją. Na diagramie agregację oznacza się poprzez umieszczenie pustego rombu po stronie klasy reprezentującej całość. Agregacja nie zmienia znaczenia nawigacji między klasami.

Oprócz zwykłej agregacji, UML udostępnia jej silniejszą wersję zwaną kompozycją. Ten rodzaj asocjacji charakteryzuje się wyłączną własnością oraz zależnością czasu życia całości i części. W odróżnieniu od agregacji, w przypadku kompozycji obiekt będący częścią może należeć wyłącznie do jednej całości. Ponadto, części mogą powstawać po utworzeniu całości i umierają razem z nią. Usunięcie całości powoduje kaskadowe usunięcie wszystkich części. Na diagramie kompozycję oznacza się poprzez umieszczenie zaczerwonego rombu po stronie klasy reprezentującej całość. Należy pamiętać, że krotność po stronie klasy reprezentującej całość w relacji kompozycji musi być zawsze równa 1.



Rysunek 5.17: Agregacja i kompozycja

Rysunek 5.17 pokazuje przykład agregacji i kompozycji. Wielokąt może zawierać 3 lub więcej instancji Punktu, przy czym żadna z tych instancji nie może należeć do Koła, które posiada własną

instancję Punktu. Instancja Stylu natomiast, może być dzielona przez wiele Wielokątów i Kół. Usunięcie Wielokąta bądź Koła spowoduje usunięcie wszystkich należących do niego Punktów, ale nie spowoduje usunięcia zawartego w nim Stylu.

Zależności

Obok asocjacji, w modelu klas można używać innego rodzaju związków pomiędzy klasami, a mianowicie zależności. Zależność oznacza, że jeden element modelu używa innego elementu. Np. jedna klasa używa drugiej jako argumentu w sygnaturze operacji. Zmiany dokonane w specyfikacji jednej klasy mogą mieć wpływ na klasę, która używa tej pierwszej.

Rysunek 5.18 przedstawia przykład zależności pomiędzy klasami. Widać na nim, że klasa Produkt jest wykorzystywana jako argument operacji w klasie Koszyk. Zmiana sposobu zachowania klasy Produkt może zatem pociągnąć za sobą zmianę w zachowaniu klasy Koszyk, która jest zależna od tej pierwszej.



Rysunek 5.18: Zależność

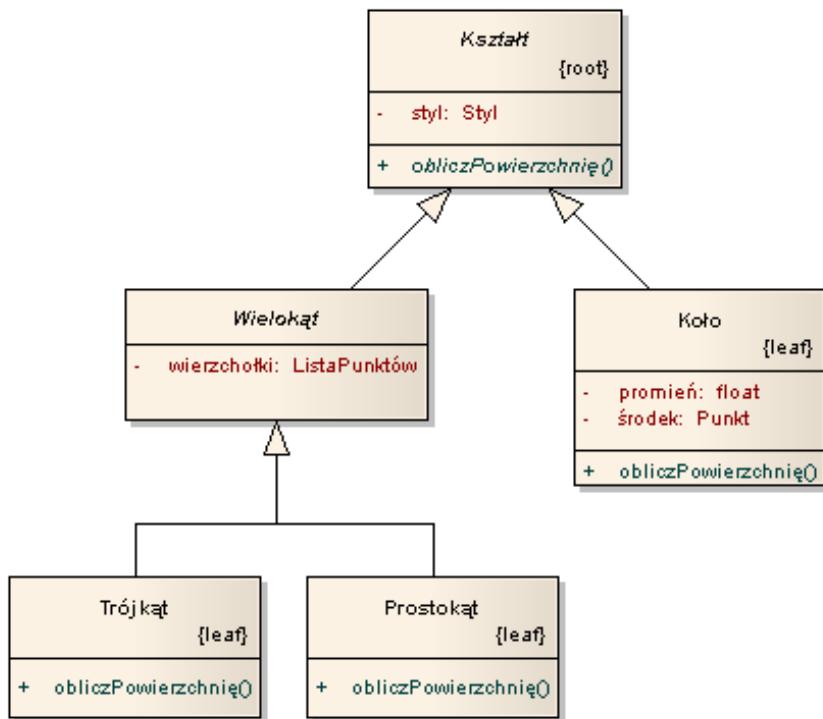
Generalizacja i klasy abstrakcyjne

Ważnym rodzajem związku pomiędzy elementami modelu w UML, zwłaszcza modelu klas, jest generalizacja. Generalizacja jest to związek między elementem ogólnym, zwanym również nadklassą lub przodkiem, a pewnym specyficzny jego rodzajem zwanym podklassą lub potomkiem. Potomek dziedziczy wszystkie właściwości przodka – jego atrybuty, operacje, związki. Potomek może rozszerzać zbiór właściwości odziedziczonych od przodka o swoje własne cechy. Potomek może zatem wystąpić wszędzie tam, gdzie spodziewany jest przodek, ale nie na odwrót. Generalizacja przedstawiana jest na diagramie za pomocą strzałki z zamkniętym pustym w środku grotkiem wskazującym przodka.

Generalizacja umożliwia stworzenie hierarchii dziedziczenia. Klasy ogólne znajdują się na górze hierarchii, a bardziej szczegółowe na dole. Klasa może mieć dowolną ilość potomków. Jeśli chodzi o przodków, to rozróżniamy dziedziczenie pojedyncze – gdy klasa ma jednego przodka, oraz dziedziczenie wielokrotne – gdy klasa ma wielu przodków.

Dla danej klasy możemy ograniczyć możliwość tworzenia klas potomnych. Taką klasę nazywamy liściem i oznaczamy na diagramie napisem „{leaf}” umieszczonym pod nazwą klasy. Klasę znajdującą się na szczytzie hierarchii dziedziczenia, która nie ma przodków nazywamy korzeniem i oznaczamy na diagramie napisem „{root}” umieszczonym pod jej nazwą (patrz Rysunek 5.19).

Niektóre klasy w hierarchii dziedziczenia mogą być określone jako abstrakcyjne, czyli takie, dla których nie można utworzyć instancji. Klasa takie nie dostarczają implementacji niektórych lub wszystkich swoich operacji, zwanych operacjami abstrakcyjnymi. Implementację operacji abstrakcyjnych powinny zapewnić klasy potomne, które następnie mogą być użyte wszędzie tam, gdzie spodziewana jest klasa abstrakcyjna. Nazwy klas i operacji abstrakcyjnych zapisuje się kursywą. W przedstawionym przykładzie (Rysunek 5.19) klasa Kształt zawiera abstrakcyjną operację obliczPowierzchnię(). Klasa Koło, Trójkąt oraz Prostokąt, będące klasami konkretnymi, dostarczają własnej specyficznej implementacji tej metody.



Rysunek 5.19: Generalizacja

5.2.2. Model komponentów

Podczas tworzenia architektury systemu informatycznego, jedną z fundamentalnych decyzji jest podział systemu na logiczne jednostki funkcjonalne. Dzięki takiemu podziałowi, system zyskuje bardzo istotny poziom abstrakcji – setki czy tysiące elementów można pogrupować w kilkanaście lub kilkadziesiąt jednostek funkcjonalnych. Te logiczne jednostki funkcjonalne nazywamy w języku UML komponentami. Komponent ma cechy pakietu, ponieważ grupuje inne elementy. W odróżnieniu od pakietu, komponent realizuje pewną funkcjonalność, tzn. potrafi dostarczać określone usługi innym komponentom. W tym sensie jest on rodzajem klasy, tylko na wyższym poziomie abstrakcji.

Dobrze skonstruowany komponent określa abstrakcję o ścisłe określonych granicach i dobrze zdefiniowanym zachowaniu. Dzięki temu dany komponent można łatwo zastąpić innym, kompatybilnym komponentem.

Komponenty i zależności między nimi

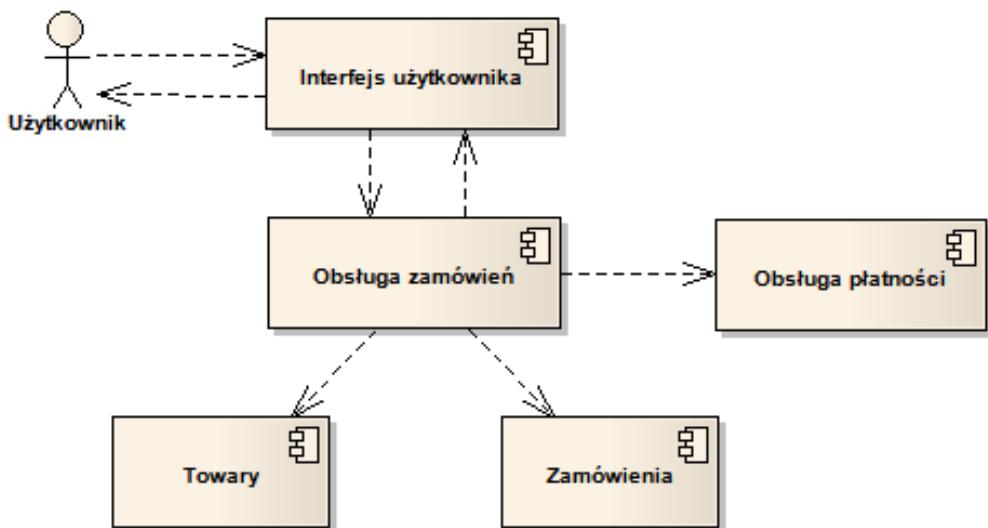
Rysunek 5.20 przedstawia symbol graficzny komponentu w języku UML. W podstawowej formie jest to prostokąt zawierający małą ikonę w prawym górnym rogu, symbolizującą komponent. Każdy komponent musi mieć przypisaną nazwę, która wyróżnia go spośród innych komponentów. Przy nazywaniu komponentów stosuje się te same zasady, co przy nazywaniu klas. Podobnie jak klasa, komponent może mieć atrybuty i operacje, chociaż stosuje się je bardzo rzadko.



Rysunek 5.20: Komponent

Same komponenty są niewystarczające do ukazania pełnej architektury logicznej systemu. Jak powiedzieliśmy, podstawową cechą komponentów jest udostępnianie oraz korzystanie z usług innych komponentów. Funkcjonalność systemu jest realizowana poprzez współpracę wielu kom-

ponentów. Oznacza to, że są one od siebie zależne. Istotnym elementem modelu komponentów są zatem zależności między nimi.



Rysunek 5.21: Zależności między komponentami

Rysunek 5.21 przedstawia prosty przykład diagramu komponentów. Na diagramie tym widzimy komponenty oraz zależności między nimi a także aktora, który bezpośrednio korzysta z komponentu udostępniającego funkcjonalność interfejsu użytkownika. Ponadto na diagramach komponentów można umieszczać takie elementy jak porty, interfejsy oraz zależności i połączenia między interfejsami. Elementy te opisane będą poniżej.

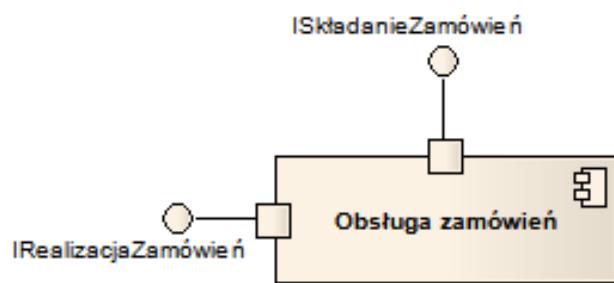
Dobrą praktyką przy tworzeniu architektury komponentowej jest podział systemu na komponenty o wysokiej zwartości oraz słabych więzach między sobą. Wysoka zwartość oznacza, że komponent powinien grupować elementy (klasy) ściśle ze sobą powiązane i realizujące pewien zamknięty podsystem. Komponent powinien izolować ten podsystem od otoczenia i udostępniać na zewnątrz tylko tę jego funkcjonalność, która jest wymagana przez inne komponenty. Innymi słowy, cała praca jest wykonywana wewnętrz komponentów a pomiędzy nimi przekazywane są jedynie efekty tej pracy. W ten sposób więzy między komponentami są słabe.

Więcej praktyk związanych z tworzeniem dobrej architektury komponentowej opisanych zostało w Rozdziale 8.

Porty i interfejsy

Na podstawie zależności możemy stwierdzić, między którymi komponentami istnieje komunikacja. Nie wiemy natomiast nic na temat usług, jakie komponenty udostępniają sobie nawzajem. Porozumiewanie się pomiędzy komponentami powinno odbywać się poprzez formalnie zdefiniowane punkty dostępowe. Punkty te nazywają się portami i stanowią jedyne miejsca, poprzez które komponent udostępnia swoje usługi na zewnątrz. Z portami ściśle związane są interfejsy. Interfejs to zestaw operacji, które określają usługi oferowane przez klasę lub komponent. Można powiedzieć, że interfejsy są deklaracją zakresu odpowiedzialności komponentu – jest on odpowiedzialny za takie operacje, jakie zostały zdefiniowane w jego interfejsach. Sposób realizacji tych usług nie jest widoczny na zewnątrz komponentu.

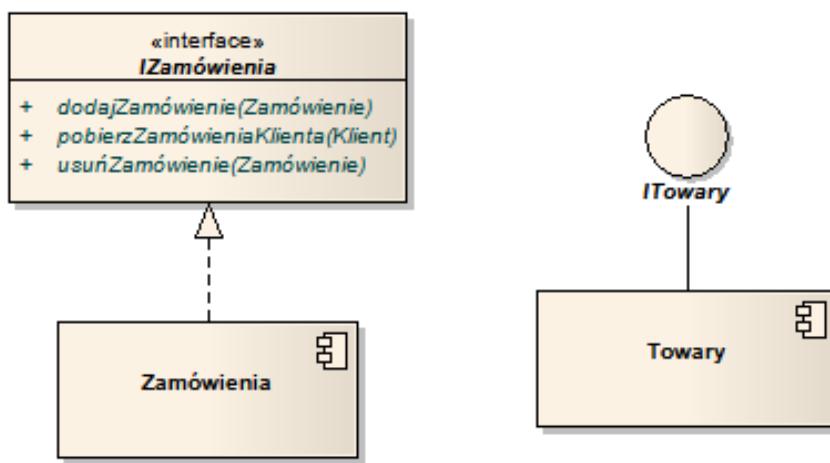
Port w języku UML przedstawiany jest w postaci małego prostokąta wystającego z symbolu komponentu (Rysunek 5.22). Może mieć on własną nazwę. Porty są dostępne publicznie dla wszystkich innych komponentów, które są związane z komponentem udostępniającym te porty.



Rysunek 5.22: Porty i interfejsy

Interfejsy mogą być umieszczane na diagramach w różnej formie. Interfejs związany z portem oznaczany jest jako wystające z niego kółko. Interfejs opisany jest nazwą. Przyjęto się, że nazwy interfejsów poprzedza się wielką literą „I”.

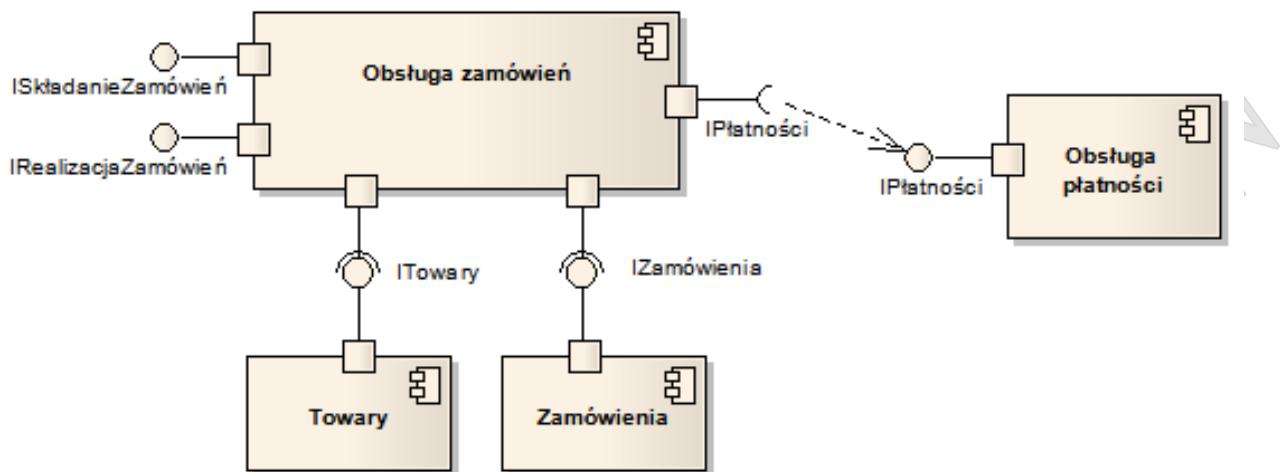
Na diagramach klas, interfejs może być też pokazany w postaci klasy oznaczonej stereotypem «interface» (Rysunek 5.23). W takiej formie prezentacji, możliwe jest pokazanie zestawu operacji, jakie dany interfejs definiuje. Na diagramach klas możliwe jest także pokazanie hierarchii dziedziczenia interfejsów, podobnie jak w przypadku klas. Należy zwrócić uwagę na to, że interfejs jest elementem abstrakcyjnym i – podobnie jak klasa abstrakcyjna – nie ma swoich instancji. Instancje są natomiast tworzone dla elementów, które realizują dany interfejs, np. komponentów.



Rysunek 5.23: Realizacja interfejsów przez komponenty na diagramach klas

Na diagramach klas realizacja interfejsu przez komponent zaznaczana jest relacją podobną do relacji generalizacji. Interfejs jest elementem ogólnym a komponentem elementem uszczegóławiającym. W odróżnieniu od relacji dziedziczenia, relacja realizacji jest oznaczana linią przerywaną. Jeżeli nie ma potrzeby przedstawiania operacji interfejsu realizowanego przez komponent, można przedstawić go w postaci koła połączonego z komponentem relacją realizacji. W tym wariantie, relacja ta jest oznaczana zwykłą linią.

Niektóre komponenty, aby spełnić swoją określzoną rolę w systemie, muszą korzystać z funkcjonalności udostępnianej przez inne komponenty. Pełna specyfikacja komponentu wymaga zatem określenia zarówno interfejsów dostarczanych jak i interfejsów wymaganych przez ten komponent. Interfejsy wymagane przez komponent oznaczane są w postaci półokrągów wystających z jego portów (Rysunek 5.24). Zależności między interfejsami możemy zaznaczyć na diagramie komponentów, łącząc interfejsy udostępniane z wymaganymi. Istnieją dwa warianty notacji dla tych zależności. Oba pokazane są na poniższym rysunku. Pierwszy wariant jest zwykłą relacją zależności łączącą odpowiednie interfejsy. Drugi wariant to specjalny rodzaj powiązania, które możemy nazywać powiązaniem montażowym (ang. assembly). Nazwa przy takim powiązaniu odpowiada nazwie interfejsu udostępnianego przez jeden komponent a wymaganego przez drugi komponent.



Rysunek 5.24: Interfejsy wymagane oraz zależności między interfejsami

5.3. Modelowanie struktury fizycznej

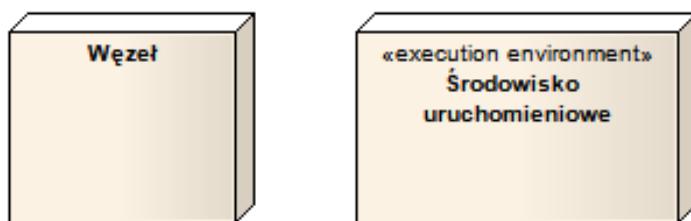
Model struktury fizycznej systemu oprogramowania opisuje infrastrukturę fizyczną systemu. Opisuje on fizyczne składniki działającego systemu jak np. maszyny czy środowiska uruchomieniowe, na których zainstalowane zostaną jednostki logiczne systemu. Model fizyczny określa też metody komunikacji pomiędzy poszczególnymi węzłami fizycznymi. W niniejszym podrozdziale opiszemy, w jaki sposób modeluje się strukturę fizyczną w języku UML.

5.3.1. Model wdrożenia

Tworząc model komponentów, architekt dokonuje podziału oprogramowania systemu na składniki logiczne. Oprócz tego, architekt musi też zaprojektować strukturę fizyczną systemu, czyli środowisko wdrożeniowe. Oznacza to, że musi podjąć decyzje dotyczące m.in. rodzajów oraz liczby komputerów oraz innych urządzeń, a także połączeń między nimi. Konieczne jest też powiązanie komponentów systemu z maszynami, na których te komponenty będą działały. Taki model struktury fizycznej można ukazać na diagramach wdrożenia. Podstawowe elementy, jakie umieszcza się na tego typu diagramach, to węzły oraz asocjacje między nimi. Można również umieszczać komponenty oraz artefakty powiązane z węzłami.

Węzły i związki między węzłami

Węzeł to fizyczny składnik działającego systemu. Reprezentuje on środowisko umożliwiające przetwarzanie danych. Najczęściej takim środowiskiem są maszyny posiadające jakieś zasoby obliczeniowe oraz pamięć (stacje robocze, stacje serwerowe, terminale, routery, itp.) bądź inne urządzenia tworzące infrastrukturę fizyczną systemu. Węzły mogą też reprezentować środowiska uruchomieniowe pracujące na określonych maszynach. Przykładem takich środowisk są serwery aplikacyjne, serwery WWW, serwery baz danych czy maszyny wirtualne.



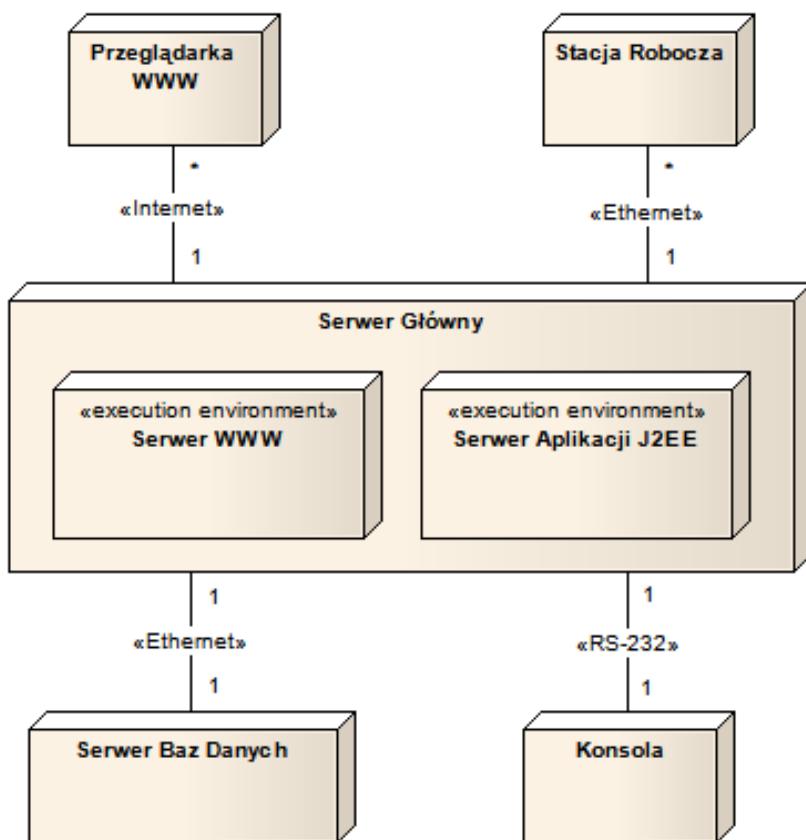
Rysunek 5.25: Węzły

Węzły na diagramie przedstawiane są w postaci prostopadłościanu. Każdy węzeł posiada nazwę. Rysunek 5.25 przedstawia notację dla węzłów.

Zauważmy, że węzeł może być oznaczony stereotypem, który precyzuje jego znaczenie. Zwykły węzeł najczęściej reprezentuje fizyczne urządzenie oferujące zasoby obliczeniowe. Natomiast, węzeł oznaczony stereotypem «execution environment» jest specjalizowanym węzłem, który może reprezentować środowiska uruchomieniowe dla określonych typów komponentów. Komponenty te są wdrażane na węzłach tego rodzaju w postaci tzw. wykonywalnych artefaktów.

Węzły mogą być zagnieżdżone w innych węzłach. Przykładowo, środowiska uruchomieniowe reprezentujące serwer WWW oraz serwer aplikacji mogą być zainstalowane na węźle reprezentującym fizyczną maszynę serwerową (patrz Rysunek 5.26 **Błąd! Nie można odnaleźć źródła odwołania.**)

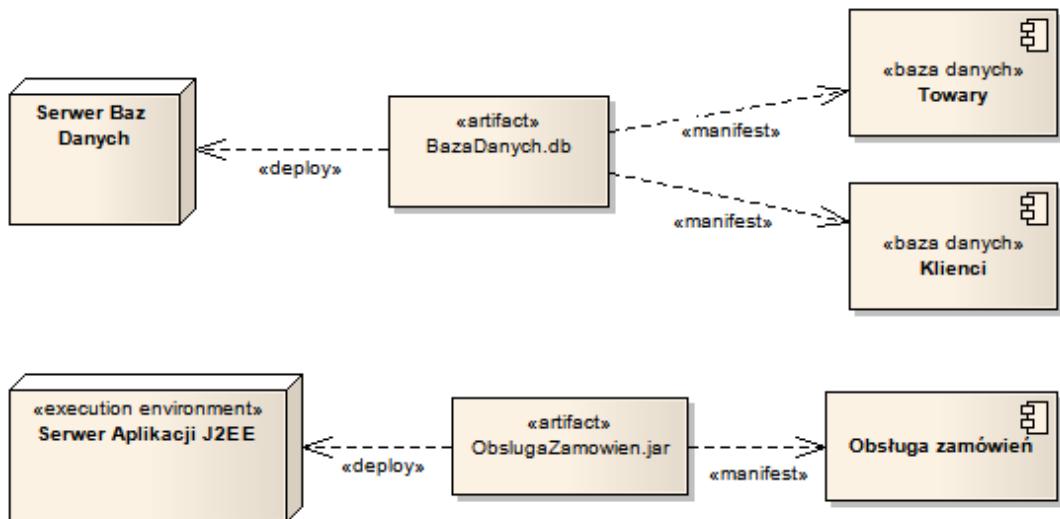
Węzły mogą być powiązane ze sobą. Najczęściej występującym powiązaniem między węzłami jest specjalny rodzaj asocjacji zwany ścieżką komunikacyjną (ang. *communication path*). Ścieżka komunikacyjna reprezentuje połączenie, przez które węzły mogą wymieniać sygnały i dane. Zazwyczaj jest to połączenie fizyczne. Za pomocą stereotypów można sprecyzować rodzaj tego połączenia, np. rodzaj sieci albo protokołu. Ścieżka komunikacyjna może mieć wszystkie właściwości asocjacji, takie jak role, liczebność, kierunek nawigacji. Przykład połączeń między węzłami przedstawia poniższy rysunek (Rysunek 5.26).



Rysunek 5.26: Ścieżki komunikacyjne oraz węzły zagnieżdżone

Artefakty

Na diagramach wdrożenia, oprócz węzłów, możemy również umieszczać artefakty. Artefakty są elementami, które reprezentują różnego rodzaju jednostki implementacji systemu zawierające możliwe do uruchomienia kod. Mogą to być pliki wykonywalne, biblioteki, strony WWW, skrypty, czy bazy danych. Artefakty przedstawiane są na diagramie w postaci prostokąta z nazwą oraz stereotypem «artifact». Nazwa opisuje plik lub komponent oprogramowania, który dany artefakt reprezentuje.

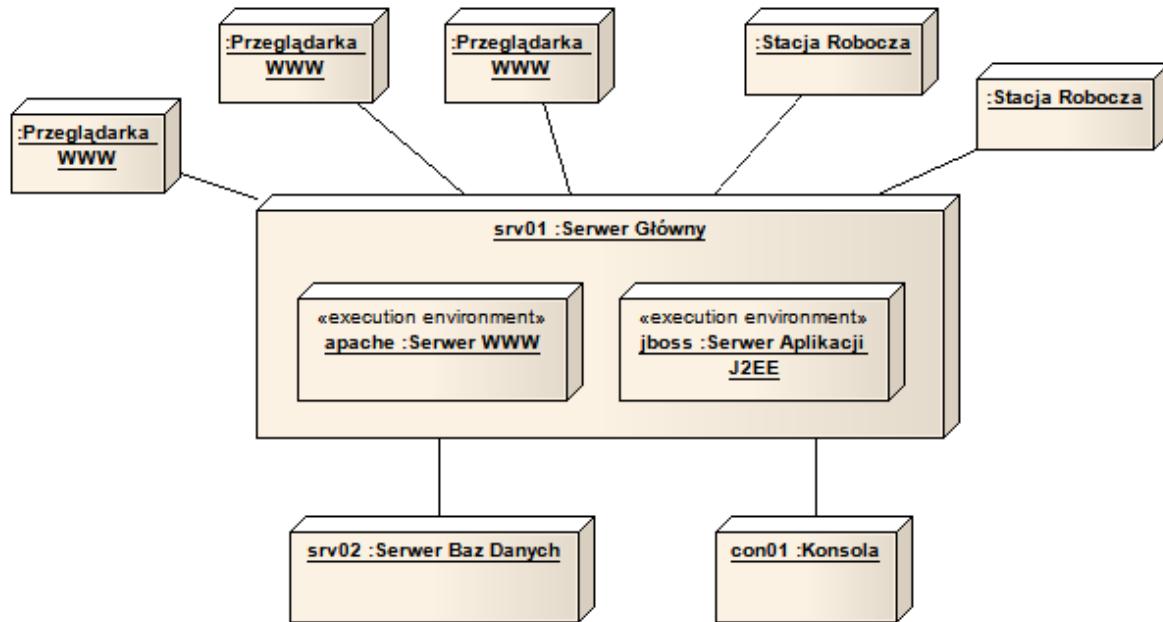


Rysunek 5.27: Artefakty

Fakt zainstalowania konkretnego artefaktu w węźle oznacza się zależnością typu «deploy». Tego typu zależności przedstawia Rysunek 5.27. Widać tam również zależność artefaktów i komponentów z modelu logicznego. Artefakty są bowiem fizyczną reprezentacją komponentów logicznych. Każdy komponent może być reprezentowany przez jeden lub wiele różnych artefaktów. Jeden artefakt może też reprezentować wiele komponentów. Zależności między artefaktami i komponentami oznacza się stereotypem «manifest».

Instancje węzłów

Podobnie jak klasy, węzły mogą mieć swoje instancje. Instancja węzła może posiadać nazwę oddzieloną dwukropkiem od typu węzła. Nazwa, podobnie jak w przypadku obiektów, jest podkreślona. Rysunek 5.28 przedstawia diagram zawierający instancje węzłów.



Rysunek 5.28: Instancje węzłów

5.4. Podsumowanie

Wszystkie przedstawione diagramy opisu struktury mogą być wykorzystywane podczas modelowania statycznych aspektów budowy systemów oprogramowania. Strukturę logiczną modeluje się

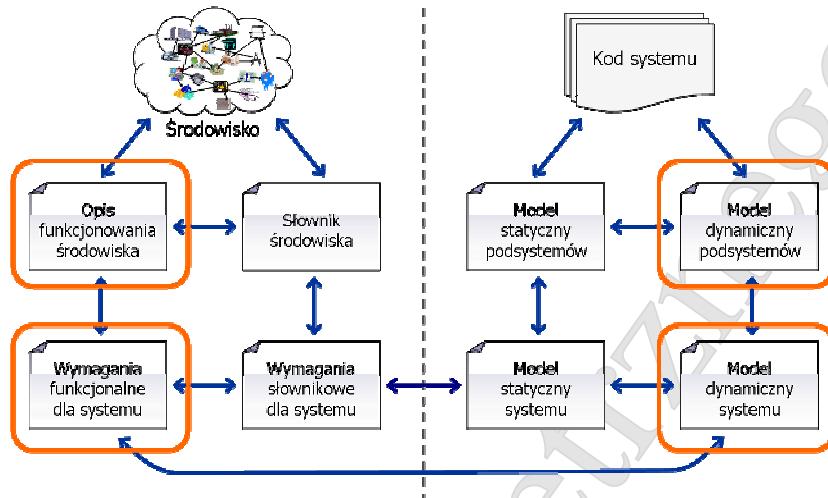
wykorzystując głównie diagramy klas oraz komponentów. Dzięki diagramom wdrożenia możliwe jest dodatkowo modelowanie fizycznej infrastruktury sprzętowej, na której oprogramowanie będzie wdrożone.

Rzadko zachodzi potrzeba stosowania wszystkich diagramów opisu struktury w jednym projekcie. Niektóre z nich używane są jedynie w specyficznych sytuacjach, natomiast do wielu zastosowań wystarczają jedynie podstawowe diagramy. Poziom szczegółowości rysowanych diagramów powinien zależeć od tego, na jakim etapie projektowania systemu jesteśmy oraz od tego, kto będzie ich odbiorcą. Próba tworzenia wszystkich możliwych diagramów z wszystkimi detalami może spowodować, że ugrzęźniemy w całej masie niepotrzebnych szczegółów, tracąc właściwy cel modelowania, jakim jest stworzenie elastycznej, skalowej struktury systemu.

Do użytku wewnętrznego

6. Modelowanie dynamiki systemu w języku UML

W niniejszym rozdziale przedstawimy diagramy, jakie oferuje język UML do opisywania dynamiki systemu oprogramowania oraz środowiska. Modele na ścieżce od środowiska do kodu, które możemy tworzyć przy pomocy diagramów opisu dynamiki zostały wyróżnione na rysunku (Rysunek 5.1).

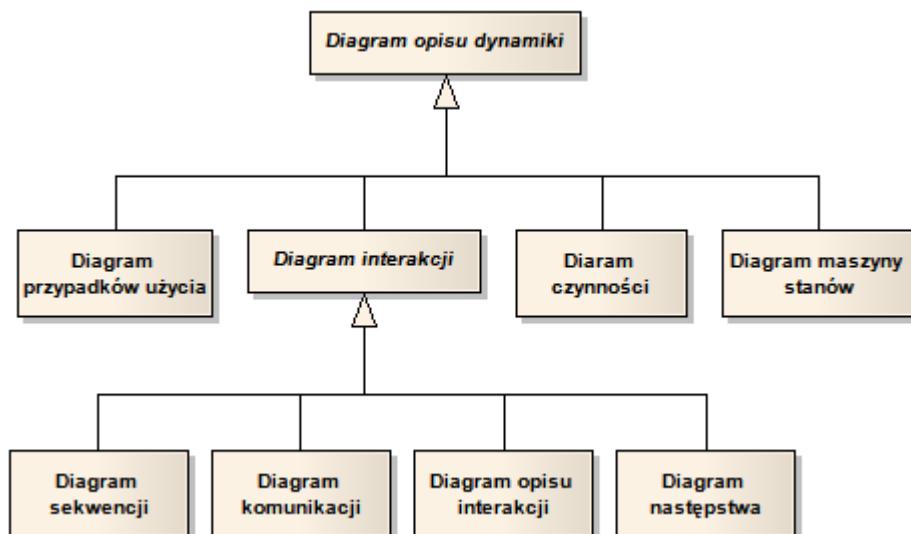


Rysunek 6.1: Modele dynamiki na ścieżce od środowiska do kodu

Modele dynamiki ukazują system w działaniu, dlatego też niezbędne jest pokazanie następstwa czasu. Modele dynamiki powinny ukazywać kolejność czynności wykonywanych przez system, bądź też kolejność interakcji zachodzących pomiędzy systemem i jego użytkownikami. Model dynamiki musi też mieć ścisły związek z modelem struktury, ponieważ modele dynamiki ukazują elementy struktury w działaniu.

6.1. Przegląd diagramów opisu dynamiki w języku UML

Rysunek 5.2 przedstawia klasyfikację diagramów języka UML, służących do opisu dynamiki. Istnieją cztery podstawowe rodzaje diagramów opisu struktury: diagram przypadków użycia, diagram interakcji, diagram czynności oraz diagram maszyny stanów. Zauważmy, że diagram interakcji nie jest żadnym konkretnym typem diagramu. Jest on generalizacją czterech konkretnych diagramów: diagramu sekwencji, diagramu komunikacji, diagramu opisu interakcji oraz diagramu następstwa.



Rysunek 6.2: Klasyfikacja diagramów opisu dynamiki w języku UML

6.1.1. Diagram przypadków użycia

W języku UML, za pomocą diagramów przypadków użycia możemy opisywać funkcjonalność systemu z punktu widzenia jego użytkowników. Przypadki użycia (ang. *use case*) są jednostkami opisu dynamiki, które prezentują zewnętrzny sposób zachowania się systemu. Przypadek użycia jest zbiorem scenariuszy (ang. *scenario*). Scenariusze, z kolei, są sekwencjami interakcji, które prowadzą do celu określonego przez przypadek użycia. Sekwencję interakcji inicjuje zazwyczaj jakiś obiekt spoza systemu, zwany aktorem (ang. *actor*). Sekwencja interakcji może skończyć się osiągnięciem celu przypadku użycia, bądź niepowodzeniem.

Przypadki użycia są powszechnie stosowane do tworzenia specyfikacji wymagań funkcjonalnych na budowany system.

Bardziej szczegółowy opis modelu przypadków użycia znajduje się w dalszej części tego rozdziału.

6.1.2. Diagram czynności

Diagramy czynności przedstawiają graf opisujący czynności, które są podejmowane w zależności od warunków zewnętrznych lub stanu systemu. Można powiedzieć, że dynamika systemów składa się z sekwencji wykonywanych przez ten system akcji. Może ona opisywać np. proces biznesowy, scenariusze przypadków użycia lub algorytm jakiejś operacji. Dlatego też, ważnym elementem modelowania dynamiki systemów są diagramy czynności.

Bardziej szczegółowy opis diagramów czynności znajduje się w dalszej części tego rozdziału.

6.1.3. Diagram sekwencji

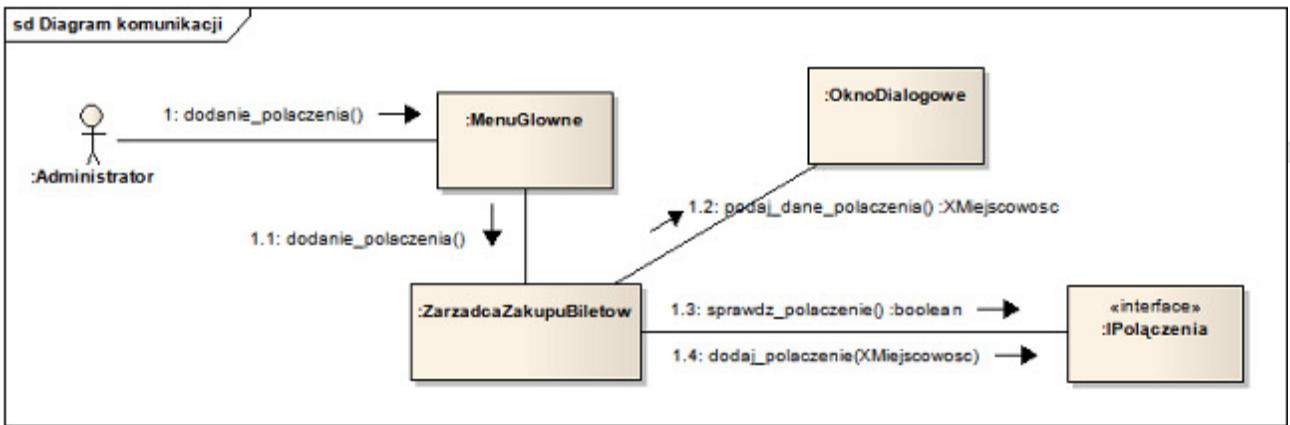
Podczas modelowania systemu jako zbioru komunikujących się obiektów, potrzebujemy sposobu na pokazanie ciągów komunikatów wymienianych między obiektami. Komunikaty wymieniane są pomiędzy liniami życia obiektów, które określają czas życia obiektów. Komunikat może oznaczać np. wywołanie operacji obiektu lub jego utworzenie. Ciąg komunikatów wymienianych pomiędzy liniami życia nazywamy interakcją (ang. *interaction*). Interakcje możemy modelować na diagramach interakcji. Diagram interakcji nie jest jednak konkretnym typem diagramu, lecz może przybrać cztery postaci.

Jedną z postaci diagramów interakcji jest diagram sekwencji. Na tego typu diagramach możemy pokazać następstwo czasowe w sekwencji komunikatów.

Bardziej szczegółowy opis diagramów sekwencji znajduje się w dalszej części tego rozdziału.

6.1.4. Diagram komunikacji

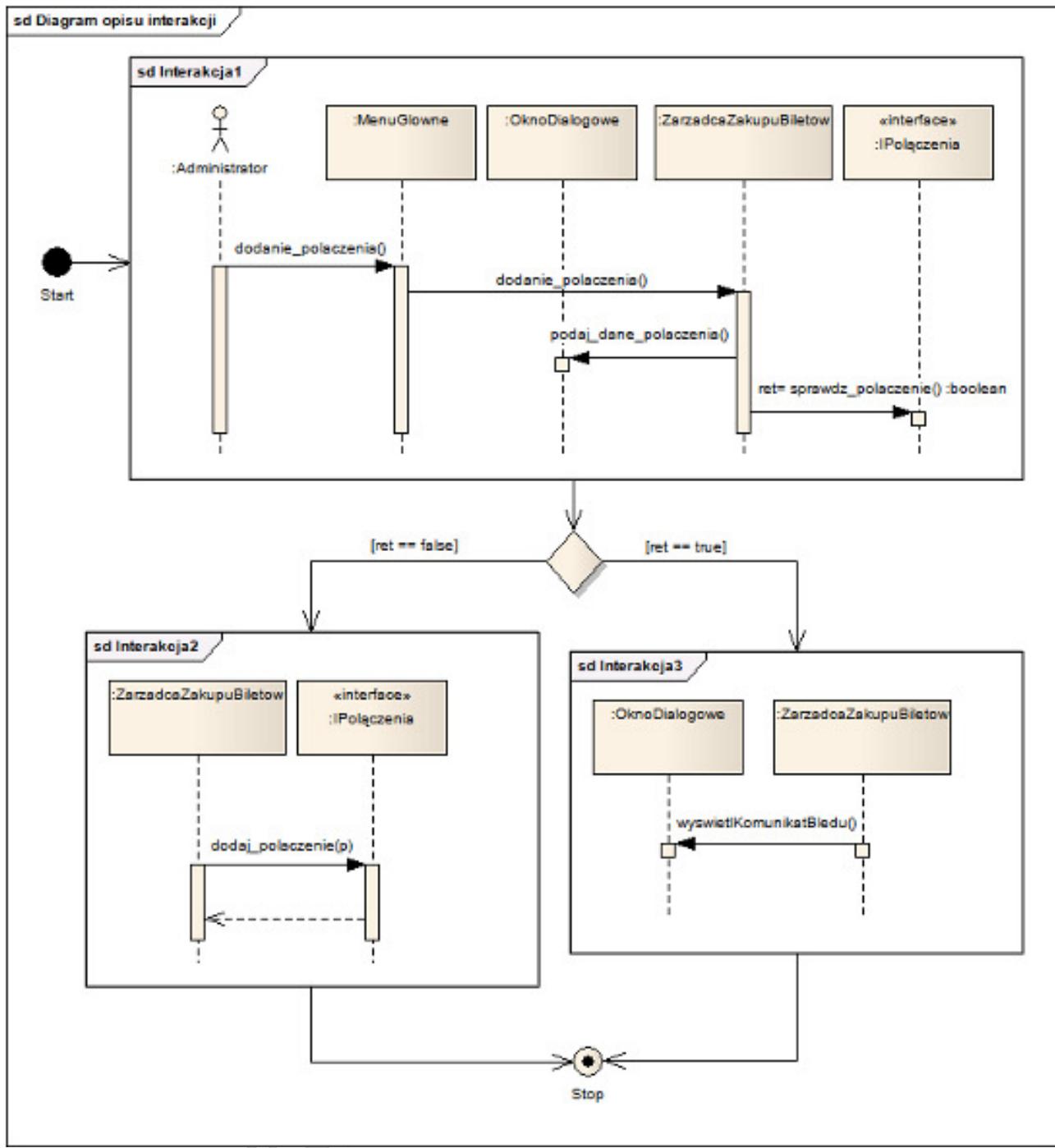
Innym rodzajem diagramu interakcji jest diagram komunikacji. Diagramy komunikacji niosą bardzo podobną zawartość informacyjną jak diagramy sekwencji. Mają jednak inny układ. Przykład diagramu komunikacji przedstawia rysunek (Rysunek 6.3). Diagram z rysunku jest odpowiednikiem przedstawionego w dalszej części rozdziału diagramu sekwencji. Diagramy komunikacji bardzo przypominają statyczne diagramy obiektów. Dynamikę współdziałania obiektów prezentujemy poprzez umieszczanie komunikatów. Komunikaty oznaczamy strzałkami i umieszczaemy je obok łączników między obiektami. Komunikaty oznaczone są liczbami, które określają kolejność ich wykonania. Komunikaty zwrotne nie są zazwyczaj pokazywane dla polepszenia czytelności diagramów.



Rysunek 6.3: Przykładowy diagram komunikacji

6.1.5. Diagram opisu interakcji

Trzecim rodzajem diagramu interakcji jest diagram opisu interakcji. Diagramy opisu interakcji pokazują następstwo interakcji. Przypominają one trochę diagramy czynności, gdyż bazują na notacji używanej na tamtych diagramach. Różnica polega na tym, że poszczególne akcje zastąpione są interakcjami. Diagramy opisu interakcji są przydatne w przypadku, kiedy chcemy zebrać złożone czynności zawierające w sobie ciąg kolejnych interakcji między obiektami.

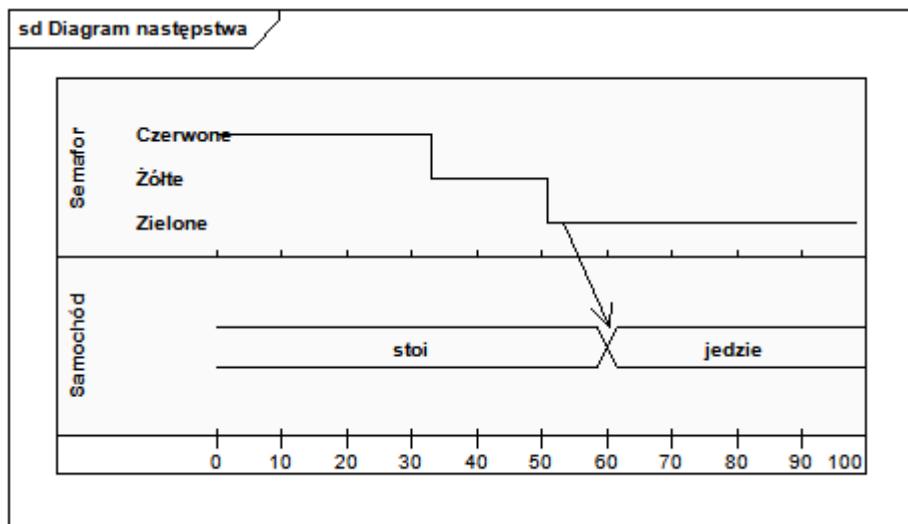


Rysunek 6.4: Przykładowy diagram opisu interakcji

6.1.6. Diagram następstwa

Diagram następstwa jest ostatnim rodzajem diagramów interakcji. Używany jest do pokazywania zmian stanu lub wartości jednego lub więcej elementów w czasie oraz pod wpływem komunikatów przesyłanych między obiektami. Rysunek 6.5 pokazuje przykład diagramu następstwa.

Na diagramie następstwa możemy umieścić dwa rodzaje linii życia elementów: linię życia stanu (ang. *state lifeline*) oraz linię życia wartości (ang. *value lifeline*). Linia życia czasu pokazuje zmianę stanu obiektu w czasie. Podziałka czasu jest pokazywana na osi poziomej. Oś pionowa zawiera możliwe stany obiektu.



Rysunek 6.5: Przykładowy diagram następstwa

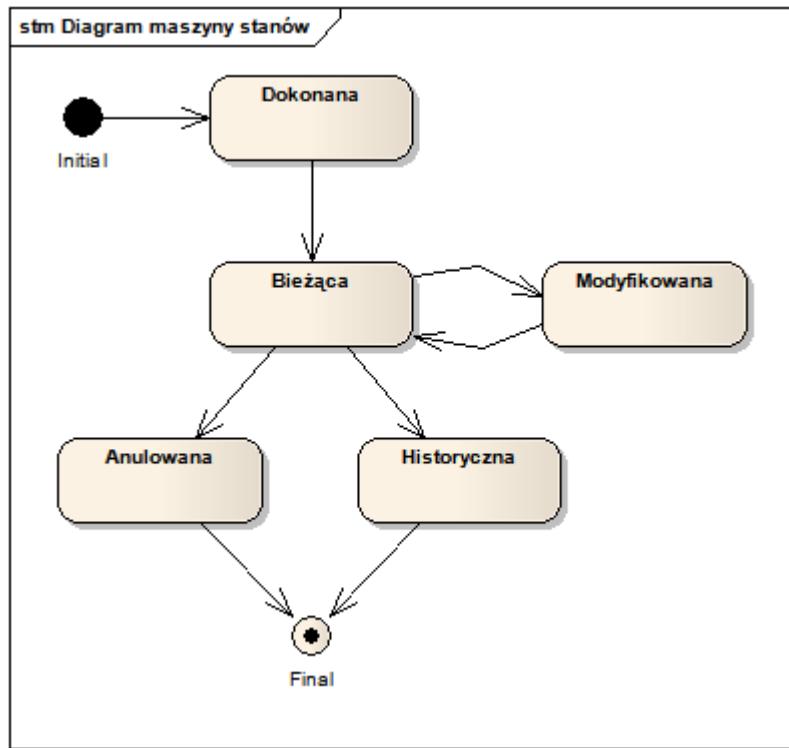
Linia życia wartości pokazuje zmianę jakiejś wartości w czasie. Podziałka czasu również jest podana na osi poziomej. Wartość jest pokazywana w postaci równoległych linii, które krzyżują się gdy wartość się zmienia.

Oba rodzaje linii życia mogą być dowolnie umieszczane na jednym diagramie. Muszą mieć jedynie tę samą podziałkę czasu. Pomiędzy liniami życia mogą być umieszczane komunikaty, które powodują zmianę stanu bądź wartości.

6.1.7. Diagram maszyny stanów

Diagramy maszyny stanów służą do pokazywania sposobu zachowania się jakiegoś elementu modelu w postaci przejść pomiędzy jego stanami. Stan oznacza pewną stabilną sytuację elementu, w jakiej może się on znaleźć. Zmiany stanów elementu mogą następować pod wpływem określonych zdarzeń, np. komunikatów przesyłanych przez inne obiekty.

Rysunek 6.6 przedstawia przykładowy diagram maszyny stanów, pokazujący możliwe stany obiektu klasy Rezerwacja w systemie rezerwacji biletów. Diagramy takie zawierają przede wszystkim stany (ang. *state*) oraz przejścia (ang. *transition*) między stanami. Stan jest reprezentowany przez zaokrąglony prostokąt, który zawiera w środku tekst opisujący stan. Przejścia pomiędzy stanami reprezentowane są poprzez strzałki. Obok przejścia można opcjonalnie umieścić nazwę komunikatu, który powoduje zmianę stanu oraz warunek zmiany stanu. Stan początkowy określony jest przez węzeł początkowy a stan końcowy – przez węzeł końcowy.



Rysunek 6.6: Przykładowy diagram maszyny stanów

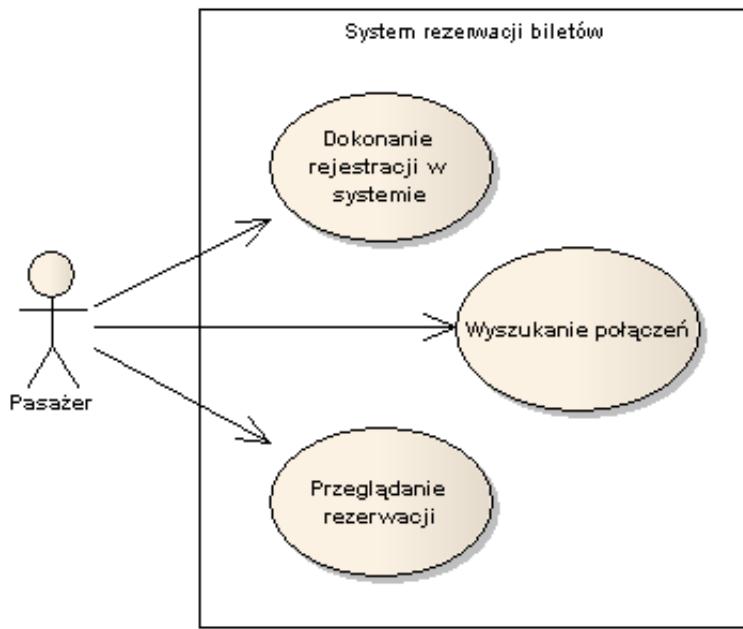
6.2. Model przypadków użycia

Żaden system oprogramowania nie istnieje sam dla siebie. Każdy system współdziała z określonym zbiorem aktorów. Aktorami są zazwyczaj ludzie wcielający się w rolę użytkowników systemu, ale mogą to być także inne systemy. Aktorzy oczekują od systemu określonego działania – korzystając z niego, chcą osiągnąć pewne cele. W języku UML, zachowanie systemu jest specyfikowane za pomocą przypadków użycia. Przypadek użycia to opis interakcji aktora z systemem, który prowadzi do dostarczenia aktorowi określonego wyniku.

Model przypadków użycia jest podstawą specyfikacji wymagań na system. Model taki pozwala określić oczekiwane zachowanie budowanego systemu bez konieczności określania sposobu implementacji tego zachowania. Podział funkcjonalności systemu na dobrze zdefiniowane jednostki, jakimi są przypadki użycia, pozwala łatwiej wypracować porozumienie między twórcami systemu a jego przyszłymi użytkownikami oraz zamawiającymi. Pozwala też łatwiej zarządzać realizacją systemu poprzez zorganizowanie przyrostowego cyklu budowy oprogramowania. Podział specyfikacji wymagań na przypadki użycia wydaje się być kluczowy dla poprawnej organizacji procesu twórczego.

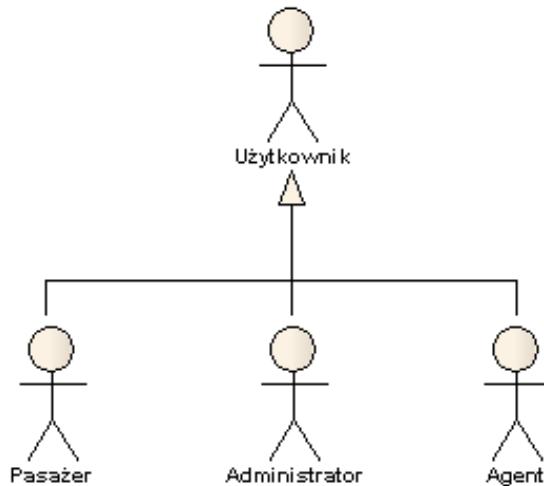
6.2.1. Aktorzy i przypadki użycia

W języku UML, reprezentacją współpracowników systemu są aktorzy. Aktor reprezentuje rolęgraną przez kogoś lub przez coś spoza modelowanego systemu w stosunku do tego systemu. Aktor współpracuje z systemem w celu uzyskania pewnych usług lub sam udostępnia pewne usługi systemowi. Aktor w notacji modelu przypadków użycia jest oznaczany jako ikona człowieka narysowanego prostymi kreskami (Rysunek 6.7).



Rysunek 6.7: Aktorzy oraz przypadki użycia

Aktor jest, w języku UML, tzw. klasyfikatorem, czyli swego rodzaju klasą, która reprezentuje wiele podobnych obiektów z otoczenia modelowanego systemu. W rolę danego aktora mogą się wcielać różne osoby czy systemy. Jednocześnie, dana osoba czy system może pełnić role różnych aktorów. Najczęściej aktor stanowi określone grupy użytkowników systemu, którzy w jednakowy sposób z nim się komunikują. Aktorzy wchodzą w bezpośrednią interakcję z systemem, np. poprzez określony interfejs użytkownika tego systemu.



Rysunek 6.8: Hierarchia dziedziczenia aktorów

Modelując aktorów, możemy, podobnie jak w przypadku klas, tworzyć hierarchię dziedziczenia, gdzie wyróżniamy aktorów najbardziej ogólnych oraz bardziej szczegółowych, mających cechy aktorów ogólnych oraz swój specyficzny zestaw cech. Każdy aktor może uczestniczyć w określonych przypadkach użycia. Jeżeli aktor dziedziczy od innego aktora, to znaczy, że posiada jego uprawnienia do korzystania z określonych przypadków użycia oraz, dodatkowo, posiada własny zbiór uprawnień.

Uczestnictwo aktora w przypadku użycia oznaczamy relacją asocjacji. Relacja ta może być ukierunkowana, czyli oznaczona strzałką (Rysunek 6.7). Ukierunkowanie w stronę przypadku użycia oznacza, że dany aktor uruchamia przypadek użycia oraz oczekuje odpowiedniego rezultatu wykonania tego przypadku. Takiego aktora nazywamy aktorem głównym danego przypadku użycia.

Ukierunkowanie w stronę aktora oznacza, że system realizujący przypadek użycia musi korzystać z usług aktora. Takiego aktora nazywamy aktorem pomocniczym dla danego przypadku użycia.

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 oczekiwania. Dzięki temu, twórcy systemu mogą szybko uzyskać informację zwrotną od użytkownika, dotyczącą poziomu akceptacji dla budowanego systemu.

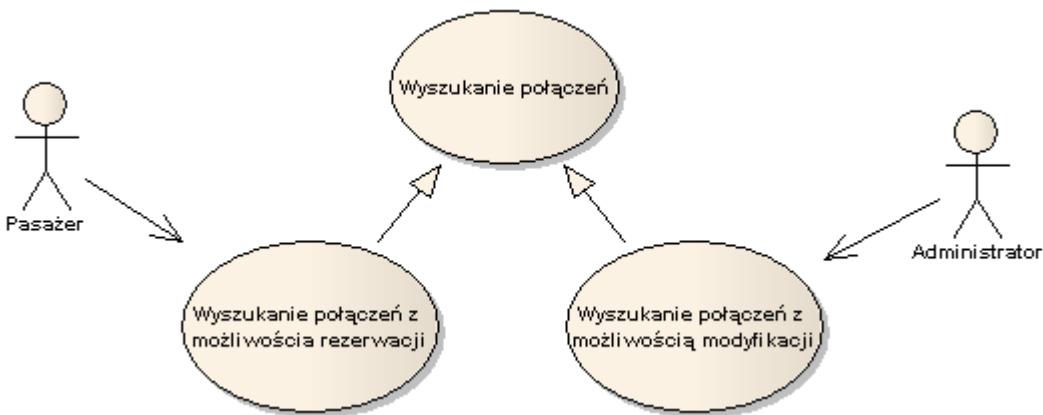
Przypadek użycia jest definiowany w języku UML jako ciąg akcji wykonywanych przez system, które dostarczają określony rezultat dla jednego z aktorów. W notacji modelu przypadków użycia, przypadek jest oznaczany jako elipsa z nazwą przypadku w środku lub pod elipsą (Rysunek 6.7). Opis przypadku użycia nie zawiera żadnych odniesień do wewnętrznej struktury systemu. Jest jedynie opisem interakcji między aktorem a systemem, który może obejmować różne warianty zachowania systemu. W interakcji tej, ważny jest sposób porozumiewania się aktora z systemem oraz opis czynności wykonywanych przez system w reakcji na interakcje aktora. Taki opis interakcji nazywany jest scenariuszem przypadku użycia.

Dla oznaczenia granicy działania systemu, na diagramie przypadków użycia możemy umieścić obrzeże (ang. *boundary*). Oznaczane jest ono prostokątną ramką z nazwą (Rysunek 6.7). Symbolizuje ono system, który realizuje przypadki użycia. Przypadki użycia realizowane przez system umieszcza się wewnątrz obrzeża, a elementy spoza systemu – na zewnątrz. Obrzeże umieszcza się na diagramie w celu zwiększenia czytelności. Jeśli wszystkie przypadki użycia na diagramie należą do tego samego systemu, to obrzeże może być pominięte.

6.2.2. Relacje między przypadkami użycia

Przypadki użycia mogą być połączone różnego rodzaju relacjami.

Ponieważ przypadek użycia jest klasifikatorem, możliwa jest relacja generalizacji między przypadkami użycia (Rysunek 6.9). Specyfikacja języka UML nie preczytuje jednakże znaczenia tej relacji dla przypadków użycia. Możemy przyjąć, że relacja ta oznacza poszerzenie funkcjonalności o pewne kroki lub scenariusze niewystępujące w ogólnym przypadku użycia. Przypadek użycia, który dziedziczy z przypadku ogólnego, musi określić, jakie nowe kroki lub scenariusze dodaje do przypadku ogólnego.

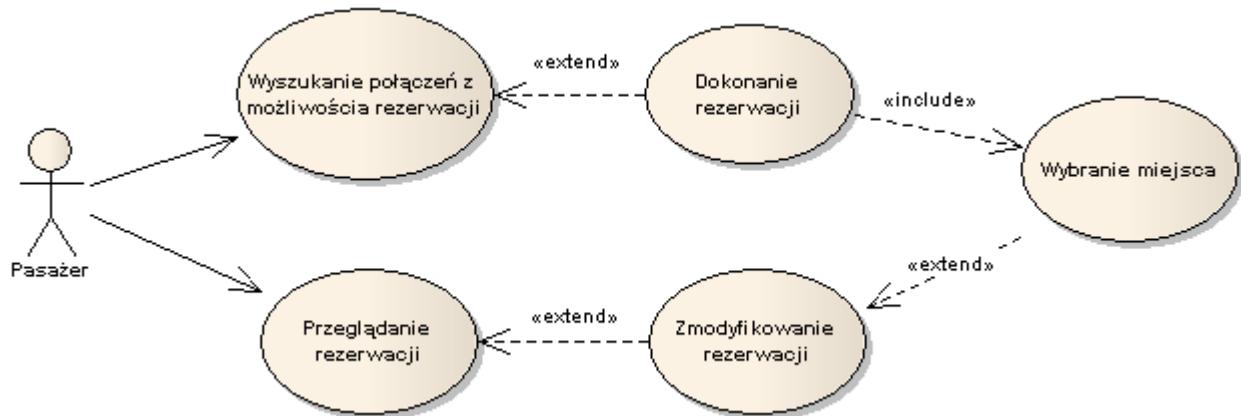


Rysunek 6.9: Relacje generalizacji między przypadkami użycia

Oprócz relacji generalizacji, możliwe jest umieszczenie między dwoma przypadkami użycia relacji rozszerzenia oraz włączenia. Obydwie relacje oznaczane są podobnie jak zależność – przerywaną strzałką. Dodatkowo relacja rozszerzenia oznaczona jest stereotypem «extend», natomiast relacja włączenia – stereotypem «include» (Rysunek 6.10).

Relacja włączenia oznacza, że wszystkie scenariusze włączanego przypadku użycia są umieszczane w odpowiednim miejscu przypadku użycia włączającego. Miejsce włączenia zaznaczone jest dokładnie w scenariuszu przypadku użycia.

Relacja rozszerzenia oznacza, że scenariusze rozszerzającego przypadku użycia włączane są pod pewnym warunkiem w odpowiednim miejscu przypadku użycia rozszerzanego. Miejsce rozszerzenia oraz warunek rozszerzenia umieszczamy w scenariuszu przypadku rozszerzanego.



Rysunek 6.10: Relacje włączania i rozszerzania między przypadkami użycia

6.2.3. Scenariusze przypadków użycia

Dobrą definicję scenariusza podał Alistair Cockburn w książce p.t. „Jak pisać efektywne przypadki użycia”:

„Scenariusz jest sekwencją interakcji dzierżących się pod pewnymi warunkami, aby osiągnąć cel aktora głównego, oraz mających określony rezultat z punktu widzenia tego celu. Interakcje rozpoczynają się od akcji uruchamiającej i są kontynuowane aż do osiągnięcia celu lub do wystąpienia niepowodzenia, oraz dopóki system nie wykona wszystkich swoich zadań z punktu widzenia interakcji.”

Powyższa definicja wskazuje na możliwość wystąpienia scenariuszy, które nie osiągają celu założonego przez aktora w momencie ich uruchamiania. Oznacza to, że wokół istotnego celu dla aktora, możemy zgrupować więcej niż jeden scenariusz. Takie scenariusze mogą się kończyć osiągnięciem tego celu, ale również mogą zakończyć się niepowodzeniem. Taką grupę scenariuszy nazywamy przypadkiem użycia systemu.

Przypadek użycia możemy zdefiniować na bazie podanej definicji scenariusza:

„Przypadek użycia jest zbiorem możliwych scenariuszy opisujących dialog między analizowanym systemem a zewnętrznymi aktorami, których główną cechą jest cel aktora w stosunku do zadeklarowanego zakresu obowiązków systemu, pokazujących jak ten cel może być osiągnięty lub nie.”

W języku UML, przypadek użycia jest klasyfikatorem, co oznacza, że ma swoje instancje. Specyfikacja języka nie wyjaśnia jednak, czym są te instancje przypadku użycia. Przyjmijmy zatem, że instancjami są scenariusze przypadku użycia. Przypadek użycia jest więc pewną klasą grupującą obiekty będące scenariuszami prowadzącymi do realizacji tego samego celu.

Wałączącą cechą przypadku użycia jest kompletność. Kompletność określa, że przypadek użycia powinien zawsze rozpoczynać się interakcją aktora z systemem. Ponadto, wszystkie scenariusze przypadku użycia powinny zawsze prowadzić do stabilnego stanu końcowego systemu. W stanie stabilnym nie są już konieczne żadne dalsze interakcje lub czynności i możliwe jest ponowne uruchomienie tego przypadku użycia.

Definicja przypadku użycia mówi, że jest on zbiorem scenariuszy prowadzących do tego samego celu. Niektóre przypadki użycia będą posiadały tylko jeden scenariusz, prowadzący do osiągnięcia celu. W większości przypadków, musimy jednak opisać różnego rodzaju scenariusze alternatywne, które mogą powodować, że podstawowy cel przypadku użycia nie zostanie osiągnięty.

Scenariusze składają się z ponumerowanych zdań opisujących kolejne działania aktora bądź systemu. W scenariuszach alternatywnych numery zdań opatrzone są dodatkowo kolejnymi literami alfabetu. Wszelkie rozgałęzienia warunkowe oznaczane są znakiem „==>”. Tym samym zna-

kiem oznaczane są miejsca w scenariuszu, w których możliwe jest rozszerzenie scenariusza o inne przypadki użycia (relacja «extend»), czyli punkty rozszerzenia. Podobnie oznaczane są miejsca, w których wstawiane są przypadki użycia (relacja «include»).

Same zdania pisane są w bardzo prostej gramatyce. Składają się z podmiotu, orzeczenia i dopełnienia. Niektóre zdania zawierają dodatkowo dopełnienie dalsze. Tak proste zdania są wystarczające do tworzenia scenariuszy, pod warunkiem, że wszystkie definicje pojęć (dopełnień) są definiowane poza scenariuszami, np. w oddzielnym słowniku pojęć. Takie podejście pozwala lepiej kontrolować spójność scenariuszy. Wprowadzając do scenariusza jakieś pojęcie, definiujemy go od razu w słowniku i we wszystkich kolejnych scenariuszach używamy dokładnie tego samego pojęcia.

Poniżej znajdują się scenariusze dla przypadku użycia „Wyszukanie połączeń z możliwością rezerwacji”. Pierwszy ze scenariuszy jest scenariuszem głównym, który dostarcza aktorowi pożądany wynik, czyli wyświetlenie listy połączeń zgodnych z zadanymi kryteriami. Na końcu tego scenariusza znajduje się punkt rozszerzenia – klient może uruchomić przypadek użycia „Dokonanie rezerwacji” dla wybranego połączenia. Drugi scenariusz jest scenariuszem alternatywnym. Kroki od 1 do 4 są identyczne jak w scenariuszu głównym. Różnica polega na tym, że system w tym przypadku nie znajduje połączeń zgodnych z kryteriami klienta – musi on ponownie wprowadzić kryteria.

Wyszukanie połączeń z możliwością rezerwacji – scenariusz główny:

1. *Klient wybiera opcję wyszukiwania połączeń.*
2. *System wyświetla okno wyszukiwania połączeń.*
3. *Klient wprowadza kryteria wyszukiwania połączeń.*
4. *System wyszukuje połączenia.*
[istnieją połączenia spełniające kryteria]
5. *System wyświetla listę połączeń.*
«extend» Dokonanie rezerwacji

Wyszukanie połączeń z możliwością rezerwacji – scenariusz alternatywny:

1. *Klient wybiera opcję wyszukiwania połączeń.*
2. *System wyświetla okno wyszukiwania połączeń.*
3. *Klient wprowadza kryteria wyszukiwania połączeń.*
4. *System wyszukuje połączenia.*
[brak połączeń spełniających kryteria]
5. *System wyświetla komunikat o braku połączeń.*
Powrót do punktu 2.

6.3. Model czynności

Model czynności to jeden z modeli, jakie język UML udostępnia w celu opisu dynamicznych aspektów systemu. Czynności modelujemy na diagramach czynności. Diagram taki jest pewnego rodzaju schematem blokowym, który przedstawia przepływ sterowania od czynności do czynności. Diagram czynności opisuje, jak są uszeregowane działania oraz daje możliwość opisu czynności warunkowych i współbieżących

Diagramy czynności mają dosyć szerokie zastosowanie w modelowaniu dynamiki systemu. Możliwe zastosowania to m.in.: modelowanie procesów biznesowych, modelowanie scenariuszy przypadków użycia, opisywanie złożonych algorytmów sekwencyjnych czy modelowanie aplikacji wielowątkowych.

6.3.1. Węzły czynności i przepływy sterowania

Diagramy czynności są pewnego rodzaju grafami zawierającymi węzły czynności (ang. *activity node*) oraz przepływy sterowania. Węzły czynności możemy podzielić na akcje (ang. *action*) oraz węzły sterujące (ang. *control node*). Notacja dla akcji to prostokąt z zaokrąglonymi rogami. Węzłami sterującymi, które są obecne na każdym diagramie, są węzły początkowe oraz węzły końcowe. Węzeł początkowy oznaczamy za pomocą czarnego koła, a węzeł końcowy za pomocą okręgu z czarną kropką w środku. Przepływy sterowania między węzłami, oznaczane są za pomocą strzałek skierowanych od jednego węzła do drugiego. Na przepływie sterowania można umieścić warunek, ujęty w nawiasy prostokątne. Każda akcja ma nazwę, zawartą wewnątrz jej symbolu. Nazwa akcji jest określeniem wykonywanej operacji. Nazwa ta może być dowolnym tekstem. Na poziomie bardziej abstrakcyjnym może to być np. zdanie scenariusza, a na poziomie bardziej konkretnym może to być np. wyrażenie w określonym języku programowania. Węzły początkowe i końcowe również mogą mieć nazwę. Wszystkie wymienione elementy pokazane są na poniższym przykładowym diagramie (Rysunek 6.11).



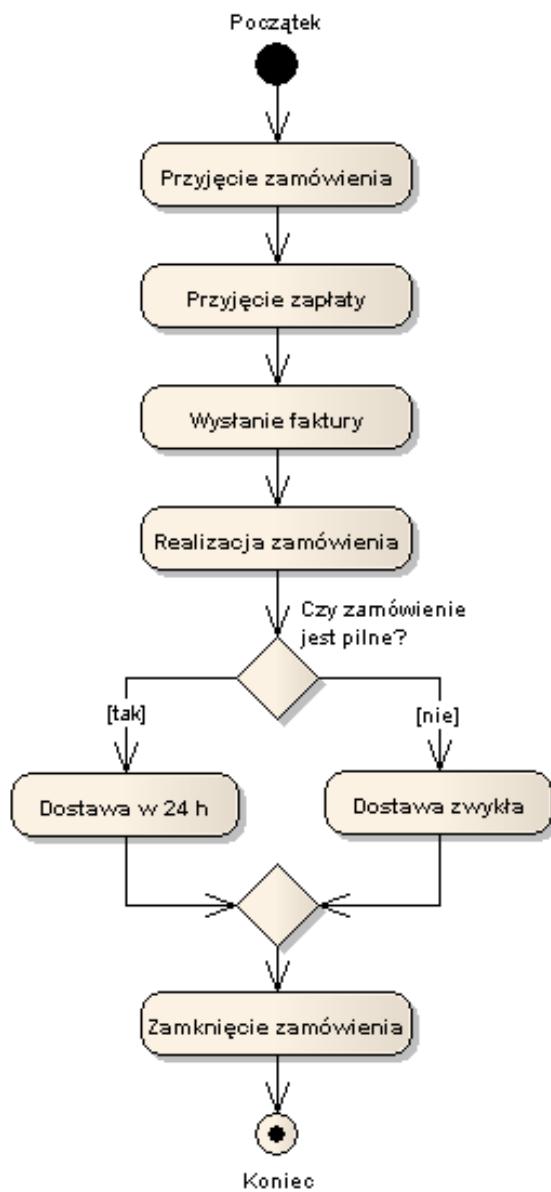
Rysunek 6.11: Diagram czynności

Elementarnymi jednostkami zachowania się na diagramach czynności są akcje. Węzły sterujące i przepływy sterowania zarządzają natomiast kolejnością wykonywania akcji. Węzeł początkowy oznacza początek sterowania w ramach całej czynności. W momencie rozpoczęcia działania czynności, sterowanie umieszczane jest właśnie w węźle początkowym. Następnie sterowanie przenoszone jest do akcji, które połączone są z węzłem początkowym przepływami sterowania. Przeniesienie sterowania do następnej akcji następuje po zakończeniu wykonywania akcji bieżącej. Jeżeli z akcji wychodzi więcej niż jeden przepływ sterowania, to powinny być one opatrzone odpowiednimi warunkami. Z jednej akcji możliwe jest przejście do wykonania tylko jednej kolejnej akcji. O tym, do której akcji przejść, decydują warunki określone dla odpowiednich przepływów sterowania. Należy tak określić warunki dla przepływów sterowania wychodzących z jednej akcji,

aby były rozłączne. Oznacza to, że tylko jeden z warunków powinien być w danej chwili prawdziwy. Wykonywanie kolejnych akcji kończy się w momencie osiągnięcia węzła końcowego.

6.3.2. Decyzje i scalenia

Bardzo często zachodzi potrzeba zamodelowania na diagramach czynności alternatywnych ścieżek przepływu sterowania, które wynikają ze spełnienia określonych warunków. Do ukazywania warunkowych rozgałęzień na diagramach czynności, używa się specjalnych węzłów sterujących: węzła decyzyjnego (ang. *decision*) oraz węzła scalenia (ang. *merge*). Węzeł decyzji ma jeden przychodzący przepływ sterowania oraz kilka przepływów wychodzących. Węzeł scalenia może mieć wiele przychodzących przepływów sterowania i jeden wychodzący. Oba te węzły oznaczane są za pomocą rombu. Rysunek 6.12 pokazuje przykładowy diagram zawierający węzły decyzyjne oraz scalenia.



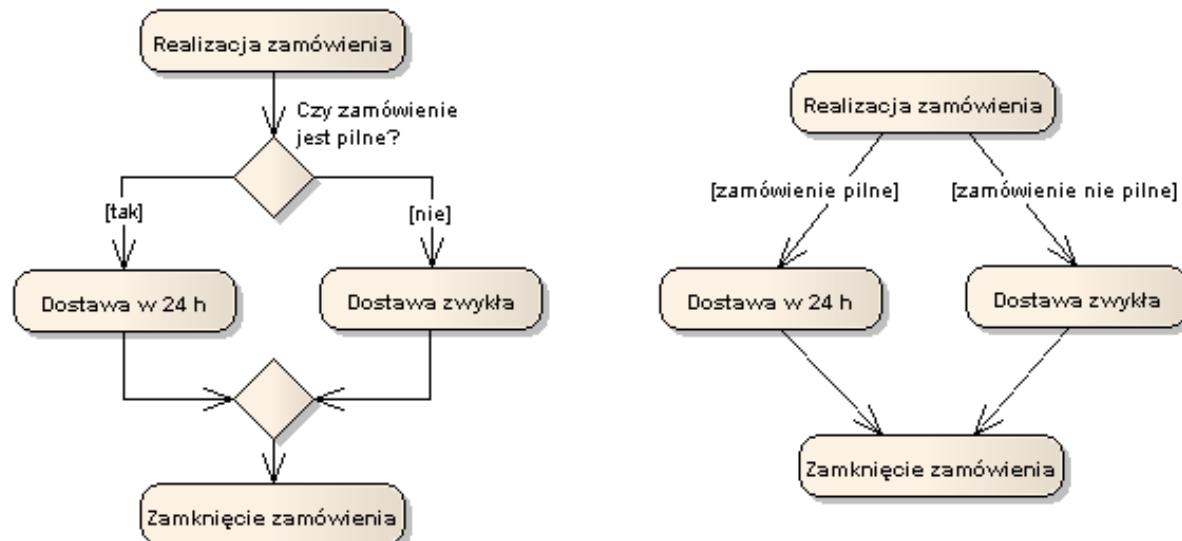
Rysunek 6.12: Decyzje i scalenia na diagramie czynności

Działanie węzła decyzji polega na przekazaniu sterowania z przepływu przychodzącego do jednego przepływu wychodzącego. W węźle podejmowana jest decyzja, do którego przepływu wychodzącego przekazać sterowanie. Decyzja ta jest podejmowana na podstawie wartości logicznej warunków przypisanych do przepływów wychodzących. Należy zadbać, aby warunki były wzajem-

nie rozłączne, co oznacza, że w danej sytuacji tylko jeden z tych warunków jest spełniony, czyli przyjmuje wartość logiczną „prawda”.

Węzeł scalenia przekazuje sterowanie z jednego z przepływów przychodzących do przepływu wychodzącego.

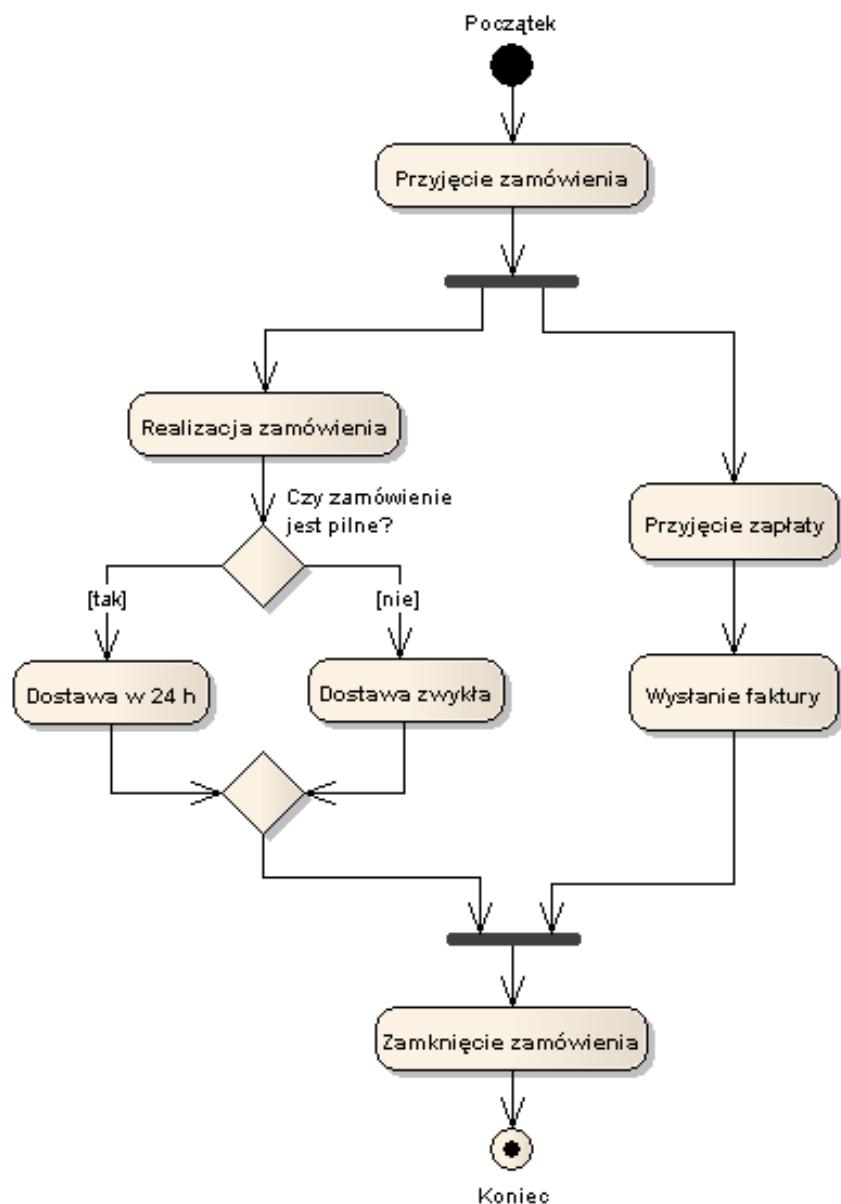
W niektórych sytuacjach możliwe jest pominięcie węzłów decyzyjnych i scalenia, np. w celu uproszczenia diagramu. W takiej sytuacji, zamiast węzłów stosujemy akcje z wieloma przepływami wychodzącymi oraz akcje z wieloma przepływami wchodzącymi. Przykład takiej notacji pokazany jest na poniższym rysunku (Rysunek 6.13).



Rysunek 6.13: Alternatywna notacja dla rozgałęzienia warunkowego

6.3.3. Rozwidlenia i złączenia

Oprócz warunkowych rozgałęzień, zachodzi czasem potrzeba ukazania wspólniego przepływu sterowania, np. podczas modelowania procesów zachodzących w przedsiębiorstwach. Do modelowania procesów równoległych na diagramach czynności wykorzystujemy specjalne węzły sterujące zwane belkami synchronizacji (ang. *synchronization bar*). Wyróżniamy dwa rodzaje belek synchronizacji: rozwidlenia (ang. *fork*) oraz złączenia (ang. *join*). Obydwa węzły oznaczane są na diagramach za pomocą belki ustawionej poziomo albo pionowo. Rozwidlenie ma zawsze jeden wchodzący przepływ sterowania i jeden lub więcej wychodzących. Złączenia może mieć natomiast wiele wchodzących przepływów sterowania i jeden wychodzący. Rysunek 6.14 pokazuje przykładowy diagram zawierający belki rozwidlenia i synchronizacji.



Rysunek 6.14: Rozwidlenia i złączenia na diagramie czynności

W rozwidleniu sterowanie jest kopiwane i przekazywane do wszystkich wychodzących przepływów sterowania. W ten sposób sterowanie płynie jednocześnie w kilku miejscach - operacje na drodze tych równoległych przepływów wykonywane są wspólnie. Jeżeli przepływ wychodzący ma przypisany warunek, to sterowanie przekazywane jest przez ten przepływ tylko w przypadku spełnienia warunku.

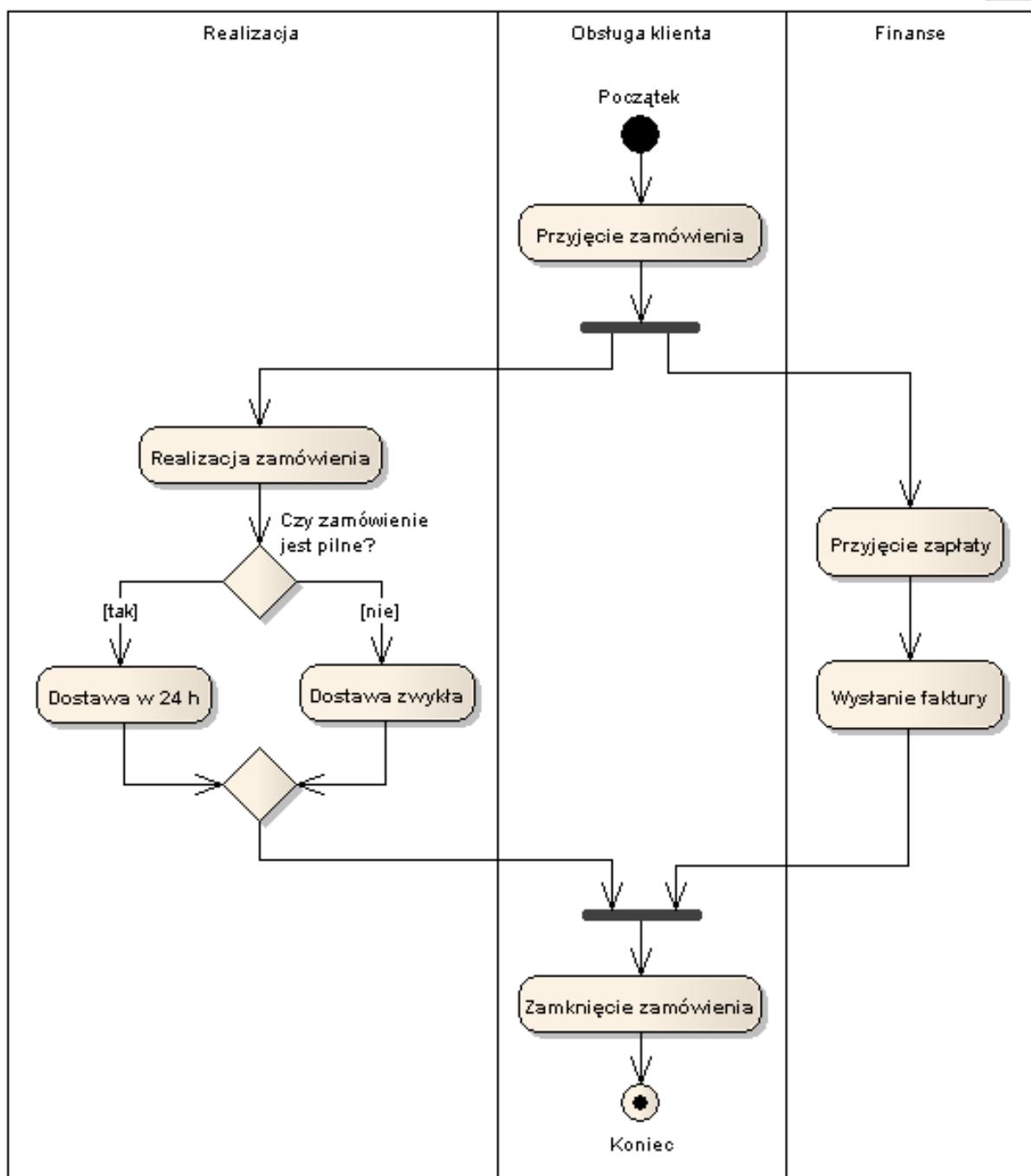
Złączenie czeka na pojawienie się sterowania na wszystkich wchodzących przepływiech. Dopiero w momencie, gdy przepływy sterowania ze wszystkich równoległych ścieżek dojdą do złączenia, są one scalane i sterowanie przekazywane jest do przepływu wychodzącego z belki złączenia.

Rozwidlenia i złączenia powinny się równoważyć, to znaczy liczba przepływów opuszczających rozwidlenie powinna być równa liczbie przepływów wchodzących do odpowiadającego mu złączenia.

6.3.4. Tory

Diagramy czynności opisują, co się dzieje, ale nie mówią, kto wykonuje określone akcje. Aby przyporządkować akcje na diagramie czynności do ich wykonawców można zastosować tory (ang. *swimlane*). Są one bardzo przydatne zwłaszcza w modelowaniu procesów biznesowych. Topy

stanowią kolumny lub wiersze na diagramach czynności wyznaczone ciągłymi liniami. Każdy tor ma nazwę, która odpowiada najczęściej nazwie obiektu lub aktora, który jest odpowiedzialny za wykonanie akcji umieszczonych w danym torze. Każda akcja powinna należeć tylko do jednego toru, ale przepływy sterowania mogą przecinać granice torów. Przykład diagramu z torami przedstawia poniższy rysunek (Rysunek 6.15).



Rysunek 6.15: Tory na diagramie czynności

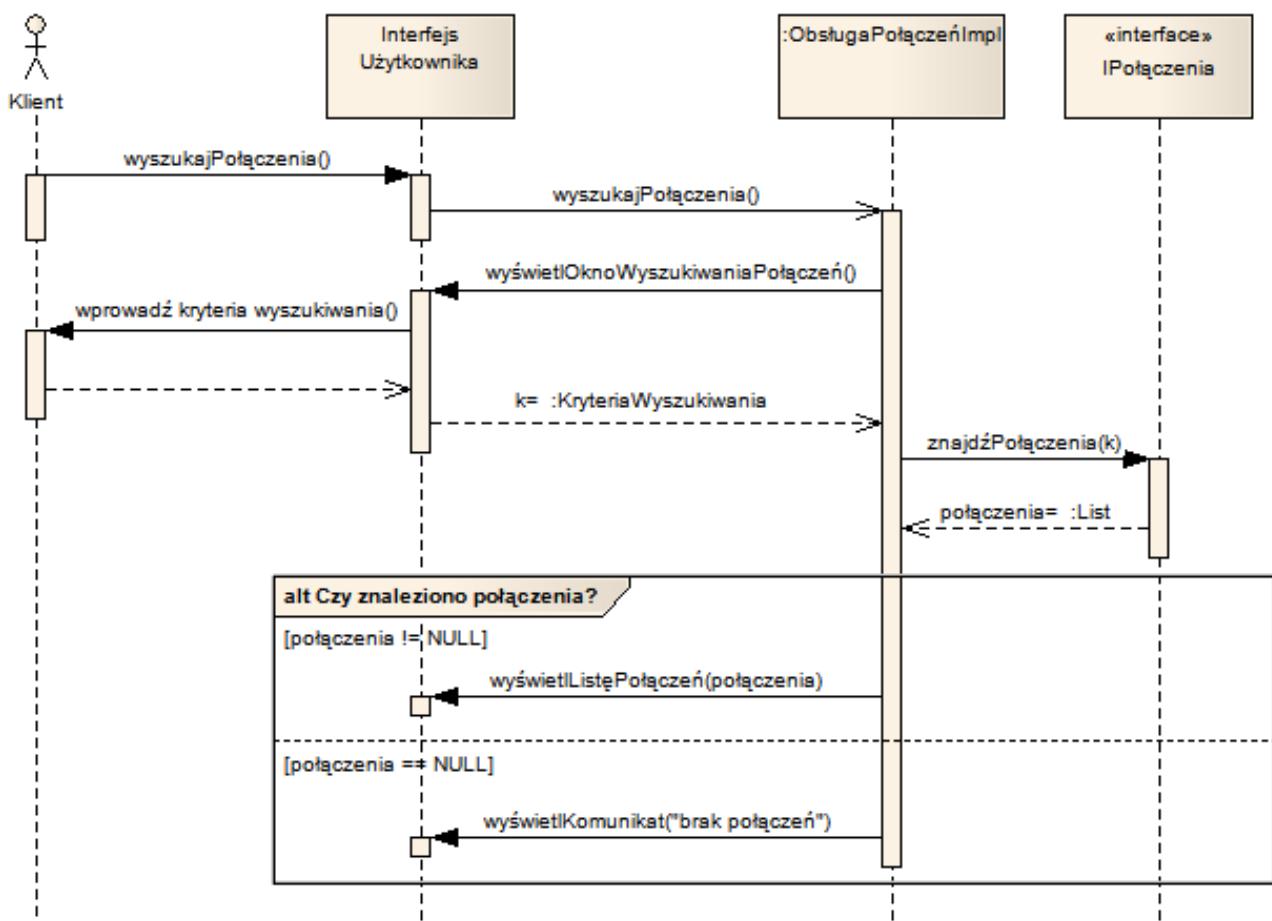
6.4. Modele interakcji

Modele interakcji służą do opisywania komunikacji pomiędzy elementami systemu. Jak wiemy, do tworzenia modelu interakcji można w języku UML wykorzystywać kilka różnych diagramów. Spśród diagramów interakcji, najpowszechniej używanym jest diagram sekwencji, którego opis znajduje się poniżej.

6.4.1. Diagram sekwencji

W języku UML, podstawowym sposobem na pokazanie dynamicznej wymiany komunikatów między obiektami są diagramy sekwencji. Diagramy te doskonale nadają się do pokazywania wymiany komunikatów w czasie. Komunikaty zaznaczane są poziomymi strzałkami skierowanymi od nadawcy do odbiorcy komunikatu. Sekwencja komunikatów czytana jest od góry do dołu, odzwierciedlając upływ czasu.

Podstawową diagramów sekwencji są pionowe kolumny zwane liniami życia (ang. *lifeline*). Każda linia życia odpowiada pojedynczemu obiektovi uczestniczącemu w sekwencji. Linia życia rozpoczyna się od góry symbolem reprezentującym obiekt. Od obiektu biegnie pionowo w dół przerywana linia oznaczająca czas życia tego obiektu. Najczęściej obiekty żyją przez cały czas trwania interakcji, czyli linia obejmuje cały diagram od góry do dołu. Linie życia mogą reprezentować obiekty różnego rodzaju. Mogą to być klasy, interfejsy, komponenty lub aktorzy. Jeżeli diagram sekwencji ma prezentować sposób działania jakiegoś podsystemu, najpewniej będzie on zawierał jedynie obiekty odpowiednich klas. Jeżeli zechcemy opisać diagramem sekwencji realizację przypadku użycia, to co najmniej jedna linia życia powinna odpowiadać aktorowi.



Rysunek 6.16: Diagram sekwencji

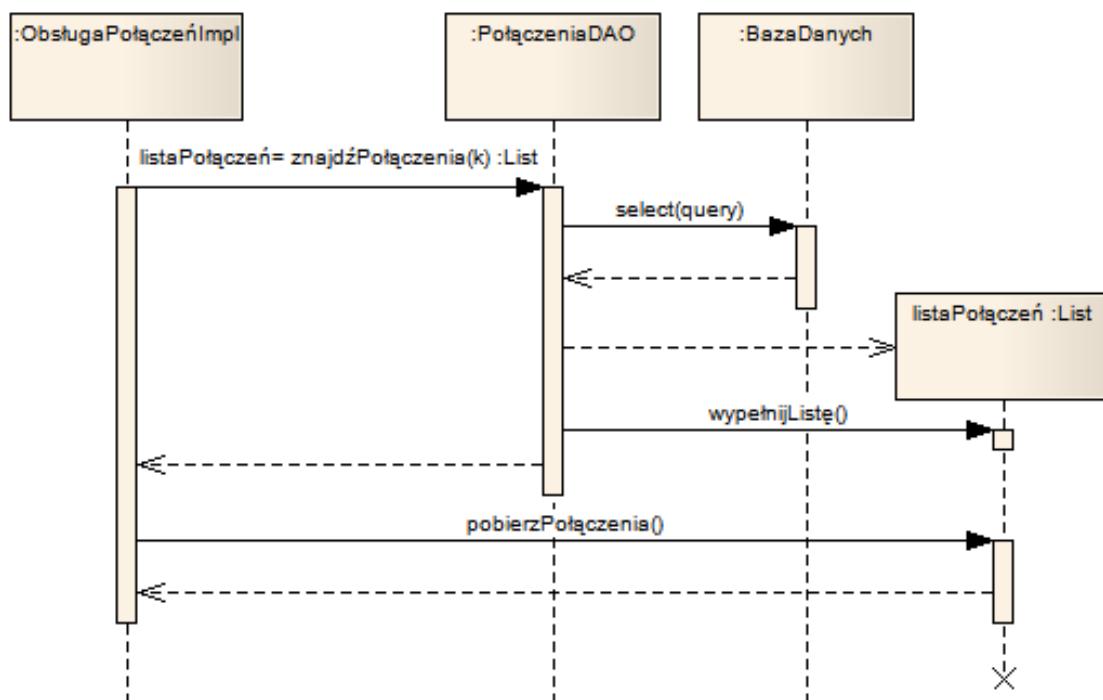
Komunikaty rysowane są na diagramie sekwencji między jedną linią życia a drugą. Komunikat ma najczęściej nazwę. Nazwa ta zazwyczaj odpowiada nazwie operacji zawartej w klasie związanej z linią życia odbiorcy komunikatu. Obsługa komunikatu, czyli wykonanie usługi przez odbiorcę, zaznaczana jest pionową belką umieszczoną wzdłuż linii życia. Belka kończy się w momencie zakończenia wykonywania usługi. Rysunek 6.16 przedstawia przykładowy diagram sekwencji obrazujący wykonanie przypadku użycia na poziomie architektonicznym – komunikaty wymieniane są pomiędzy interfejsami.

Mogą wyróżnić dwa rodzaje komunikatów: synchroniczne i asynchroniczne. Komunikaty synchroniczne zaznaczane są strzałkami z pełnym grotem. Dla komunikatu synchronicznego belka na linii

życia wyznacza czas, w którym wykonywany jest kod operacji obsługującej ten komunikat. Po zakończeniu obsługi komunikatu, sterowanie powraca do obiektu, który był nadawcą komunikatu. Powrót sterowania oznacza się komunikatem zwrotnym w postaci przerywanej strzałki. Komunikat zwrotny może być opisany np. wynikiem wykonania operacji.

Komunikaty asynchroniczne zaznaczane są za pomocą strzałek z otwartymi grotami. Komunikat asynchroniczny oznacza, że jego przesłanie powoduje uruchomienie działania operacji niewymagającej na końcu synchronizacji z obiektem wywołującym tę operację, tzn. nadawca komunikatu nie oczekuje biernie na zakończenie wykonywania operacji związanej z tym komunikatem. Komunikaty asynchroniczne nie wymagają więc przesyłania komunikatów powrotnych przez odbiorcę.

Na diagramach sekwencji możliwe jest zaznaczenie utworzenia (konstrukcji) nowego obiektu. W tym celu należy umieścić na diagramie odpowiedni komunikat tworzący obiekt. Oznaczany on jest strzałką z otwartym grotem, przy czym konstruowany obiekt umieszczony jest na linii życia na poziomie komunikatu tworzącego obiekt. Destrukcja obiektu, oznaczająca usunięcie go z pamięci, rysowana jest w postaci znaku X umieszczonego na końcu linii życia obiektu. Rysunek 6.17 przedstawia przykład tworzenia i destrukcji obiektu.



Rysunek 6.17: Tworzenie i destrukcja obiektu

6.4.2. Fragmenty w diagramach sekwencji

Bardzo przydatnymi elementami, które możemy wykorzystywać na diagramach sekwencji są tzw. fragmenty włączone (ang. *combined fragment*). Fragmenty takie umieszcza się na diagramach w postaci ramki z nagłówkiem, która obejmuje pewien zbiór komunikatów. Fragment włączony oznacza pewien fragment diagramu sekwencji stanowiący osobną interakcję wykonywaną w określony sposób pod określonymi warunkami. W nagłówku ramki określony jest typ fragmentu włączonego oraz, ewentualnie, jego nazwa. Wewnątrz fragmentu umieszczany jest jeden lub więcej warunków. Jeżeli warunków jest kilka, to cały fragment podzielony jest na tyle części, ile jest warunków. Części oddzielone są poziomą przerywaną linią.

Istnieje wiele typów fragmentów włączonych. Najważniejsze z nich są następujące:

- „alt” (alternatywa) – zawiera dwie lub więcej części z warunkami rozłącznymi. Wykonywana jest interakcja zawarta w tej części, dla której warunek jest spełniony. Fragment ten działa na podobnej zasadzie jak instrukcja „if-else” w większości języków programowania. Rysunek 6.16 przedstawia przykład tego typu ramki.

- „opt” (opcja) – zawiera dokładnie jedną część opisaną warunkiem. Interakcja zawarta w tej części jest wykonywana wtedy, gdy warunek jest spełniony. Odpowiada to pojedynczej instrukcji „if” w języku programowania.
- „loop” (pętla) – zawiera dokładnie jedną część z warunkiem. Wykonywanie sekwencji komunikatów zawartej w tej części powtarzane jest dotąd, aż zostanie spełniony warunek zakończenia pętli. Warunek może zawierać liczbę określającą ilość iteracji lub warunek logiczny zakończenia pętli. Fragment ten działa na podobnej zasadzie jak instrukcje „for” czy „while” spotykane w różnych językach programowania.
- „break” (przerwanie) – zawiera jedną lub więcej części z warunkami. Po wykonaniu interakcji zawartej w części, dla której spełniony jest warunek, przerywane jest wykonywanie całej sekwencji w ramach aktualnej interakcji. Odpowiada to instrukcji „break” występującej w większości języków programowania.
- „par” (zrównoleglenie) – zawiera kilka części, które mogą być wykonywane równolegle. W ramach takiego wykonania może następować przeplatanie komunikatów między sekwencjami zawartymi w różnych częściach.

6.5. Podsumowanie

O ile w modelu struktury pokazujemy aspekty statyczne, to model dynamiki pokazuje system w działaniu. Wszystkie przedstawione w tym rozdziale diagramy, mogą być wykorzystywane podczas modelowania dynamicznych aspektów systemów oprogramowania. Model dynamiki jest o tyle istotny, że podczas budowy oprogramowania staramy się zrealizować dynamiczną funkcjonalność systemu odpowiadającą wymaganiom zamawiających. Dobry model dynamiki przekłada się na realizację systemu o wysokiej jakości mierzonej poziomem zadowania zamawiającego.

Podobnie jak w przypadku diagramów statycznych, nie zawsze jest potrzeba stosowania podczas projektu wszystkich rodzajów diagramów dynamicznych. Niektóre z nich używane są jedynie w specyficznych sytuacjach. Np. diagram następstwa będzie najczęściej wykorzystywany do modelowania systemów czasu rzeczywistego. W przypadku większości systemów biznesowych, wystarczają jedynie podstawowe diagramy, takie jak diagramy przypadków użycia, diagramy czynności oraz diagramy sekwencji.

7. Podstawy inżynierii wymagań

7.1. Co to są wymagania i na czym polega inżynieria wymagań?

Inżynieria wymagań jest dziedziną inżynierii oprogramowania zajmującą się pozyskiwaniem, analizą, specyfikacją i walidacją wymagań wynikających z rzeczywistych potrzeb przyszłych użytkowników. Dziedzina ta jest pomostem łączącym świat biznesu (ogólnie: rzeczywistość dotyczącą zadaneego problemu) i inżynierii oprogramowania. Te dwa światy posługują się zupełnie różnymi językami a wzajemne zrozumienie jest kluczem do osiągnięcia sukcesu przez projekt. Klient z reguły nie rozumie procesu powstawania oprogramowani, za to posiada wiedzę z dziedziny, którą zajmuje się projektowany system. Twórcy oprogramowania nie zawsze w pełni rozumieją procesy zachodzące w środowisku, które ma odzwierciedlać system. Zadaniem inżynierii wymagań jest pozyskanie wiedzy, stworzenie modelu środowiska, pozyskanie lub odkrycie potrzeb użytkownika a następnie sformułowanie ich w sposób zrozumiały dla twórców oprogramowania. Efektem końcowym procesu pozyskiwania, analizy i modelowania wymagań powinna być specyfikacja jasno określająca zakres funkcjonalności systemu (swoista umowa pomiędzy zamawiającym a twórcą oprogramowania), pozwalająca na realizację systemu spełniającego oczekiwania użytkowników.

Kluczową rolę pełnią tu pozyskiwanie i analiza wymagań. Proces pozyskiwania wymagań służy określeniu, co jest potrzebne w systemie i dlaczego. Użytkownicy mogą sami nie rozumieć, czego tak naprawdę potrzebują. Zadaniem pozyskiwania wymagań jest poznanie istniejących (lub odkrycie nowych) procesów zachodzących w modelowanym środowisku oraz rzeczywistych potrzeb użytkownika stawianych przed systemem. Analiza wymagań jest natomiast procesem służącym zrozumieniu pozyskanych i pozyskiwanych wymagań.

Czym jest wymaganie?

Wymaganie jest własnością 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.

Wymagania odpowiadają na najistotniejsze w całym procesie budowy systemu pytanie: jaki system mamy zbudować? W dobrze zorganizowanym procesie wytwórczym, wymagania sterują konstrukcją systemu. Dobrze skonstruowany system spełnia wszystkie testy oparte na wymaganiach.

Poprawnie sformułowane wymagania powinny wynikać z rzeczywistych potrzeb zamawiającego, być kompletne, być niesprzeczne ze sobą, być jednoznaczne (definiować jeden możliwy system, a nie kilkadziesiąt różnych; być niezależne od interpretacji), być testowalne i mierzalne (powinny dać się przetestować w sposób jednoznaczny). Spełnienie tak określonych wymagań oznacza zadowolonego zamawiającego.

Więcej na temat jakości wymagań w rozdziale „Cechy dobrej specyfikacji wymagań. W dalszych rozważaniach będziemy się koncentrować na wymaganiach dla systemów biznesowych. Jednakże, praktycznie wszystkie informacje o wymaganiach dla systemów biznesowych można odnieść również do systemów innego rodzaju (symulacje zjawisk fizycznych, systemy czasu rzeczywistego itd.).

Źródła wymagań

W trakcie procesu pozyskiwania wymagań możemy zidentyfikować różne źródła wymagań. Podstawowym źródłem wymagań są zamawiający i przyszli użytkownicy systemu. Zamawiający, jako osoby decydujące o procesach biznesowych w organizacji i zakresie systemu są kluczowi dla

określenia zakresu systemu (tym samym kosztu projektowanego systemu). Przyszli użytkownicy, jako osoby bezpośrednio pracujące z systemem są źródłem wiedzy z dziedziny budowanego systemu oraz realnych potrzeb stawianych przed tym systemem.

Kolejnymi źródłami wymagań są istniejące w organizacji procesy biznesowe oraz modyfikacje tych procesów, które mają usprawnić działanie organizacji po wprowadzeniu nowo projektowanego systemu.

Projektowane systemy często zastępują istniejące systemu, dodając do nich nową funkcjonalność oraz modyfikując istniejącą. W takim przypadku podręczniki użytkownika istniejących systemu są doskonałym źródłem informacji na temat automatyzacji procesów w danej organizacji oraz tego, co można w tym obszarze usprawnić.

Odbiorcy wymagań

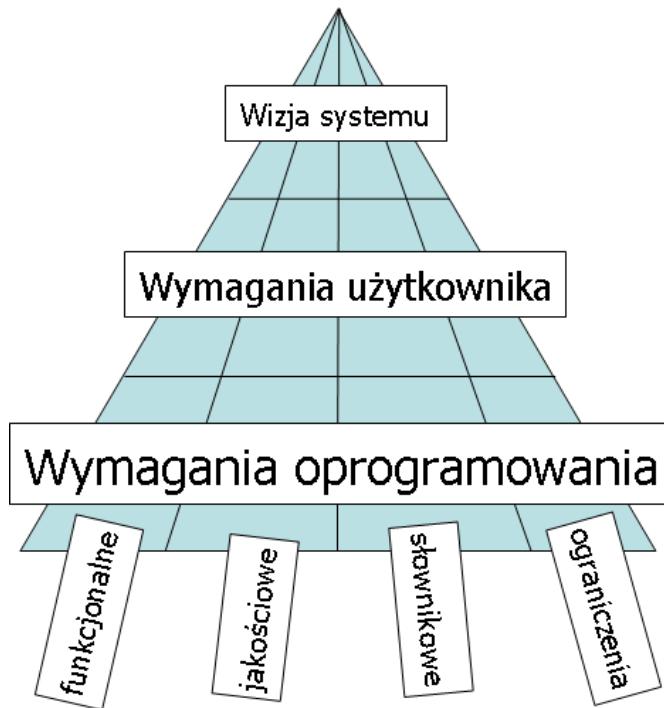
Tak jak istnieje wiele źródeł wymagań, podobnie wymagania mają różnych odbiorców. Z odbiorcami specyfikacji wymagań mamy do czynienia zarówno po stronie zamawiającego oprogramowanie jak i po stronie wykonawcy oprogramowania. Z punktu widzenia wykonawcy oprogramowania najważniejszą grupą odbiorców są architekci i projektanci systemu. Zadaniem tych osób jest przygotowanie architektury i projektu szczegółowego systemu realizującego funkcjonalność założoną w wymaganiach funkcjonalnych oraz spełniających ograniczenia i miary zdefiniowane w wymaganiach pozafunkcjonalnych. Dla odbiorców wymagań po stronie realizującego system ważna jest precyza i jednoznaczność. Wymagania muszą w sposób precyzyjny definiować funkcjonalność systemu oraz być jednoznaczne, aby można było je zinterpretować tylko w jeden możliwy sposób. Dobrze zdefiniowana specyfikacja wymagań jest też kluczowa dla kierowników projektów, ponieważ jednostki funkcjonalności odgrywają istotną rolę w planowaniu implementacji i zarządzaniu realizacją projektu. Kolejną grupą odbiorców są projektanci testów akceptacyjnych, których zadaniem jest zaprojektowanie testów w taki sposób, aby pozwalały one na zweryfikowanie czy gotowy produkt spełnia założoną funkcjonalność.

Wymagania kierowane są również do różnych grup odbiorców po stronie zamawiającego. Wymagania określają zakres budowanego systemu, tym samym stanowiąc kontrakt pomiędzy zamawiającym a wykonawcą. Wymagania muszą być czytelne dla zamawiającego, aby mógł on zweryfikować, czy zakładana funkcjonalność odpowiada rzeczywistym potrzebom oraz czy wykonawca wywiązał się ze swojej umowy (zrealizował funkcjonalność założoną przez specyfikację). Nie bez znaczenia są również przyszli użytkownicy systemu, którzy powinni brać czynny udział w specyfikowaniu wymagań na budowany system. To oni posiadają niezbędną wiedzę dziedzinową i oni są źródłem realnych potrzeb, jakie powinien zaspokoić budowany system.

Specyfikacja wymagań posiada na tyle szerokie grono odbiorców, że musi być zdefiniowana w sposób zrozumiałym dla różnych grup o różnym poziomie wiedzy technicznej bądź dziedzinowej. Z jednej strony specyfikacja powinna być zrozumiała dla „zwykłych ludzi” nie posiadających wiedzy technicznej, za to posiadających wiedzę z dziedziny, której dotyczy budowany system, z drugiej musi być zdefiniowana w sposób na tyle precyzyjny i jednoznaczny, aby twórcy systemu, nie posiadający wiedzy dziedzinowej, potrafili ją prawidłowo zinterpretować i na jej podstawie zbudować system zgodny z jej założeniami.

7.2. Podział wymagań

Wymagania możemy poklasyfikować według ich poziomu szczegółowości oraz ze względu na ich typ. Oba podziały przedstawia Rysunek 7-1. Dwa pierwsze poziomy „Piramidy” wymagań – wizja systemu i wymagania użytkownika – wspólnie stanowią wymagania zamawiającego. Charakteryzują się one mniejszym poziomem szczegółowości i stanowią kontrakt pomiędzy zamawiającym a wykonawcą określający zakres budowanego systemu. Dolną część piramidy stanowią wymagania oprogramowania. Charakteryzują się one dużo większym poziomem szczegółowości i stanowią ostateczną podstawę do zbudowania systemu.



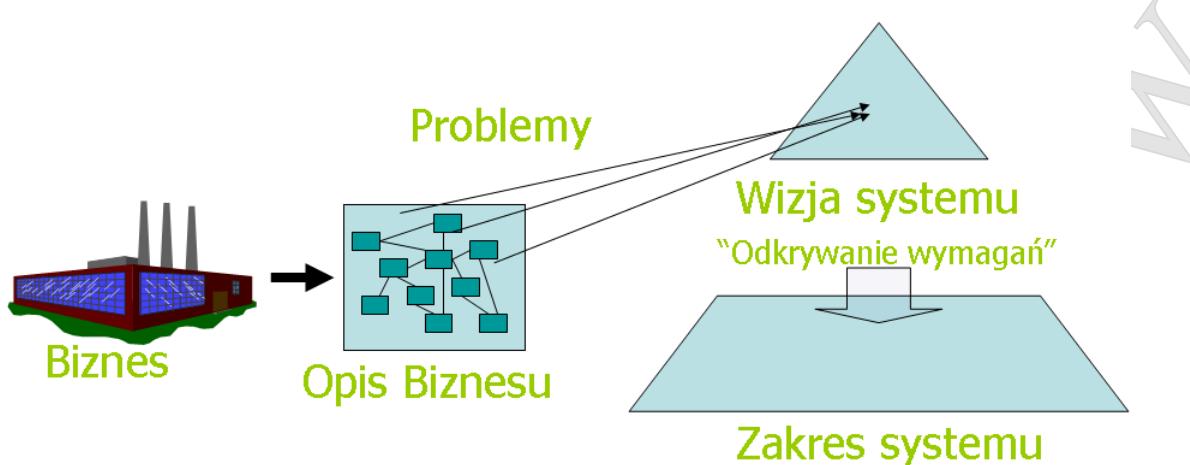
Rysunek 7-1. „Piramida” wymagań

Ze względu na typ, wymagania możemy podzielić na wymagania funkcjonalne, jakościowe, słownikowe i ograniczenia na system (pionowy podział na rysunku), Wymagania funkcjonalne określają „co” budowany system ma robić (bez opisywania „jak” system ma to robić). Wymagania słownikowe określają słownictwo używane we wszystkich rodzajach wymagań. Wymagania pozafunkcjonalne (ograniczenia na system i wymagania jakościowe) określają „jak” system ma realizować funkcjonalność założoną w wymaganiach funkcjonalnych, poprzez narzucenie ograniczeń oraz określenie miar jakościowych systemu (takich jak np. wydajność lub bezpieczeństwo).

7.2.1. Wymagania zamawiającego a wymagania oprogramowania

Wymagania zamawiającego różnią się od wymagań oprogramowania poziomem szczegółowości oraz przeznaczeniem. Pierwsze stanowią kontrakt pomiędzy zamawiającym a wykonawcą i definiują zakres systemu oraz określają realne potrzeby zamawiającego. Wymagania zamawiającego powinny być zdefiniowane w sposób czytelny dla zamawiającego (z wykorzystaniem słownictwa dziedzinowego) oraz być na tyle precyzyjne i jednoznaczne, aby na ich podstawie zdefiniować szczegółowe wymagania oprogramowania. Wymagania oprogramowania, jako podstawa do zbudowania systemu, powinny uszczegóławiać wymagania zamawiającego w sposób wystarczający do zdefiniowania na ich podstawie projektu architektonicznego, szczegółowego oraz implementacji systemu. O ile wymagania zamawiającego mogą definiować poszczególne jednostki funkcjonalności (przypadki użycia) przy pomocy akapitów tekstu w języku naturalnym lub naszkicowania głównych scenariuszy, o tyle wymagania oprogramowania powinny uzupełniać tę specyfikację o szczegółowy opis interakcji użytkownika z systemem w formie szczegółowych scenariuszy przypadków użycia, obejmujących scenariusze alternatywne i sytuacje wyjątkowe. Dodatkową pomocą w trakcie implementacji systemu może być zdefiniowanie scenopisów – tj. projektów ekranów powiązanych z poszczególnymi krokami scenariuszy. Z jednej strony pozwala to na wstępne zaprojektowanie interfejsu użytkownika, z drugiej umożliwia zobrazowanie przyszłym użytkownikom sposobu działania systemu już na etapie analizy wymagań, co ułatwia również wyłapanie nieścisłości i niepoprawnego zinterpretowania potrzeb zamawiającego.

Wymagania zamawiającego



Rysunek 7-2 Wymagania zamawiającego

Wymagania zamawiającego zajmują dwa najwyższe poziomy „piramidy” wymagań przedstawionej powyżej. Wymagania zamawiającego składają się z wizji systemu oraz wymagań użytkownika. Wizja systemu stanowi ogólną specyfikację cech systemu w ścisłym powiązaniu z potrzebami biznesowymi. Wymagania zamawiającego są odzwierciedleniem rzeczywistych potrzeb zdefiniowanych przez biznes. Do zdefiniowania konkretnych wymagań potrzebujemy „wizji”. Wizja jest odpowiedzią na problemy wynikające z opisu istniejącego biznesu. Na podstawie wizji systemu „odkrywamy” wymagania (Rysunek 7-2). Zakres systemu (wymagania użytkownika) jest wynikiem poszukiwania dróg automatyzacji biznesu zgodnie z wizją systemu.

Wizja systemu zawiera wymagania bezpośrednio wynikające z chęci usprawnienia zamawiającej organizacji. Są to wymagania zatwierdzane bezpośrednio przez sponsorów (np. kierownictwo organizacji), określające:

- „co ten system ogólnie ‘ma robić?’”,
- „jakie biznesowe korzyści przyniesie nam ten system?”
- „w jaki sposób system usprawni nasze działania?”

Wymagania użytkownika specyfikują wymagania na system w sposób wystarczający do określenia jego rozmiaru i nakładu pracy potrzebnego na jego zbudowanie. Na wymagania zamawiającego składają się ogólne wymagania definiujące zakres systemu, określające:

- „jak dużo ten system będzie robił?”,
- „ile ten system będzie nas kosztował?”,
- „kto będzie się tym systemem posługiwał?”

Na poziomie wymagań zamawiającego, wymagania często są formułowane za pomocą języka naturalnego (wizja) lub przypadków użycia wraz ze słownym opisem ich funkcjonalności i ewentualnie szkicem głównych scenariuszy

Wymagania oprogramowania

Wymagania oprogramowania stanowią uszczegółowienie wymagań zamawiającego. Na tym poziomie, wymagania muszą definiować w sposób jednoznaczny i spójny dialog pomiędzy użytkownikiem a systemem prowadzący do osiągnięcia celu biznesowego przypadku użycia. W tym celu opis słowny przypadków jest rozwijany o szczegółowe scenariusze przypadków użycia definiujące przepływ sterowania i obsługę sytuacji wyjątkowych.

Szczegółowe wymagania oprogramowania definiują specyficzne cechy systemu oprogramowania:

- „w jaki sposób system będzie ‘rozmawiał’ z użytkownikiem?”,
- „jakie dane system będzie przetwarzał?”

O określenie danych przetwarzanych przez system osiągamy poprzez uszczegółowienie słownika dziedziny na poziomie wymagań oprogramowania. Do słownika zdefiniowanego w wymaganiach zamawiającego powinniśmy dodać relacje pomiędzy użytymi pojęciami, krotności zdefiniowanych relacji oraz atrybuty poszczególnych pojęć. Słownik zdefiniowany na poziomie wymagań oprogramowania będzie również zawierał pojęcia „odkryte” w trakcie uszczegółowiania scenariuszy przypadków użycia, a nieobecne w słowniku zdefiniowanym dla wymagań zamawiającego. Dodatkowo powinniśmy w słowniku zdefiniować wszystkie operacje powiązane z danymi pojęciami (frazy czasownikowe ze scenariuszy przypadków użycia – patrz dalsze sekcje tego rozdziału).

7.2.2. Wymagania funkcjonalne a wymagania pozafunkcjonalne

Wymagania funkcjonalne określają, co budowany system ma robić, tym samym wyznaczając zakres systemu. Wymagania pozafunkcjonalne (jakościowe i ograniczenia wynikające ze środowiska) ograniczają przestrzeń możliwych realizacji wymagań funkcjonalnych. O ile wymaganie funkcjonalne mówi nam np., że system musi pozwalać na autoryzację użytkowników, o tyle wymagania pozafunkcjonalne mogą nam ograniczać możliwości implementacji poprzez określenie miar bezpieczeństwa (np. określenie standardów szyfrowania danych autoryzacyjnych przesyłanych przez sieć), wydajności (np. przy dużym obciążeniu systemu zwłoka w autoryzacji nie powinna być dłuższa niż 5 sekund) lub ograniczeń wynikających ze środowiska, w którym system będzie działał (np. autoryzacja ma korzystać z istniejącej bazy użytkowników przechowywanych w usłudze katalogowej LDAP). Wszelkie te ograniczenia i miary jakościowe wpłyną w dalszych etapach realizacji projektu na decyzje architektoniczne i projektowe, aby budowany system nie tylko realizował założone wymagania funkcjonalne, ale również realizował je w sposób zgodny z założeniami wymagań pozafunkcjonalnych.

Wymagania funkcjonalne

Wymagania funkcjonalne określają sposób zachowania się systemu oprogramowania w odpowiedzi na interakcje użytkownika. Określają, jakie usługi powinien oferować budowany system, w jaki sposób reagować na określone komunikaty wejściowe oraz jak zachowywać się w określonych sytuacjach. Zestaw wymagań funkcjonalnych określa zakres budowanego systemu. Zakres systemu możemy również określić stwierdzając, czego system nie powinien robić.

Wymagania pozafunkcjonalne

Wymagania pozafunkcjonalne określają cechy jakościowe oraz ograniczenia środowiskowe i techniczne, które musi spełniać system oprogramowania. Cechy te określają m.in., w jakim stopniu budowany system ma być niezawodny, sprawny, bezpieczny i przyjazny dla użytkownika oraz jakie normy powinien pełnić. Określają również pewne założenia dotyczące użytego sprzętu i oprogramowania (np. preferowany system bazy danych lub posiadane zasoby sprzętowe).

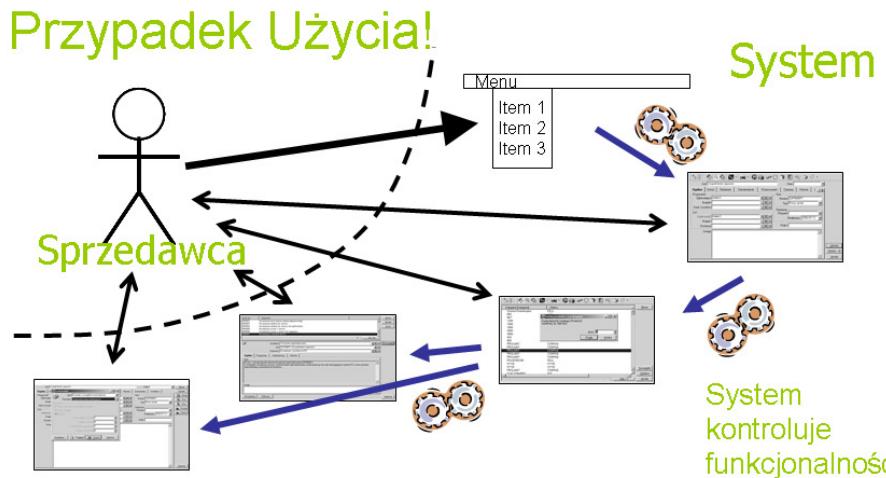
Wymagania jakościowe mogą być bezpośrednio związane z poszczególnymi wymaganiem funkcjonalnymi lub dotyczyć całości systemu (lub wybranej jego części).

7.3. Specyfikowanie widocznego sposobu zachowania się systemu

W celu opisania widocznego sposobu zachowania się systemu możemy posłużyć się modelem przypadków użycia wprowadzonym w rozdziale 6. Model przypadków użycia powinien być uzupełniony szczegółowym opisem każdego z przypadków użycia, a na poziomie wymagań oprogramowania również scenariuszami przypadków użycia.

Opisując przypadek użycia powinniśmy traktować system jako „czarną skrzynkę”. Opis przypadku użycia powinien koncentrować się na zachowaniu widocznym dla użytkownika. Cała „mechanika” wewnętrz systemu powinna zostać ukryta. System jest opisany jako „biała skrzynka” dopiero na etapie projektowania architektury. Przypadki użycia z aktorami ludzkimi opisują tylko interfejs użytkownika i efekty biznesowe zachowania systemu. Nie umieszczamy w nich żadnej informacji o danych przekazywanych między maszynami, danych przechowywanych w tabelach bazodano-

wych, rozpoczynaniu nowych wątków, czy tym podobnych aspektach technicznych realizacji systemu.



Rysunek 7-3 Modelowanie funkcjonalności systemu

Czym jest przypadek użycia? Przypadek użycia jest opisem zachowania systemu komunikującego się z jednym bądź więcej aktorami. Aby opis zachowania stanowił przypadek użycia, muszą być spełnione trzy warunki :

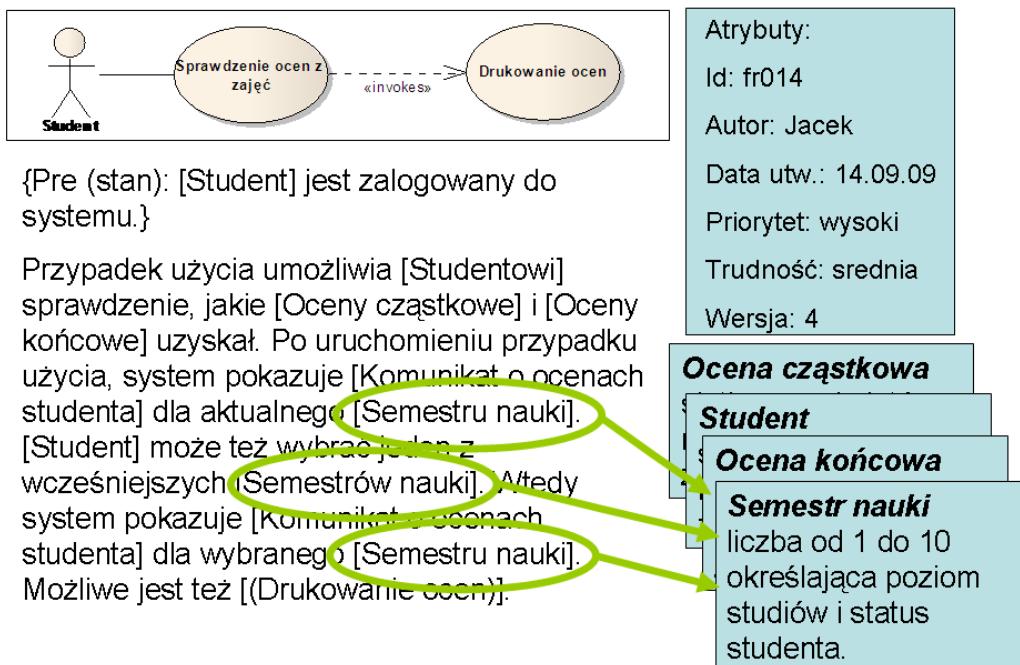
- Opis musi rozpoczynać się od interakcji aktora z systemem
- Opis musi przedstawiać wymianę komunikatów pomiędzy aktorem a systemem
- Opis musi jasno określić ostateczny cel osiągnięty na końcu wymiany komunikatów

Po osiągnięciu celu musi istnieć możliwość rozpoczęcia innego przypadku użycia (lub ponownego wykonania obecnego).

Przypadek użycia z reguły zawiera również alternatywne sekwencje wymiany komunikatów, które prowadzą do tego samego celu lub do porażki. W tym celu wykorzystujemy scenariusze alternatywne oraz rozgałęzienia scenariuszy.

Opis przypadków użycia systemu

Kompletny opis przypadku użycia powinien zawierać, oprócz opisu słownego charakteryzującego funkcjonalność reprezentowaną przez dany przypadek użycia, szereg dodatkowych informacji zapisanych w czytelny sposób. Dobry opis przypadku użycia powinien mieć wyszczególnionych aktorów biorących udział w interakcji (aktora głównego i ew. aktorów pobocznych, jeśli występują). Dobrze jest również wyszczegolić punkty rozszerzeń – pozwoli to na łatwe dotarcie do tej informacji bez zbędnego przedzierania się przez scenariusze przypadków użycia. Opis przypadku użycia może być uzupełniony ujednoliconymi na przestrzeni całej specyfikacji wymagań atrybutami takimi jak wersja wymagania, unikalny identyfikator, autor/autorzy wymagania, data stworzenia/modyfikacji, priorytet i trudność realizacji wymagania. Elementy takie jak unikalny identyfikator, autorzy i daty modyfikacji/utworzenia wymagania ułatwiają proces zarządzania zmianami oraz śledzenia tych zmian w projekcie architektonicznym i szczegółowym. Priorytet i trudność realizacji wymagania są atrybutami ułatwiającymi zarządzanie projektem prowadzonym w cyklu iteracyjnym. Priorytety pozwalają na przyporządkowanie wymagań do poszczególnych iteracji, oszacowanie poziomu trudności realizacji natomiast ułatwia oszacowanie pracochłonności poszczególnych faz projektu.

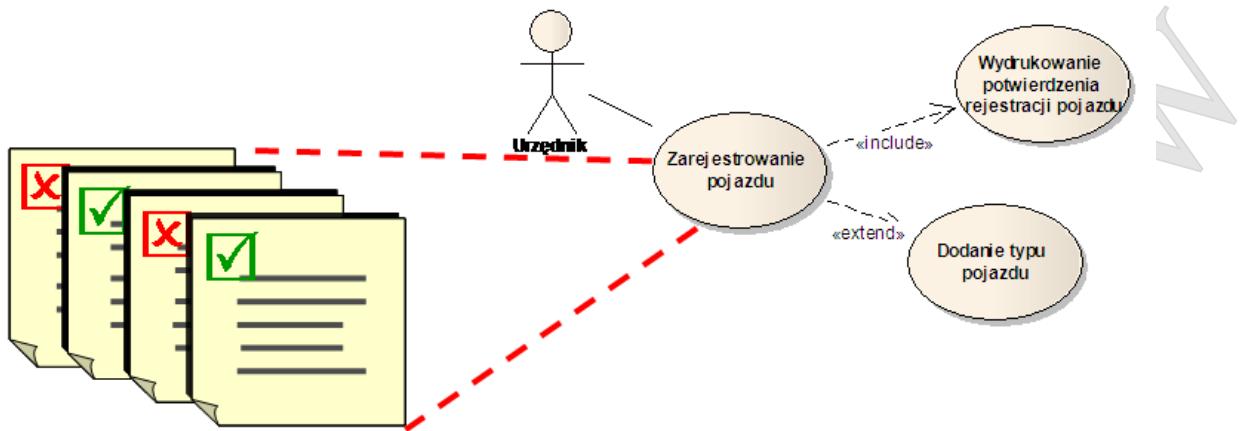


Rysunek 7-4 Przykład opisu przypadku użycia

Kolejnym istotnym aspektem opisu przypadku użycia jest precyzja sformułowań stosowanych w słownym opisie przypadków. Aby uniknąć niejednoznaczności, wszystkie istotne pojęcia występujące w opisie powinny znaleźć się wraz z ich definicjami w słowniku dziedzinowym (sam opis wymagań powinien być pozbawiony wszelkich definicji pojęć). Więcej informacji na temat sposobu definiowania słownika oraz jego znaczenia dla jakości specyfikacji wymagań znajduje się w rozdziałach 7.4 oraz 7.6. Opis na poziomie szczegółowości przedstawionym powyżej jest wystarczający na poziomie specyfikacji wymagań zamawiającego. Na poziomie wymagań oprogramowania, opis powinien być rozszerzony o szczegółowe scenariusze oraz ew. scenopisy z projektami interfejsu użytkownika. Rysunek 7-4 przedstawia przykład kompletnego opisu przypadku użycia na poziomie specyfikacji wymagań zamawiającego. Pokazane są również niektóre odnośniki do pojęć w słowniku.

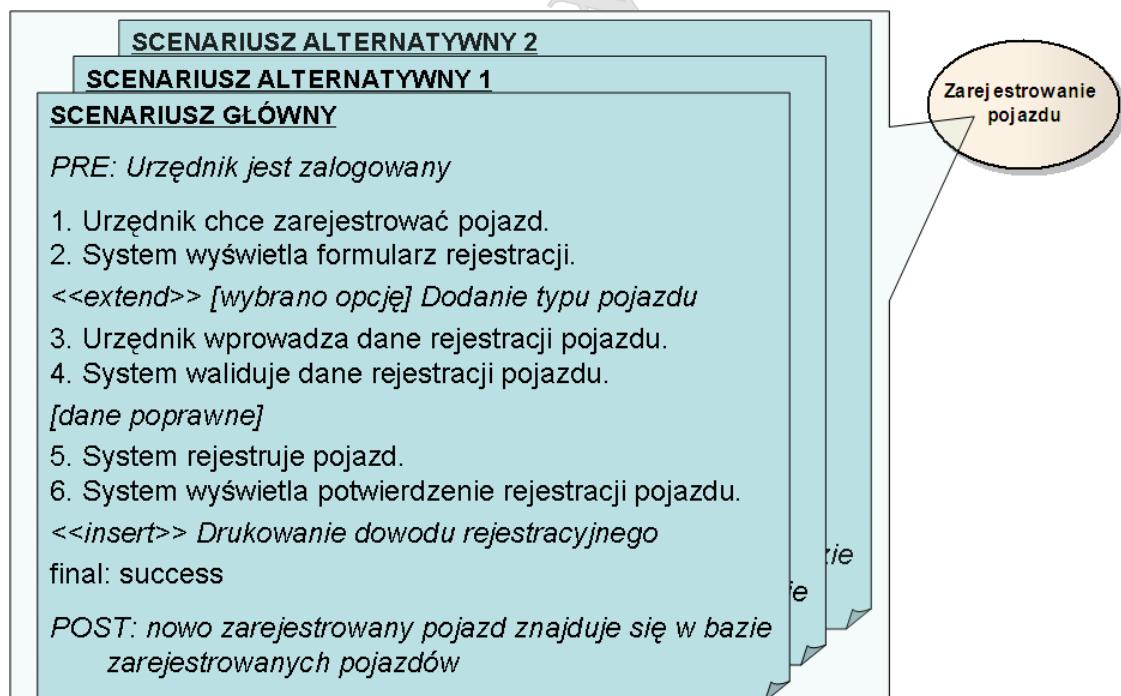
Scenariusze przypadków użycia

Zgodnie z wcześniejszą definicją przypadku użycia, przypadek powinien zaczynać się od interakcji aktora, zawierać sekwencję interakcji pomiędzy aktorem i systemem oraz kończyć się osiągnięciem celu przypadku użycia bądź porażką. W celu opisania takiej interakcji użytkownika z systemem posłużymy się scenariuszami przypadków użycia. Scenariusze mogą reprezentować różne alternatywne ścieżki prowadzące do osiągnięcia jednego celu bądź prowadzące do porażki. Przypadek użycia grupuje wszystkie scenariusze powiązane z danym celem (patrz Rysunek 7-5).



Rysunek 7-5 Przypadek użycia i scenariusze

Podczas tworzenia wymagań oprogramowania musimy wyznaczyć najbardziej prawdopodobne scenariusze (być może z pominięciem kilku oczywistych, które mogą być opisane ogólnie na początku specyfikacji, np. typowe scenariusze anulowania). Rysunek 7-5 przedstawia grupę przypadków użycia powiązanych z przypadkiem „Zarejestrowanie pojazdu” oraz wyróżnia scenariusze tego przypadku użycia. Rysunek 7-6 przedstawia przykładowy scenariusz główny tego przypadku użycia. Scenariusz taki składa się z warunku początkowego i końcowego oraz uporządkowanej sekwencji wymiany komunikatów pomiędzy aktorem a systemem. Interakcja rozpoczyna się z inicjatywy aktora („Użytkownik chce zarejestrować pojazd”). Kolejne zdania stanowią „dialog” użytkownika systemem. Aby scenariusz był jednoznaczny i łatwy do zrozumienia, do opisu interakcji wykorzystujemy zdania w prostej gramatyce: podmiot – orzeczenie – dopełnienie bliższe (dopełnienie dalsze), gdzie podmiotem zawsze jest system, aktor główny lub aktorzy pomocni przypadku użycia, a fraza czasownikowa opisuje akcję wykonywaną przez system lub użytkownika (zależnie od podmiotu zdania).



Rysunek 7-6 Przypadek użycia wraz z scenariuszami

W celu uniknięcia niejednoznaczności w specyfikacji wymagań, wszystkie obiekty (dopełnienia) występujące w zdaniach scenariuszy oraz opisie słownym przypadku użycia powinny być zdefiniowane w słowniku dziedziny. Pozwoli to uniknąć stosowania homonimów i synonimów w specyfikacji (więcej w sekcji 7.6).

Oprócz zwykłych zdań opisujących interakcję użytkownika z systemem, w scenariuszach możemy używać dodatkowych zdań sterujących opisujących przepływ sterowania pomiędzy poszczególnymi scenariuszami lub pomiędzy przypadkami użycia.

Jak było wspomniane wcześniej, grupa scenariuszy powiązanych z danym przypadkiem użycia opisuje różne alternatywne ścieżki wykonania danego przypadku użycia. W tym celu wprowadza się rozgałęzienia scenariuszy i zdania warunkowe. W przykładowym scenariuszu głównym, po zdaniu „System waliduje dane rejestracji pojazdu”, następuje warunek „dane poprawne” i scenariusz prowadzi dalej do osiągnięcia celu przypadku użycia. W scenariuszu alternatywnym scenariusz będzie się rozgałęziać i w dalszej części będą następowały komunikaty obsługujące sytuację, w której dane wprowadzone przez aktora są niepoprawne. Rozgałęzienia mogą następować w wyniku stanu systemu (jak w przykładzie powyżej) bądź w wyniku wyboru aktora.

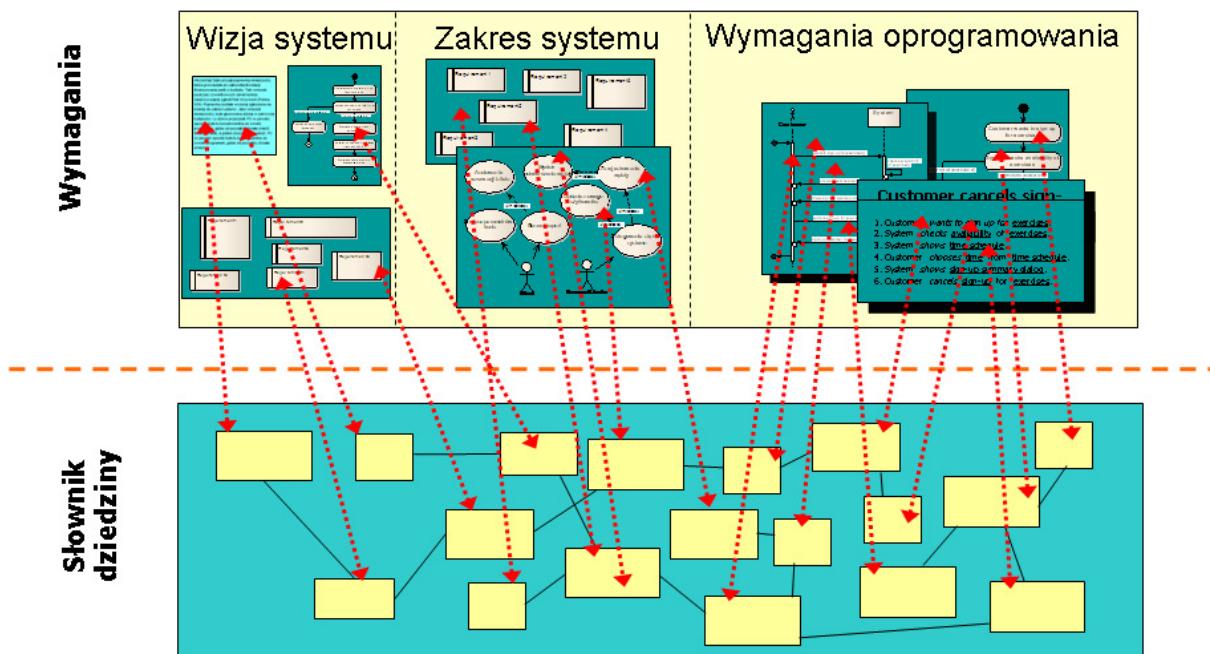
Kolejnym przykładem przekazania sterowania mogą być zdania typu <<extend>> i <<insert>>. Zdania te oznaczają odpowiednio warunkowe i bezwarunkowe przekazanie sterowania do innego przypadku użycia. W przykładowym scenariuszu, po zdaniu 2, jeśli użytkownik wybierze taką opcję, zostanie wywołany przypadek użycia „Dodanie typu pojazdu”, natomiast po zdaniu 6, niezależnie od decyzji aktora, zostanie wywołany przypadek „Drukowanie dowodu rejestracyjnego”. W obu sytuacjach, po wykonaniu wywołanego przypadku użycia sterowania powraca do przypadku „Zarejestrowanie pojazdu” w miejscu w którym inny przypadek użycia został wywołany.

Ostatnim zdaniem sterującym w przedstawionym przykładzie jest zdanie „final: success” informujące o tym, że dany scenariusz kończy się powodzeniem (osiągnięciem celu przypadku użycia). W alternatywnych scenariuszach interakcja może się zakończyć niepowodzeniem. Dodatkowo, scenariusze mogą się zakończyć przekazaniem sterowania do pewnego kroku innego scenariusza. W takiej sytuacji, od miejsca powrotu sterowania do scenariusza interakcja w obu scenariuszach przebiega identycznie.

7.4. Specyfikowanie słownika danych przetwarzanych przez system

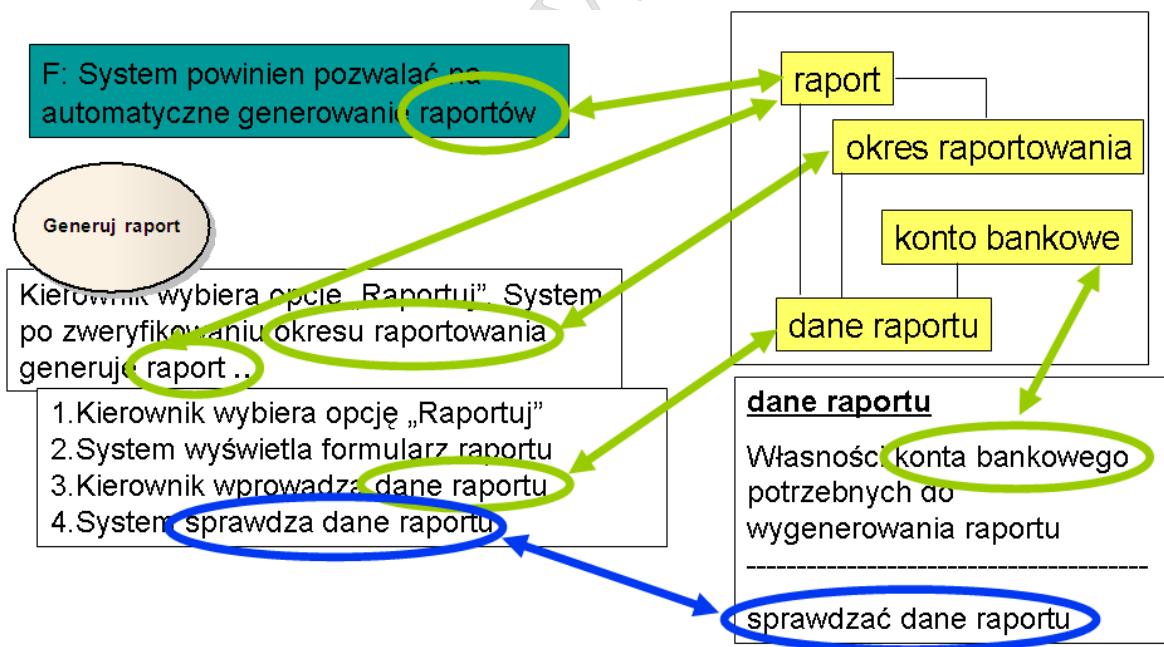
Wymagania słownikowe definiują zakres pojęć i danych, jakimi system oprogramowania ma操作. Słownik zawiera definicje pojęć używanych w ramach wymagań funkcjonalnych i pozafunkcjonalnych. Oprócz samej definicji pojęć opisujemy też zależności między nimi. Zależności te mogą być wyrażone w formie graficznej. Słownik dziedziny powinien być tworzony, aktualniany i używany przez cały etap definiowania zakresu systemu i wymagań oprogramowania.

Zadaniem słownika dziedziny jest odseparowanie opisu akcji (dynamiki systemu) od definicji pojęć. Rysunek 7-7 przedstawia schematycznie kompletną specyfikację wymagań z wyróżnionym podziałem na część opisującą wymagania i słownik dziedziny. W górnej części rysunku znajduje się specyfikacja wymagań na różnych poziomach szczegółowości (Wizja systemu, wymagania użytkownika oraz wymagania oprogramowania). Każdy z tych komponentów specyfikacji wymagań odnosi się do wspólnego słownika dziedziny. Wymagania jako takie powinny charakteryzować pożąданie cechy budowanego systemu, lecz unikać definiowania pojęć wewnętrz opisu. Wszelkie definicje pojęć występujących w każdym typie wymagań na każdym poziomie szczegółowości powinny znaleźć się we wspólnym słowniku dziedziny, który rozrasta się i ewoluje wraz z uszczegółowianiem specyfikacji.



Rysunek 7-7 Słownik dziedziny

Definiowanie słownika obejmuje zdefiniowanie wszystkich pojęć użytych w opisach wymagań i scenariuszach przypadków użycia, uzupełnienie tych pojęć o definicje fraz użytych w zdaniach scenariuszy oraz uzupełnienie modelu o relacje pomiędzy pojęciami. Rysunek 7-8 przedstawia proces budowania słownika podczas specyfikacji wymagań. Dobrą praktyką stosowaną w celu wyszukiwania pojęć jest podkreślanie wszystkich rzeczowników występujących w opisach wymagań. Przy zwięzle sformułowanych opisach wymagań, większość - jeśli nie wszystkie rzeczowniki pojawiające się w tych opisach potencjalnie stanowią pojęcia dziedzinowe których definicja powinna znaleźć się w słowniku.



Rysunek 7-8 Jak szukać pojęć do słownika dziedziny?

Jak przedstawiono na rysunku (patrz Rysunek 7-1), źródłem pojęć mogą być zarówno wymagania wysokiego poziomu, zdefiniowane w wizji systemu (cechy systemu), opisy słowne przypadków użycia zdefiniowane na etapie specyfikacji wymagań użytkownika, szczegółowe scenariusze zdefiniowane w specyfikacji wymagań oprogramowania jak i definicje innych pojęć znajdujących się w słowniku.

Tworząc słownik dziedziny, powinniśmy kierować się kilkoma prostymi zasadami, które spowodują, że stworzony słownik będzie jednoznaczny i poprawny. Każde pojęcie umieszczone w słowniku powinno zawierać definicję. Nazwy pojęć umieszczanych w słowniku powinny być uzgodnione między wszystkimi stronami zaangażowanymi w formułowanie wymagań. Definicje pojęć umieszczone w słowniku powinny być na tyle rozbudowane, aby nie pozostawały wątpliwości co do znaczenia danego pojęcia. Dobrą praktyką (jeśli słownik jest definiowany np. w procesorze tekstu) jest ustalenie spójnej notacji dla słownika (użycie czcionek, format wpisu w słowniku, itp.). Słownik powinien być aktualizowany przez cały czas trwania projektu.

Sporym wyzwaniem jest zapewnienie spójności i aktualności słownika. Najczęściej zapewniane jest to poprzez ręczną aktualizację słownika w formie dokumentu tekstowego. Wymaga to sporego nakładu pracy i jest powodem wielu błędów. Alternatywą dla słownika przechowywanego w dokumencie tekstowym jest zastosowanie serwisu typu Wiki. Ułatwia to jednoczesną pracę nad słownikiem wielu osobom (scentralizowany słownik) oraz pozwala na zachowanie spójności i łatwą nawigację po słowniku poprzez hiperłącza. Wadą takiego rozwiązania jest brak integracji z modelem wymagań oraz brak wsparcia dla wizualizacji słownika. Najlepszą (choć wymagającą zazwyczaj zakupu odpowiedniego narzędzia) metodą przechowywania słownika jest skorzystanie ze zintegrowanego systemu zarządzania wymaganiami posiadającego wbudowany edytor słownika dziedziny. Rozwiązanie takie daje pełną integrację słownika z modelem wymagań, wspiera wizualizację słownika (diagramy) oraz często pozwala na wykorzystanie hiperłączy zarówno w samym słowniku jak i w opisach wymagań.

7.5. Cechy dobrej specyfikacji wymagań.



Rysunek 7-9 Cechy dobrej specyfikacji wymagań

Kluczem do osiągnięcia sukcesu przez projekt jest zapewnienie dobrej jakości specyfikacji wymagań. Rysunek 7-9 przedstawia pożądane cechy dobrej specyfikacji wymagań. Są nimi: Kompletność, Jednoznaczność, Poprawność, Spójność, Możliwość śledzenia, Możliwość zarządzania, Możliwość testowania oraz Podatność na zmiany. Cechy te omówimy w kolejnych sekcjach poniżej.

Kompletność

Dobra specyfikacja wymagań to taka, która jest kompletna. Co to oznacza? Oznacza to przede wszystkim, że specyfikacja odzwierciedla wszystkie realne potrzeby użytkownika. Kompletna specyfikacja wymagań w wyczerpujący sposób definiuje wymagania na system zgodne z oczekiwaniemi użytkownika. Kompletność specyfikacji oznacza również, że specyfikacja obejmuje wszystkie typy wymagań (zgodnie z klasyfikacją podaną w sekcji 7.2). Aby specyfikacja była kompletną upewnij się, że zawiera wszystkie potrzebne wymagania:

- Nieustannie zadawaj sobie (i klientowi) pytanie : czy jest coś jeszcze? Czy niczego nie pominieliśmy?
- Korzystaj z odpowiednich technik odkrywania „nie odkrytych” wymagań.

- Pamiętaj o pokryciu wymaganiami wszystkich czterech obszarów (funkcjonalne, pozafunkcjonalne, słownikowe, ograniczenia).

Specyfikacja wymagań systemu jest kompletna kiedy zawiera:

- Wszystkie istotne scenariusze dla wszystkich przypadków użycia.
- Wszystkie wymagania pozafunkcjonalne obejmujące wszystkie obszary.
- Wszystkie atrybuty i operacje dla wszystkich pojęć użytych w scenariuszach i wymaganiach pozafunkcjonalnych .
- Wszystkie ograniczenia obejmujące wszystkie obszary.

Jednoznaczność

Jednoznaczność specyfikacji oznacza, że jest ona zrozumiała przez wszystkie grupy odbiorców, oraz że jest przez nich interpretowana w ten sam sposób. Jednoznaczna specyfikacja wymagań nie może zawierać sformułowań i definicji wymagań które mogą być interpretowany w różny sposób np. zależnie od kontekstu lub poziomu wiedzy technicznej/dziedzinowej osoby która ją czyta. Aby zapewnić jednoznaczność specyfikacji, upewnij się że wszystkie wymagania mogą być zinterpretowane tylko w jeden sposób:

- Czy wymagania są przejrzyste dla użytkownika? Dla developera?
- Rozmawiaj z klientem, twórz modele, pokazuj je klientowi i znów rozmawiaj z klientem...
- Najlepszym sposobem na rzeczywiste zweryfikowanie jednoznaczności jest zbudowanie systemu – dla tego celu istnieje proces iteracyjny

Wymagania są przejrzyste i jednoznaczne jeśli:

- Nazwy przypadków użycia określają przejrzysty i istotny cel
- Zdania scenariuszy formułowane są w prostej gramatyce (POD(D))
- Scenopisy precyzyjnie definiują interfejs użytkownika
- Wszystkie dopełnienia (klasy i atrybuty) oraz orzeczenia (operacje) użyte w zdaniach posiadają precyzyjne definicje
- Przypadki użycia i diagramy klas mają nałożone odpowiednie ograniczenia

Poprawność

Poprawna specyfikacja wymagań to taka, która odzwierciedla realne potrzeby zamawiającego. Oznacza to, że potrzeby zamawiającego/przyszłych użytkowników zostały prawidłowo zrozumiane i poprawnie zinterpretowany przez osobę specyfikującą wymagania. W większości przypadków, specyfikujący wymagania nie posiada wiedzy dziedzinowej lub jego wiedza z dziedziny budowanego systemu jest znacznie ograniczona w porównaniu z wiedzą zamawiającego. Aby zapewnić poprawność specyfikacji, upewnij się ze własności systemu zdefiniowane z wymaganiach odpowiadają realnym potrzebom klienta:

- Nieustannie zadawaj sobie pytanie i konsultuj to z zamawiającym: czy system na pewno ma się tak zachowywać? Czy na pewno powinien przetwarzać takie dane (pojęcia)?
- W rzeczywistości własności systemu można uznać za poprawne dopiero wtedy, kiedy zamawiający zapłaci za system... Następuje to dopiero wtedy, kiedy system jest już gotowy i klient przyznaże, że tego właśnie chciał.

Kolejną dobrą praktyką ułatwiającą sformułowanie poprawnej specyfikacji wymagań jest pisanie scenopisów (uzupełnienie scenariuszy przypadków użycia projektami ekranów z nimi powiązanych). Scenopisy wyjaśniają zamawiającemu jak system powinien się zachowywać i jakimi danymi będzie się posługiwał

- Porównuj wymagania z biznesem zamawiającego

- Uwzględniaj szczegółowo uwagi zamawiającego na scenopisach

Spójność

Specyfikacja wymagań definiuje wymagania na różnych poziomach szczegółowości. Od ogólnie zdefiniowanych pożądanych cech systemu w wizji systemu, poprzez opisane językiem naturalnym jednostki funkcjonalności w wymaganiach użytkownika po szczegółową specyfikację poszczególnych wymagań w wymaganiach oprogramowania. Aby zapewnić spójność specyfikacji, musimy się upewnić, że wymagania zdefiniowane na różnych poziomach szczegółowości są ze sobą spójne oraz że specyfikacja nie zawiera wymagań sprzecznych. Aby stworzyć spójne wymagania:

- Upewnij się, że każde pojęcie użyte w scenariuszach bądź wymaganiach pozafunkcjonalnych jest zdefiniowane w słowniku.
- Upewnij się, że w słowniku nie ma synonimów (pojęć o takiej samej definicji ale różnych nazwach) lub homonimów (pojęć o takich samych nazwach ale różnych definicjach).
- Upewnij się, że orzeczenia w scenariuszach są połączone z ich definicjami w słowniku
- Upewnij się, że scenariusze mają poprawne rozgałęzienia (używaj diagramów aktywności)

Możliwość śledzenia

Możliwość śledzenia jest cechą specyfikacji umożliwiającą dotarcie od danego wymagania do wymagań na wyższym poziomie ogólności z których wynika, bądź wymagań bardziej szczegółowych które zapewniają zrealizowanie danego wymagania w systemie. Aby to osiągnąć:

- Upewnij się ze nadajesz identyfikatory wszystkim wymaganiom (przypadki użycia, scenariusze, pojęcia).
- Upewnij się, że każdy identyfikator wymagań użytkownika posiada identyfikator źródła wymagania.

Śledzenie na podstawie wymagań oznacza:

- Każdy scenariusz (diagram aktywności) jest powiązany z odpowiednim przypadkiem użycia
- Każda klasa jest powiązana z odpowiednim pojęciem lub cechą
- W uogólnieniu – każde wymaganie posiada swoje źródło i może być przetransformowane do projektu szczegółowego

UWAGA: ślady nie są relacją 1 do 1 – kilka wymagań może wskazywać na jedno bądź wiele wymagań

Możliwość zarządzania

Możliwość zarządzania wymaganiami oznaczy, iż każde wymagania można przejrzyście umiejszczyć w cyklu życia oprogramowani. Kluczem do zarządzania wymaganiami jest stosowanie atrybutów, zapewniające możliwość manipulowania nimi w projekcie wytwarzania oprogramowania. Aby zapewnić możliwość zarządzania wymaganiami, każde z nich powinno posiadać odpowiednie atrybuty takie jak :

- Priorytet użytkownika (jak ważne jest dla użytkownika)
- Trudność techniczna (jak ważne jest z punktu widzenia architektury)
- Przypisanie (osoba odpowiedzialna za wymaganie)
- Wersja (liczba określająca sekwencje zmian)
- Wersja ostateczna (iteracja, w której powinno być zrealizowane)
- Stabilność (prawdopodobieństwo zmiany)

Możliwość testowania

Możliwość testowania jest cechą specyfikacji umożliwiającą nam zweryfikowanie każdego wymagania niezależnie od jego typu. Co innego oznacza to w wypadku wymagań funkcjonalnych, a co innego w wypadku pozafunkcjonalnych. Aby możliwe było przetestowanie wymagań sformułowanych w specyfikacji, upewnij się czy każde wymaganie posiada miarę jakości pozwalającą na weryfikację końcowego produktu (patrz również “poprawność”).

- Nieustannie pytaj siebie (i klienta): jak zweryfikujemy, że to wymaganie jest spełnione przez produkt końcowy?
- Nie zapominaj, że implementacja wymaga „dowodu” (najczęściej testu), że system jest poprawny.

Możliwość testowania dla różnych typów wymagań oznacza że:

- Wymagania funkcjonalne (przypadki użycia) posiadają procedury (przypadki testowe) pozwalające testerom sprawdzenie poprawności działania systemu.
- Wymagania słownikowe (klasy) posiadają załączone dane testowe weryfikujące poprawność wprowadzania, przetwarzania i wyników wyjściowych systemu.
- Wymagania pozafunkcjonalne posiadają miary (precyzyjne liczby) które mogą być sprawdzone w sposób obiektywny przez odpowiednie procedury testowe.

Podatność na zmiany

Podatność na zmiany charakteryzuje na ile łatwo jest wprowadzić zmiany do specyfikacji wymagań. Zapewnienie podatności na zmiany specyfikacji zmniejsza ryzyko związane z późnym wykryciem błędów/nieścisłości w specyfikacji i zabezpiecza przed sytuacją w której klient nagle żąda zmian w zaawansowanej już specyfikacji wymagań. Pisząc specyfikację wymagań, pamiętaj, że zmiany wymagań zdarzają się w (prawie) każdym projekcie! Aby ułatwić wprowadzanie zmian upewnij się że Twoja specyfikacja umożliwia precyzyjne wskazanie miejsca w którym nastąpiła zmiana.

Wymagania umożliwiające wprowadzanie zmian podzielone są na „fragmenty”:

- Przypadki użycia, które mają przypisane precyzyjne scenariusze, zdania w scenariuszach i elementy zdań.
- Klasy, które mają listy atrybutów i operacji.

7.6. Zachowanie spójności i jednoznaczności specyfikacji wymagań.

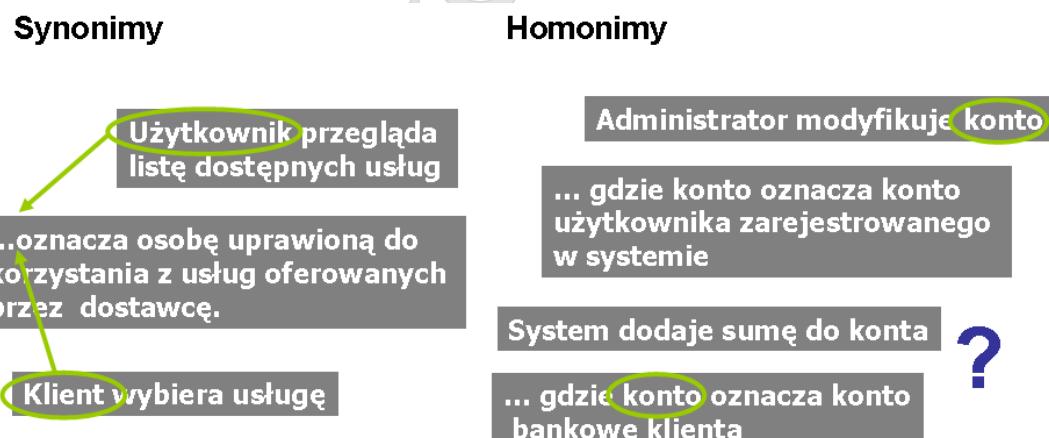
W miarę uzupełniania specyfikacji o nowo pozyskane wymagania, specyfikacja staje się coraz bardziej skomplikowana i zwiększa swoją objętość. Powoduje to, iż wraz z rozrastaniem się specyfikacji, coraz trudniej zapewnić jej spójność i jednoznaczność. Sprawa komplikuje się w wypadku dużych projektów, kiedy specyfikacja tworzona jest przez więcej jak jedną osobę. W celu zapewnienia nad złożonością specyfikacji stosuje się techniki takie jak prawidłowa dekompozycja wymagań oraz wykorzystanie słownika dziedziny w celu uspójnienia słownictwa stosowanego w specyfikacji.

Jednym ze sposobów radzenia sobie ze spójnością specyfikacji wymagań jest stosowanie prostej, przejrzystej gramatyki zdań scenariuszy przypadków użycia. W większości przypadków zdania w gramatyce „Podmiot-orzeczenie-dopełnienie bliższe-dopełnienie dalsze” są zupełnie wystarczające do opisania dialogu pomiędzy użytkownikiem a systemem. Konsekwentne stosowanie tej zasady wyklucza pokusę umieszczenia definicji wewnętrz specyfikacji interakcji użytkownika z systemem. Gramatyka ta wymusza umieszczenie definicji poszczególnych pojęć we wspólnym słowniku. W ten sposób otrzymujemy jasne rozdzielenie opisu statycznego systemu (słownik z definicjami pojęć) od opisu dynamicznego (opis interakcji użytkownik-system).



Rysunek 7-10 Znaczenie słownika w zachowaniu spójności wymagań

Rysunek 7-10 pokazuje to oddzielenie na przykładzie. W górnej części rysunku znajduje się Scenariusz napisany zgodnie ze wspomnianą wcześniej gramatyką. Jak widać wymaganie napisane w ten sposób stanowi jedynie opis dynamiki systemu, a pozbawione jest wszelkich definicji. W pierwszym zdaniu przykładowego scenariusza występuje fraza „dodać nowy wykład do kursu”. Zarówno pojęcie kurs jak i wykład może występować w wielu miejscach w całej specyfikacji. Jeśli definicje tych pojęć znajdowałyby się w miejscu ich wystąpienia w wymaganiach, niemożliwe byłoby łatwe dotarcie do nich. Mogło by to również doprowadzić do sytuacji, w której definicja jednego pojęcia znajduje się w kilku różnych miejscach specyfikacji. W wypadku zmiany lub rozszerzenia definicji pojęcia, jest prawie niemożliwe uaktualnienie jej we wszystkich miejscach wystąpienia. Każde wystąpienie pojęcia w specyfikacji powinno wskazywać na jedno miejsce gdzie to pojęcie jest zdefiniowane (tym miejscem jest oczywiście słownik dziedziny). Aby specyfikacja była spójna, tę zasadę powinniśmy rozciągnąć na opisy wszystkich wymagań w każdej formie. Zarówno opisy słowne przypadków użycia jak i definicje wymagań pozafunkcjonalnych powinny być pozbawione definicji pojęć. Wszelkie pojęcia użyte w tych opisach powinny być zdefiniowane jedynie w słowniku.



Rysunek 7-11 Synonimy i homonimy w specyfikacji wymagań

Zastosowanie słownika pojęć wykorzystywanych w specyfikacji znacznie ułatwia zapewnienie jednoznaczności specyfikacji wymagań, jednakże sam fakt stosowania słownika nie zapewni nam tej jednoznaczności. Konieczne jest jeszcze poprawne stosowanie tego słownika. Podstawową zasadą podczas pisania wymagań powinno być unikanie homonimów i synonimów zarówno w opisach słownych wymagań, definicjach pojęć w słowniku jak i w zdaniach scenariuszy przypadków użycia. Rysunek 7-11 przedstawia niejednoznaczności jakie wprowadza używanie homonimów i synonimów w definicji wymagań. W lewej części rysunku przedstawiono dwa wymagania, jedno posługujące się pojęciem „użytkownik” a drugie pojęciem klient. Jak widać na rysunku, w kontekście danej specyfikacji wymagań oba pojęcia oznaczają to samo i wskazują na tę samą

definicję. Stosowanie synonimów w odniesieniu do tych samych pojęć może doprowadzić do sytuacji, w której np. projektant lub architekt potraktuje „użytkownika” i „klienta” jako dwa oddzielne pojęcia, co doprowadzi do błędnego projektu systemu. Analogicznie sytuacja w której stosuje się tę samą nazwę dla odrębnych pojęć, może doprowadzić do błędnej interpretacji wymagań przez osoby realizujące system. Prawa część tego samego rysunku przedstawia dwa wymagania z jednej specyfikacji posługujące się pojęciem „konto”. Jednakże w obu tych kontekstach „konto” oznacza coś zupełnie innego (w jednym wypadku jest to konto użytkownika systemu, w drugim konto bankowe). W projekcie systemu opartego na takiej niejednoznacznej specyfikacji może dojść do sytuacji w której powstanie sztuczny twór łączący w sobie cechy obu tych pojęć, zupełnie niezgodny z oryginalnym zamysłem osoby specyfikującej wymagania. Im później taki błąd zostanie odkryty, tym większe będą koszty jego naprawienia. W prawidłowo napisanej specyfikacji wymagań, każde z tych pojęć powinno mieć własną unikalną nazwę (np. „konto bankowe” i „konto użytkownika”). Przy zastosowaniu unikalnych nazw, niezależnie od kontekstu, każda osoba czytająca specyfikację zinterpretuje ją poprawnie.

W powyższym podrozdziale skupiono się jedynie na wybranych najważniejszych technikach zapewnienia spójności i jednoznaczności specyfikacji wymagań. Należy pamiętać również o stowarzyszeniu wizualizacji, która znacznie ułatwia wychwycenie wszelkich błędów i niespójności. Zarówno wizualizacja słownika i relacji pomiędzy pojęciami w postaci np. diagramu klas, jak i wizualizacja przepływu sterownia wewnętrz przypadku użycia, poprzez reprezentacje jego scenariuszy w formie diagramu aktywności mogą wyeliminować wiele potencjalnych źródeł błędów trudnych do wyśledzenia w notacji tekstowej. Kolejnym nieocenionym sprzymierzeńcem w walce z chaosem w specyfikacji wymagań są relacje i zależności pomiędzy wymaganiami. Pozwalają one na wyśledzenie tych obszarów specyfikacji, które np. wymagają aktualizacji po zmianie określonych wymagań.

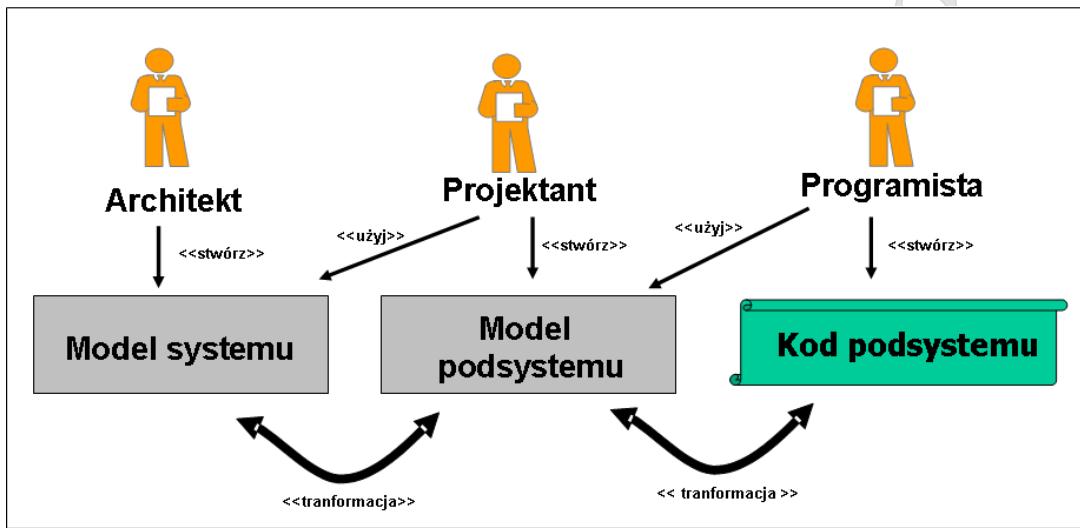
7.7. Podsumowanie

Inżynieria wymagań obejmuje swoim zakresem aktywności na najwcześniejszym i najistotniejszym etapie budowy systemu. Błędy popełnione na etapie specyfikacji wymagań są najdroższe do naprawienia. Od jakości specyfikacji wymagań zależy też czy klient otrzyma produkt jakiego oczekiwali. Istotą specyfikowania wymagań jest nie tylko odzwierciedlenie realnych potrzeb użytkownika ale również zrobienie tego w takiej formie, która będzie zrozumiała dla wszystkich grup odbiorców wymagań oraz umożliwiało zweryfikowanie, czy budowany system spełnia te wymagania.

8. Podstawy projektowania

8.1. Projektowanie na różnych poziomach złożoności.

Współczesne systemy informatyczne charakteryzują się coraz większym poziomem złożoności. Projekty składają się z setek klas a ostateczny produkt zawiera setki tysięcy linii kodu. Żaden członek zespołu realizującego projekt nie jest w stanie ogarnąć go jako całości. Z powodu tej złożoności, twórcy systemów zmuszeni są stosować dekompozycję problemu na mniejsze zamknięte jednostki.



Rysunek 8-1 Projektowanie na różnych poziomach złożoności

Schemat projektowania systemu na różnych poziomach szczegółowości przedstawia Rysunek 8-1. Architekt tworzy ogólny model systemu, dzieląc jego funkcjonalność na poszczególne komponenty oraz definiując interfejsy komponentów i przepływ komunikatów pomiędzy nimi. Z punktu widzenia architekta, komponenty są czarnymi skrzynkami komunikującymi się z pozostałą częścią systemu poprzez dobrze zdefiniowane interfejsy dostarczane i wymagane, oraz zgodnie z pewnym schematem wymiany komunikatów określonym np. za pomocą diagramów sekwencji dla modelu architektonicznego. Diagramy takie przedstawiają jedynie publiczne interfejsy komponentów oraz same komponenty, ale nie reprezentują wymiany komunikatów wewnętrz komponentów. Projektowanie systemu na takim poziomie abstrakcji powoduje, że architekt musi zapanować nad kilkoma/kilkunastoma komponentami zamiast setkami klas na raz. Architekt dodatkowo może zaprojektować część modeli projektowych komponentów, jako wskazówkę dla projektanta uszczegóławiającego te komponenty.

Projektant korzystając ze specyfikacji komponentów stworzonej przez architekta, projektuje „wnętrze” 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 jedynie na kilku/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 szczegółowego komponentów współpracujących z aktualnie projektowanym. Programista na podstawie projektu szczegółowego implementuje poszczególne klasy wewnątrz komponentów.

8.1.1. Projektowanie architektoniczne.

Celem projektowania architektonicznego jest opracowanie ogólnego projektu systemu na poziomie komponentów. Architektura systemu reprezentuje strukturę oraz dynamikę budowanego systemu na wyższym poziomie ogólności. Decyzje podjęte na etapie projektowania architektonicznego mają również kluczowe znaczenie dla spełnienia wymagań pozafunkcjonalnych.

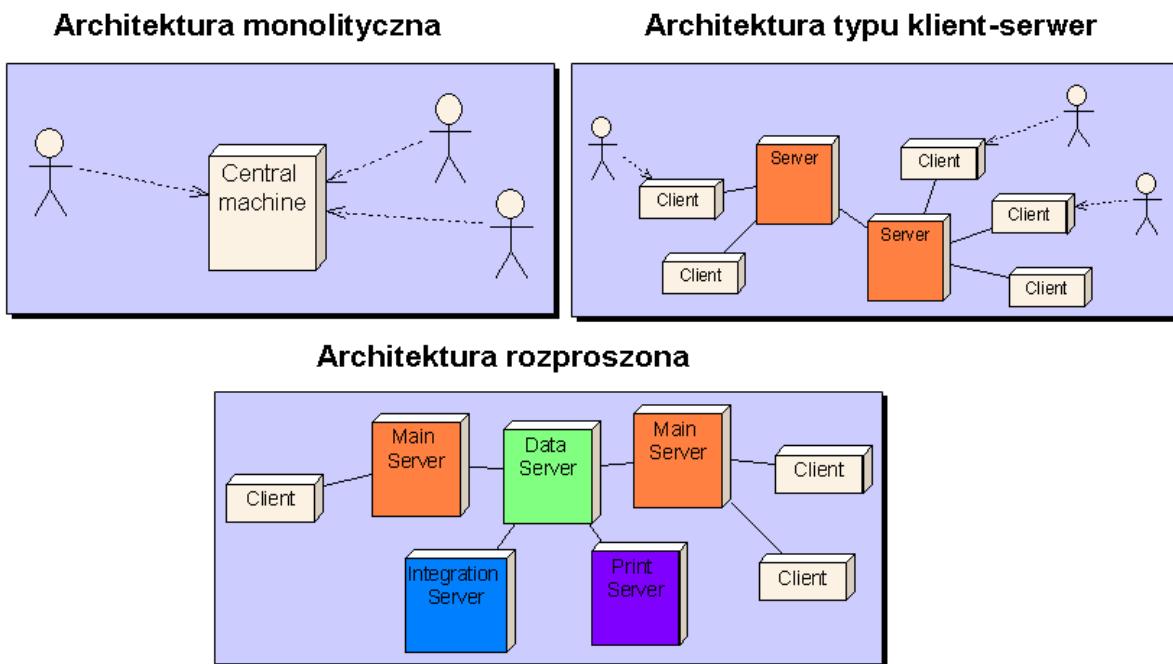
Projekt architektoniczny stanowi swego rodzaju zapis decyzji architektoniczny podjętych przez architekta. Decyzje te są podejmowane na podstawie wymagań oraz doświadczenia architekta w zakresie technologii wytwarzania oprogramowania. Projekt architektoniczny powinien również uwzględniać ograniczenia ekonomiczne oraz technologiczne pozostawiając jednocześnie możliwość wprowadzania zmian i rozszerzeń do początkowych wymagań.

Model architektoniczny, jako wysoko-poziomowy model systemu, powinien być czytelny i zrozumiałym dla projektantów i deweloperów budowanego systemu. Stosowanie modeli graficznych w modelowaniu architektury (np. diagramu komponentów) pozwala na stosowanie abstrakcji – na poszczególnych diagramach można ukryć informację nieistotną z punktu widzenia danego diagramu.

Architektura logiczna i architektura fizyczna

Aby model architektoniczny był kompletny, potrzebujemy projektu systemu z dwóch punktów widzenia: logicznego i fizycznego. Model logiczny dzieli system na logiczne fragmenty (moduły, pakiety, komponenty) które ze sobą współpracują realizując funkcjonalność wymaganą od systemu. Model fizyczny determinuje infrastrukturę fizyczną (maszyny), na których zainstalowane zostaną jednostki logiczne systemu, oraz określa metody komunikacji pomiędzy poszczególnymi węzłami fizycznymi. Te dwa modele stanowią ścisłe powiązane ze sobą aspekty architektury systemu – komponenty logiczne instalowane są w konkretnych węzłach fizycznych.

Logiczny aspekt architektury zostanie przedstawiony bardziej szczegółowo w następnej sekcji na przykładzie architektury komponentowej. Tutaj skupimy się na fizycznym aspekcie architektury.



Rysunek 8-1 przedstawia 3 rodzaje architektury fizycznej systemu. Pierwszym typem architektury jest architektura monolityczna. Rozwiązanie takie jest stosowane w sytuacji gdy wszystkie procesy działają na jednej maszynie a system nie wymaga zdalnego dostępu. Jest to najprostsza architektura, ale z reguły może być stosowana w ograniczonej liczbie przypadków, kiedy np. nie ma wy-

mogu zdalnego jednoczesnego dostępu wielu użytkowników do systemu oraz system nie będzie działał pod dużym obciążeniem.

Kolejnym przykładem jest architektura typu klient-serwer. W takiej sytuacji działanie systemu jest rozdzielone na wielu klientów i jeden typ serwera (serwer taki może działać na pojedynczej maszynie, lub jeśli wymagają tego ograniczenia nałożone na wydajność systemu – rozdzielony na klika maszyn pracujących równolegle).

Najbardziej zaawansowanym typem architektury fizycznej jest architektura rozproszona. W takiej architekturze, przetwarzanie rozdzielone jest na maszyny klientów oraz klika klas połączonych ze sobą serwerów. Przykładem może być wydzielenie serwera bazodanowego na oddzielną maszynę, osadzenie logiki biznesowej systemu na oddzielnym serwerze aplikacyjnym (lub kilku, jeśli system powinien zapewniać wysoką wydajność i niezawodność a pojedyncza maszyna nie jest w stanie temu sprostać) oraz np. osadzenie warstwy prezentacji na jednym bądź więcej serwerach WWW lub wydzielenie usługi drukowania na oddzielną maszynę. Jak widać rozdzielenie komponentów na poszczególne węzły fizyczne może następować według warstwy architektonicznej do której należą (oddzielnny serwer baz danych, serwer aplikacyjny, WWW etc.) lub według ich funkcjonalności (wydzielenie usługi drukowania, oddzielnny serwer integracyjny którego zadaniem jest komunikacja z innymi systemami). Decyzje tego typu mogą być podejmowane nie tylko wymaganiami co do wydajności i niezawodności ale również np. kwestiami bezpieczeństwa (np. serwer integracyjny oraz serwery WWW w wydzielonej podsieci połączonej z Internetem, a pozostałe serwery za firewalllem wewnętrznej sieci organizacji).

Architektura komponentowa

Architektura komponentowa jest architekturą w której system jest podzielony na wiele (dla dużych systemów kilkanaście lub nawet kilkadziesiąt) modułów logicznych, z których każdy stanowi zamkniętą całość z precyzyjnie określona funkcjonalnością i odpowiedzialnością.

Komponent, jak było wspomniane wcześniej, stanowi „czarną skrzynkę” zawierającą wewnątrz wiele mniejszych jednostek funkcjonalności (klas) niewidocznych z zewnątrz. Komponent udostępnia funkcjonalność klas, które zawiera, poprzez dobrze zdefiniowane interfejsy publiczne. Jeśli istnieje taka potrzeba, komponent korzysta z funkcjonalności innych komponentów, ale jedynie poprzez ich udostępnione interfejsy publiczne. W sytuacji takiej, komponenty są ze sobą powiązane relacją zależności (komponent korzystający z interfejsu innego jest zależny od dostarczanej przez niego funkcjonalności).



Rysunek 8-3 Architektura komponentowa

Rysunek 8-3 przedstawia prosty przykład fragmentu komponentowego projektu architektury. W przykładzie tym mamy trzy komponenty. Komponent Presentation korzysta z interfejsu ILogic komponentu Logic a sam dostarcza mu interfejsu IScreen. Komponent Logic korzysta z interfejsu Data komponentu Data. Jak widać komponent Presentation jest zależny od komponentu Logic, komponent Logic polega na obu pozostałych komponentach, natomiast komponent Data jest jedynie dostarczycielem usług dla dwóch pozostałych.

Jakie są zalety stosowania architektury komponentowej? Architektura taka pozwala na łatwiejsze zrozumienie działania systemu jako całości poprzez dekompozycję i abstrakcję poszczególnych części systemu. Dla architektów, projektantów, deweloperów i innych osób zaangażowanych w powstawanie systemu informatycznego, architektura stanowi „spis treści” całego systemu. Jeśli ktoś jest zainteresowany szczegółami jakiejś części systemu, może skorzystać z projektu szczegółowego danego komponentu (nie martwiąc się całą resztą systemu).

Większość dużych systemów jest wykonywana przez kilka niezależnych grup deweloperów. Wykorzystanie architektury komponentowej pozwala na łatwy podział prac pomiędzy grupy. Komponenty, z założenia słabo ze sobą powiązane i niezależne od siebie (z wyjątkiem udostępnienia ustalonych interfejsów) mogą zostać przydzielone do implementacji różnym zespołom mogących pracować zupełnie niezależnie. Każda z takich grup może skoncentrować się na pracy nad swoim komponentem. Przy dobrze zaprojektowanym modelu architektonicznym, grupy te posiadają całą potrzebną im informacje do zrealizowania komponentu. Minimalizuje to potrzebę częstej komunikacji i ścisłej współpracy poszczególnych grup. W końcowych iteracjach wytwarzania oprogramowania, osoba odpowiedzialna za integrację systemu zapewnia odpowiednią jakość poszczególnych komponentów (poprzez ich testowanie). Rozdzielenie funkcjonalności na niezależne jednostki ułatwia testowanie, ponieważ komponenty są odporne na błędy występujące w innych komponentach. Pozwala to na testowania ich jako zamkniętych całości w izolacji od innych komponentów. Architektura komponentowa umożliwia łatwe lokalizowania błędu (z dokładnością do komponentu w którym on występuje) oraz przekazanie błędu do poprawianie odpowiedniemu zespołowi projektowemu.

Architektura komponentowa ułatwia również wprowadzanie zmian i czyni projekt bardziej elastycznym. Dzięki niej, łatwo jest wprowadzać do systemu nowe komponenty (a wraz z nimi nową funkcjonalność) oraz zainstalować komponenty gotowego systemu na różnych maszynach (np. w celu zapewnienia bezpieczeństwa lub odpowiedniego poziomu wydajności/niezawodności).

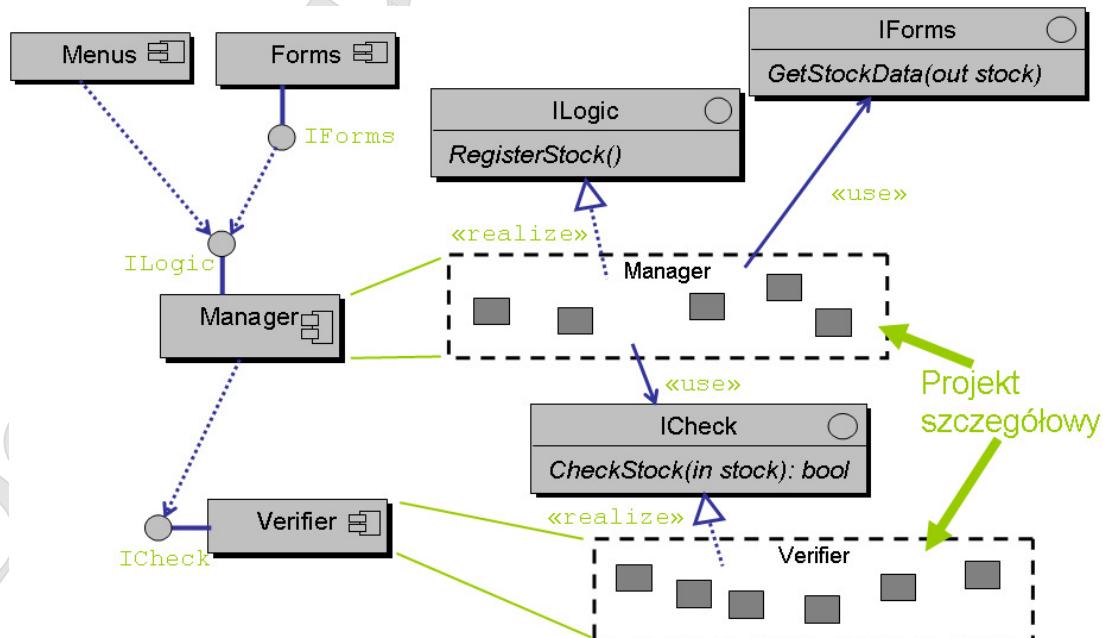
Dobrze zaprojektowana architektura komponentowa, pozwala unikać tzw. „spaghetti code”, czyli sytuacji w której w projekcie istnieje bardzo dużo powiązań pomiędzy poszczególnymi jednostki kodu.

8.1.2. Projektowanie szczegółowe.

Architekt szkicuje ogólną strukturę systemu, dzieli system na komponenty, określa interfejsy za pomocą których te komponenty będą się komunikować oraz uporządkowuje wymianę komunikatorów pomiędzy komponentami (określa dynamikę systemu na poziomie komponentów).

Projektowanie szczegółowe polega na zaprojektowaniu detali zarysowanych na poziomie modelu architektonicznego. To tutaj otwieramy „czarne” skrzynki (komponenty) i projektujemy ich wnętrze. Jak to zrobić? Na poziomie projektu szczegółowego podejmujemy decyzje w jaki sposób zaimplementować interfejsy zdefiniowane dla poszczególnych komponentów oraz ich operacje.

Rysunek 8-4 przedstawia fragment modelu statycznego architektury (modelu komponentów) oraz projekt szczegółowy w kontekście tego modelu. Architekt wydzielił m.in. komponenty Manager oraz Verifier oraz ich interfejsy dostarczane: odpowiednio ILogic i ICheck. Zadaniem projektanta jest stworzenie wewnętrznej struktury oraz określenie wewnętrznej dynamiki tych komponentów.



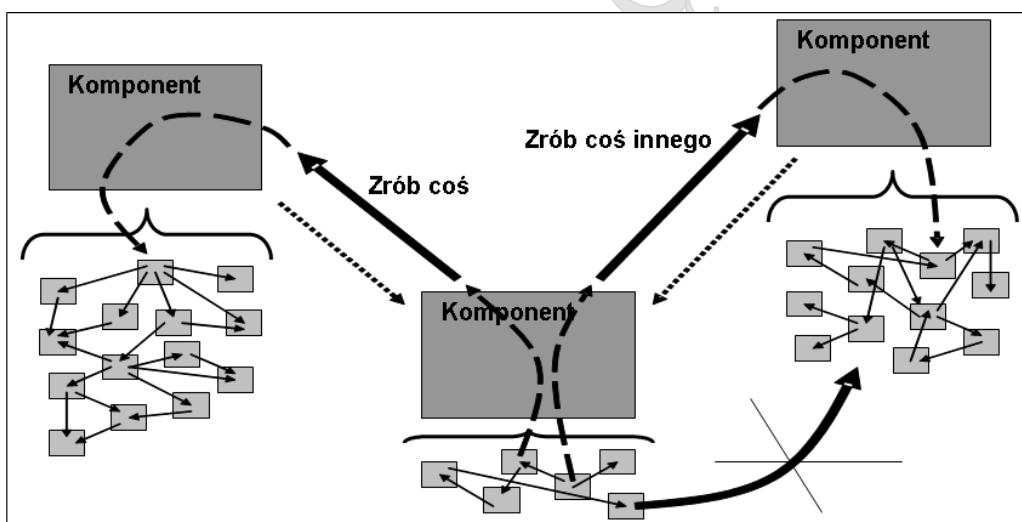
Rysunek 8-4 Projekt szczegółowy w kontekście modelu architektonicznego

Struktura oraz dynamika projektowanych komponentów jest zbyt skomplikowana do wyrażenia jej np. za pomocą tekstu, dlatego podczas projektu szczegółowego również posługujemy się modelami graficznymi w języku UML. Podczas projektowania szczegółowego posługujemy się podobnymi modelami i technikami stosowanymi dla modelu architektonicznego, jednak dla jednostek funkcjonalności w mniejszej skali. Do opisu statycznego wnętrza komponentu (jego struktury) zastosujemy diagram klas. Dynamikę wewnętrz komponentu opiszymy diagramem sekwencji (tym razem na poziomie klas a nie komponentów jak to było w przypadku modeli architektonicznych). Modele na poziomie projektu szczegółowego są bardzo blisko kodu (klasy zamodelowane na tym poziomie odpowiadają konkretnym klasom w kodzie źródłowym), dlatego też projekt szczegółowy powinien być na bieżąco synchronizowany z kodem (wsparcie dla takiej synchronizacji zapewnia narzędzia CASE).

8.2. Projektowanie struktury systemu

8.2.1. Projektowanie struktury modelu architektonicznego

Struktura modelu architektonicznego wyrażona jest za pomocą modelu komponentowego. Pojedynczy komponent stanowi logiczna jednostkę funkcjonalności systemu. Komponent podobnie jak pakiet, zawiera wewnątrz inne elementy (np. klasy). Komponent posiada również pewne zachowanie, dostępne użytkownikom systemu bądź innym komponentom. Zachowanie to (funkcjonalność dostarczana przez komponent) udostępniane jest poprzez porty. Porty są publicznie dostępnymi punktami interakcji pomiędzy komponentem i innymi elementami systemu. Szczegóły sposobu komunikacji pomiędzy komponentami określone są poprzez interfejsy udostępniane przez porty.



Rysunek 8-5 Komponenty

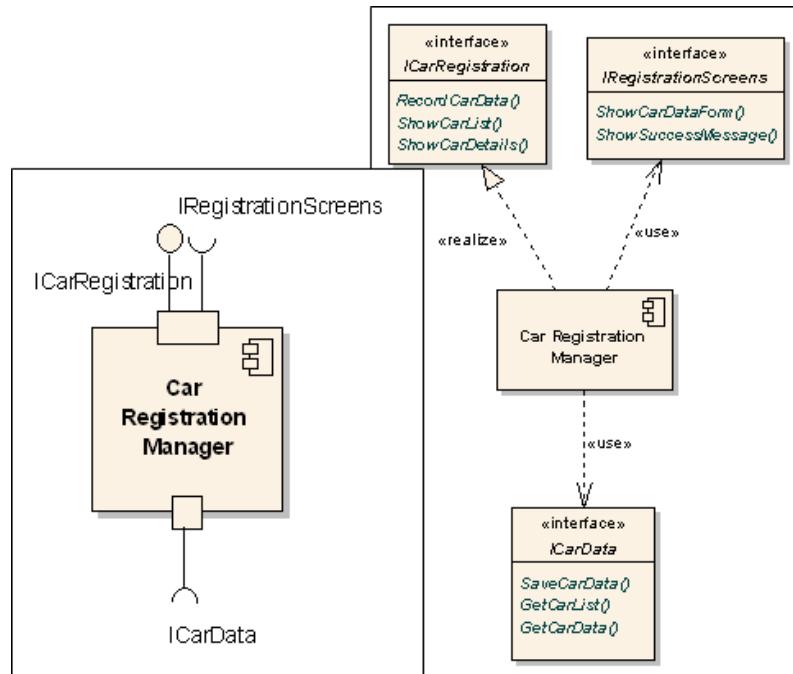
Poprawnie zdefiniowany model komponentowy powinien być zgodny z pewnymi dobrymi praktykami projektowania komponentowego. Poszczególne komponenty powinny być dobrze określonymi zamkniętymi całościami luźno powiązanymi między sobą. W sytuacji, kiedy wymiana komunikatorów pomiędzy dwoma komponentami jest zbyt intensywna i komponenty te powiązane są ze sobą poprzez wiele interfejsów, oznacza to że są zbyt mocno ze sobą związane i być może powinny być połączone w jeden komponent. Z odwrotną sytuacją mamy do czynienia w momencie, kiedy zaprojektujemy komponent odpowiedzialny „za wszystko”. Taki super-komponent powinien być rozdzielony na mniejsze jednostki logiczne powiązanych ze sobą grup funkcjonalności. Logiczne pogrupowanie funkcjonalności systemu na poszczególne komponenty powoduje, że są one pewną abstrakcją fragmentów funkcjonalności systemu. Kolejnym istotnym elementem modelu komponentowego jest poprawne zdefiniowanie komunikacji pomiędzy poszczególnymi komponentami. Komponenty powinny komunikować się poprzez ścisłe określone wąskie interfejsy.

Jak było wspomniane wcześniej, komponenty komunikują się poprzez porty lub porty z udostępnionymi interfejsami. Porty z interfejsami określają komunikację pomiędzy komponentami w opar-

ciu o sygnatury operacji. Porty bez zdefiniowanych interfejsów mogą oznaczać komunikację opartą na innej zasadzie niż wymiana komunikatów (np. komunikacja człowieka z komputerem, ODBC etc.).

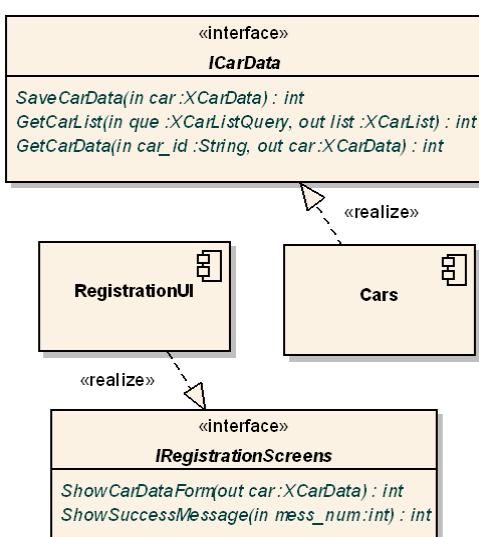
Komponent może posiadać dwa rodzaje interfejsów:

- Interfejs wymagany – definiuje funkcjonalność wymaganą od innych komponentów
- Interfejs dostarczony – definiuje funkcjonalność oferowaną przez ten komponent innym komponentem



Rysunek 8-6 Definiowanie interfejsów komponentów

Diagram komponentów nie powinien prezentować wszystkich szczegółów interfejsów a jedynie zaznaczyć jakie interfejsy są przez komponent dostarczane a jakie wymagane (lewa część rysunku powyżej). Detale interfejsów takie jak sygnatury metod powinny znaleźć się na oddzielnym diagramie klas dla danego komponentu (prawa strona rysunku). Jak widać na powyższym rysunku, na diagramie klas interfejsy dostarczane powiązane są z komponentem relacją <<realize>> a interfejsy wymagane relacją ze stereotypem <<use>>.



Rysunek 8-7 Interfejsy oraz ich operacje

Po zdefiniowaniu ogólnej struktury systemu oraz punktów poprzez które komponenty będą się komunikować (portów oraz interfejsów udostępnianych poprzez te porty) kolejnym etapem projektowania architektonicznego jest uszczegółowienie interfejsów. Operacje interfejsów są "abstrakcyjne", co zaznaczone jest na diagramie kursywą (Rysunek 8-7). Metody implementujące te operacje zrealizowane będą przez klasy znajdujące się wewnątrz komponentu. Sygnatury metod wraz z ich parametrami mają dokładnie taką samą notację jak dla operacji klas. Parametry zdefiniowanych metod mogą posiadać zdefiniowany ich kierunek: in, out lub inout. Informacja ta powoduje, iż specyfikacja sygnatur metod jest bardziej precyzyjna.

Kolejnym zabiegiem doprecyzowującym specyfikację interfejsów jest określenie ograniczeń narzuconych na poszczególne operacje. Takie ograniczenia mogą przyjmować postać warunków początkowych (tj. określenie dopuszczalnych wartości parametrów tylu in i inout przed wywołaniem operacji) oraz warunków końcowych (określających dopuszczalne wartości parametrów inout i out po wykonaniu operacji).

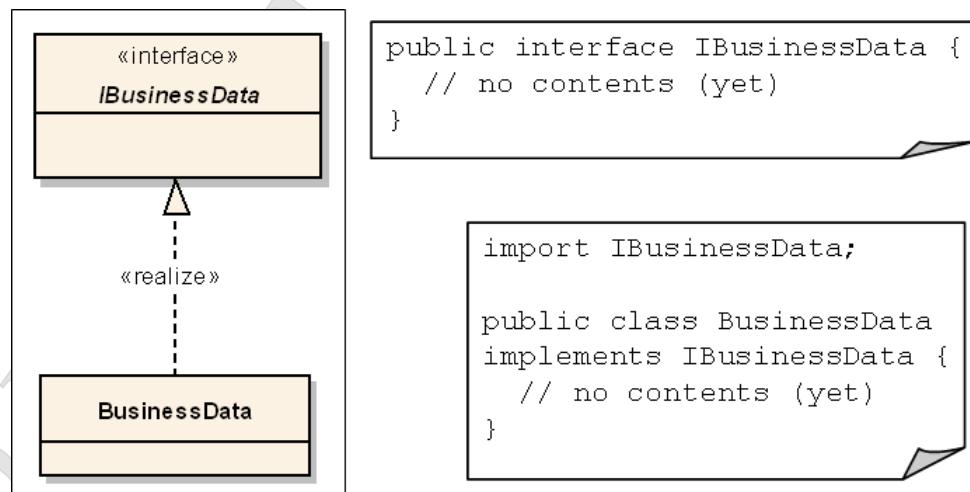
8.2.2. Projektowanie struktury modelu szczegółowego

Projektowania struktury modelu szczegółowego oznacza zaprojektowanie modelu klas dla poszczególnych elementów systemu. Tworząc model klas dla fragmentu systemu powinniśmy wykorzystać klasy zdefiniowane na poprzednich etapach projektowania. Źródłem takich klas będą: pojęcia słownikowe zdefiniowane na etapie specyfikacji wymagań oraz klasy określające dane wymieniane pomiędzy komponentami zdefiniowane na poziomie projektu architektonicznego (parametry operacji interfejsów publicznych komponentów).

Klasy zdefiniowane na poprzednich etapach projektowania powinny zostać uzupełnione klasami wynikającymi z technologii stosowanych do realizacji projektu. Takimi klasami będą np. dla warstwy prezentacji klasy obsługujące mechanikę interfejsu użytkownika takie jak obsługujące zdarzenia oraz zarządzające wyświetlaniem okien. Dla warstwy logiki aplikacji będą to wszelkiego rodzaju klasy zarządzające odpowiedzialne za realizację logiki. Do tego dojdą klasy techniczne np. obsługujące komunikację z bazą danych oraz inne wynikające z wyboru technologii realizującej założone wymagania.

Projektując system na poziomie modelu klas, musimy mieć na uwadze, że klasy przekładają się bezpośrednio na jednostki kodu. Techniki modelowania zastosowane na tym poziomie będą miały istotny wpływ na kształt i działanie kodu realizującego klasy. Wybrane aspekty modelowania na poziomie projektu szczegółowego oraz ich konsekwencje w kodzie zostaną przedstawione poniżej.

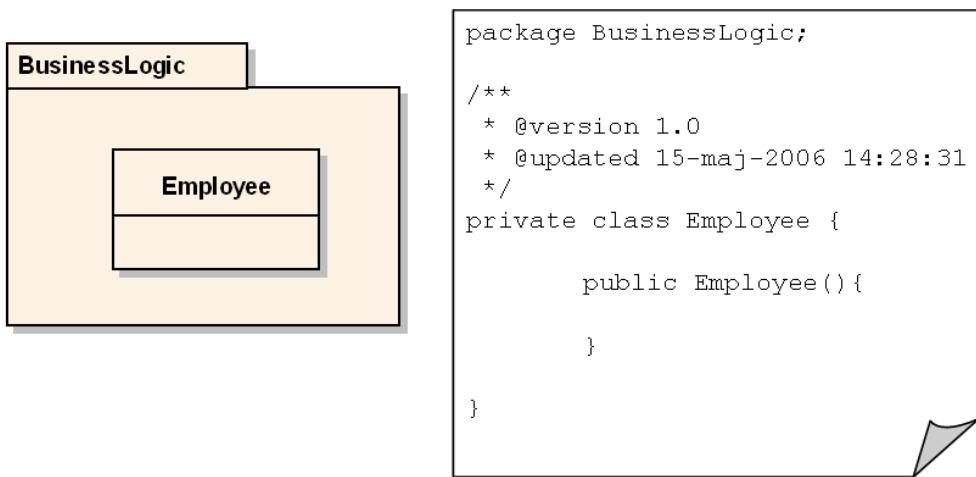
Realizacja interfejsu przez klasę



Rysunek 8-8 Realizacja interfejsu przez klasę

Rysunek 8-8 przedstawia przykład realizacji interfejsu przez klasę. Zarówno klasa jak interfejs są publiczne, co oznacza, że są dostępne dla innych klas wszędzie w systemie. Plik zawierający klasę BusinessData zawiera linie importującą implementowany interfejs IBusinessData.

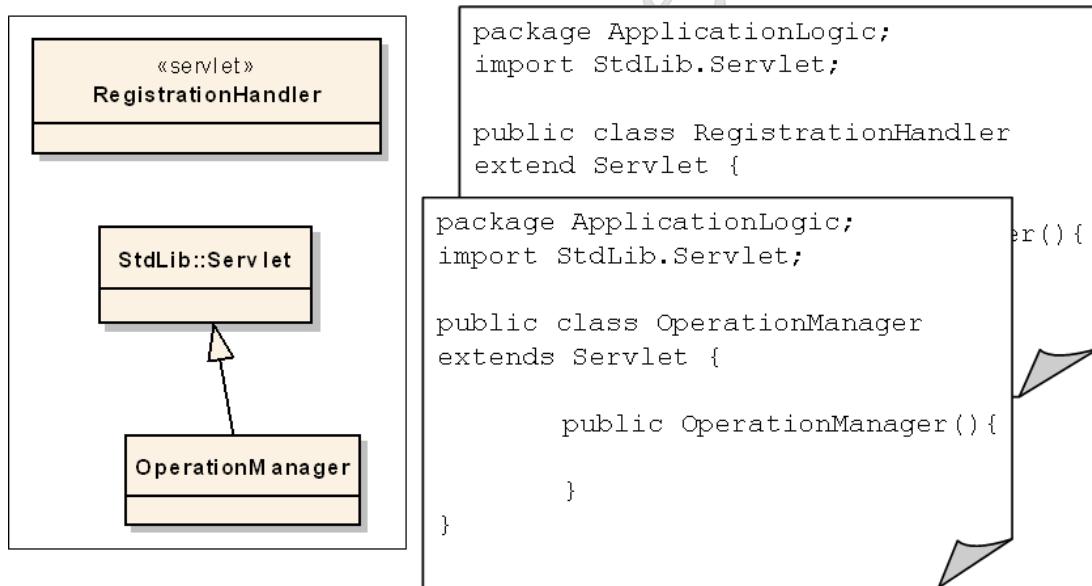
Klasa w pakiecie



Rysunek 8-9 Klasa w pakiecie

Przykład na rysunku powyżej przedstawia klasę umieszczoną w pakiecie. Klasa ta zwiera jedynie konstruktor domyślny, dlatego nie ma potrzeby przedstawiania go w modelu. W podanym przykładzie klasa jest oznaczona jako prywatna, co oznacza iż będzie ona dostępna jedynie wewnątrz pakietu BusinessLogic

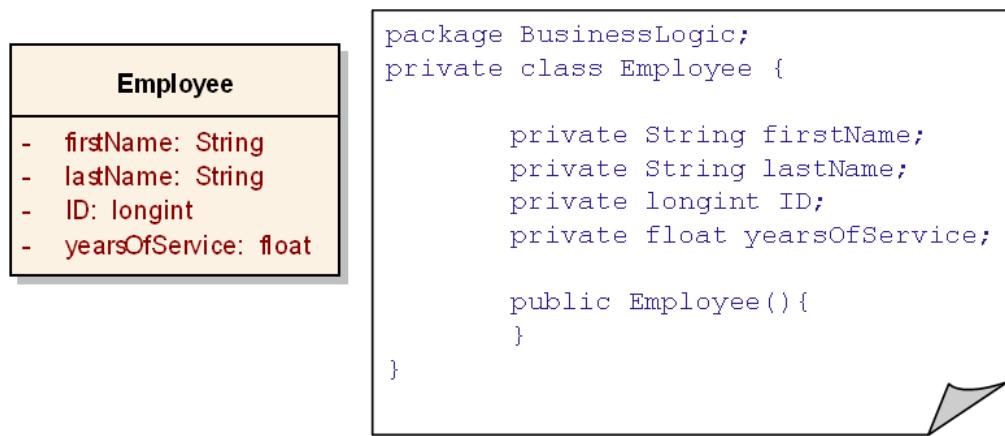
Generalizacja a stereotypy



Rysunek 8-10 Generalizacja kontra stereotypy

W powyższym przykładzie pokazano dwie metody osiągnięcia tego samego celu w kodzie wynikowym. Zarówno klasa RegistrationHandler jak i OperationManager w zamyśle projektanta powinny być servletami. W przypadku klasy OperationManager, projektant jawnie w modelu zaznaczył dziedziczenie klasy z klasy bibliotecznej. W drugim przypadku klasa RegistrationHandler została oznaczona stereotypem <>servlet<>. Informacja ta może zostać wykorzystana podczas generacji kodu, dodając dziedziczenie z klasy servlet bez potrzeby jawnego umieszczania tego dziedziczenia w modelu.

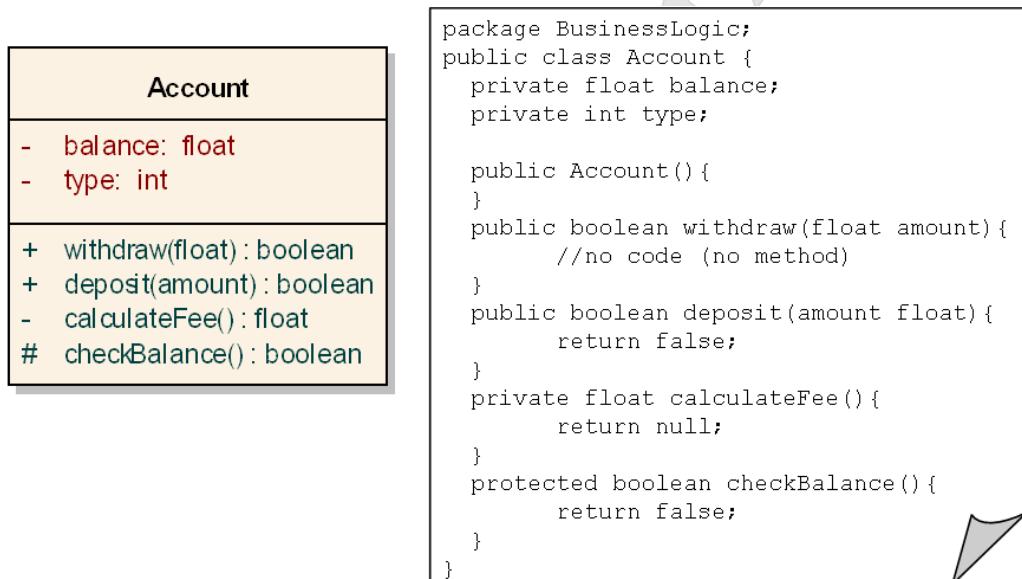
Atrybuty klas



Rysunek 8-11 Atrybuty klasy

Powyższy rysunek przedstawia klasę z atrybutami w modelu oraz jej implementację w kodzie źródłowym. Atrybuty z reguły powinny mieć dostęp prywatny (znak „-” po lewej stronie atrybutu)

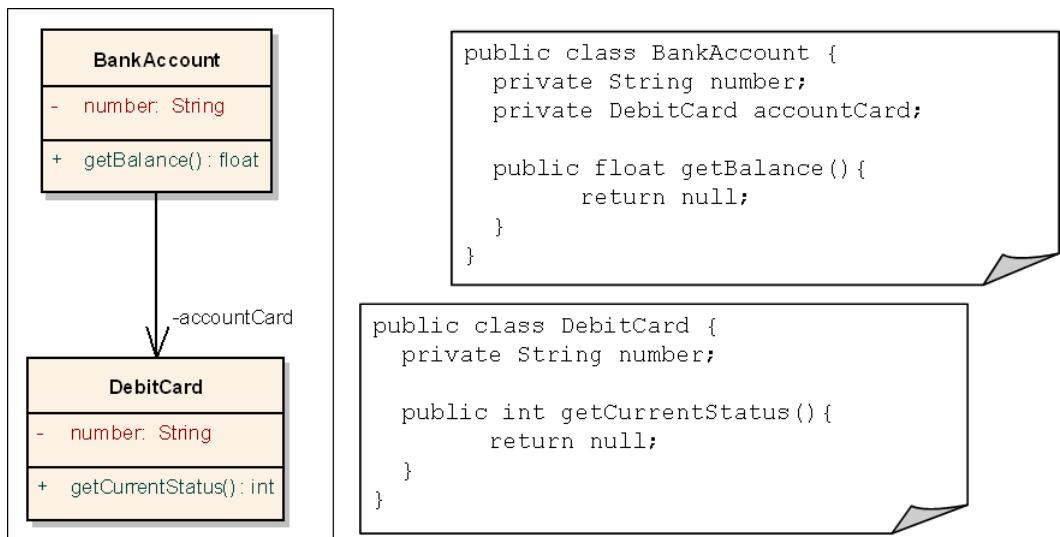
Operacje klas oraz ich widoczność



Rysunek 8-12 Operacje klas oraz ich widoczność

Klasa `Account` w powyższym przykładzie posiada kilka metod o różnym poziomie dostępu. Operacje o widoczności publicznej oznaczone są znakiem „+” po prawej stronie operacji. Metody chronione oznaczane są znakiem „#” a prywatne znakiem „-”.

Asocjacje pomiędzy klasami

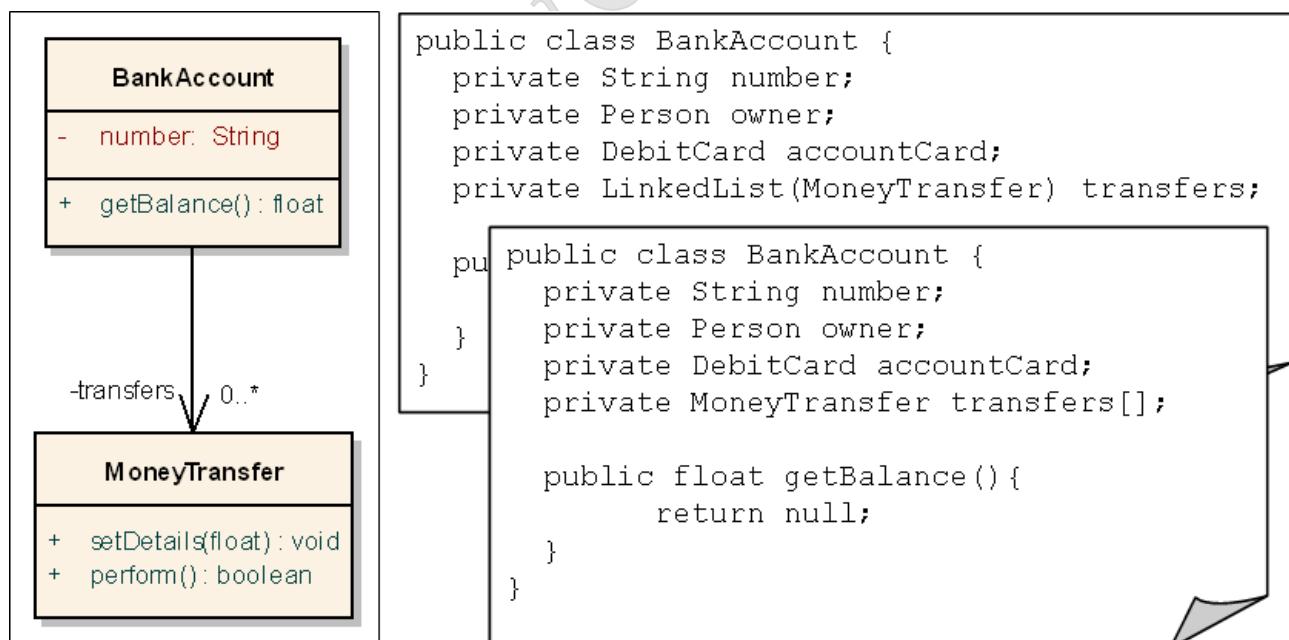


Rysunek 8-13 Asocjacje pomiędzy klasami

Asocjacje w modelu oznaczane są strzałką pomiędzy dwoma klasami. W powyższym przykładzie asocjacja jest nawigowalna od klasy `BankAccount` do klasy `DebitCard`. Oznacza to iż w wygenerowanym kodzie klasa `BankAccount` będzie posiadała atrybut typu `DebitCard` (o nazwie `accountCard`, zgodnie z nazwą końca asocjacji). Klasa `DebitCard` „nie widzi” klasy `BankAccount`, dlatego w jej kodzie nie ma atrybutu typu `BankAccount`.

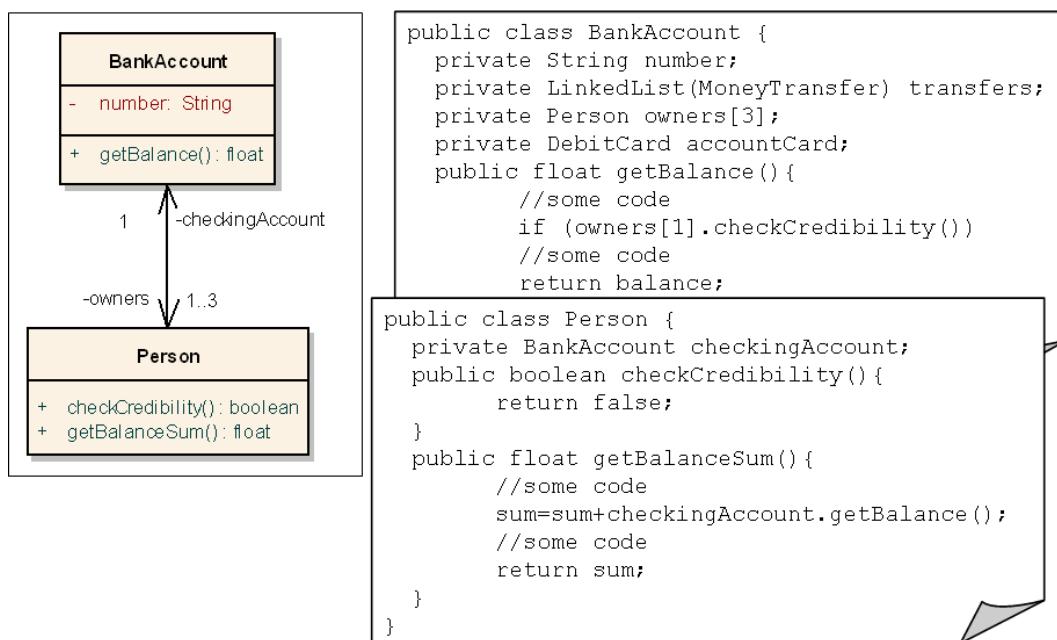
Asocjacje o krotności większej jak jeden

Asocjacje o krotności większej jak jeden (np. `0..*`, `1..*`, `1..20` etc.) przekładają się na występowanie w kodzie źródłowym kolekcji. W poniższym przykładzie przedstawione zostały dwa (z wielu możliwych) sposoby realizacji takiej zależności w kodzie. W pierwszym przypadku klasa `BankAccount` posiada Listę o nazwie `transfers` przechowującą obiekty typu `MoneyTransfer`. W drugim przypadku jest to tablica typu `MoneyTransfer`.



Rysunek 8-14 Asocjacje o krotności większej od jeden

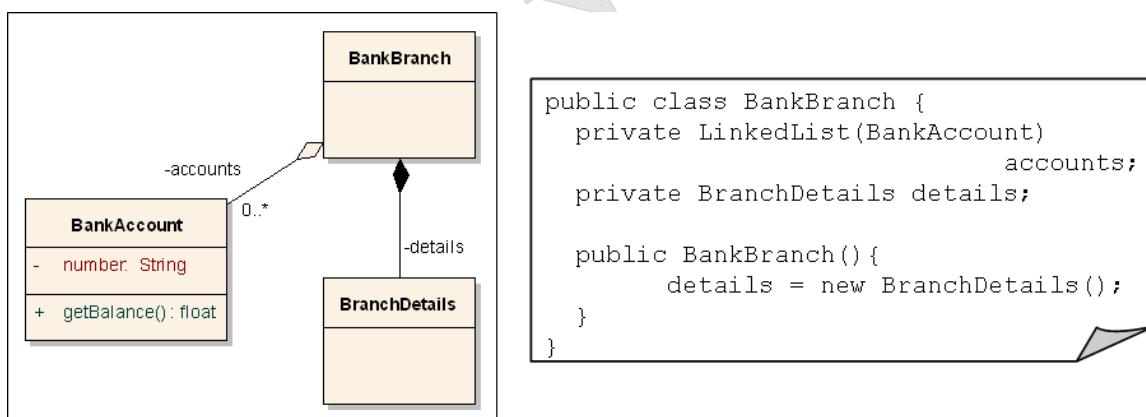
Nawigowalność asocjacji pomiędzy klasami



Rysunek 8-15 Nawigowalność asocjacji pomiędzy klasami

Asocjacja nawigowalna w obu kierunkach oznacza, iż każda z klas może wywoływać metody drugiej. Jak widać w przykładzie, klasa **BankAccount** posiada trzyelementową tablicę obiektów typu **Person** (wynika to z krotności określonej w modelu) a klasa **Person** zawiera atrybut typu **BankAccount**. W widocznych fragmentach kodu obu klas widoczne są wywołania metod drugiej klasy.

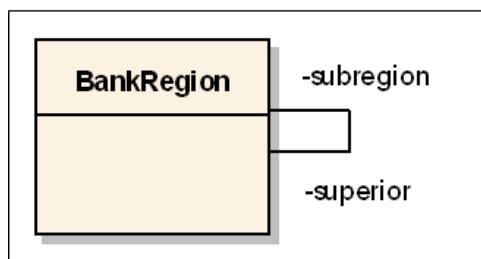
Kompozycja i agregacja



Rysunek 8-16 Kompozycja i agregacja

Agregacja nie przekłada się w żaden określony sposób na kod. Możemy jedynie przypuszczać że np. agregacja jest zawsze nawigowalna w obie strony. Kompozycja oznacza z reguły iż komponent jest tworzony równocześnie z kompozytem w jego konstruktorze.

Asocjacja klasy do samej siebie



```
public class BankRegion {  
    private BankRegion superior;  
    private BankRegion subregion;  
  
    public BankRegion() {  
    }  
}
```

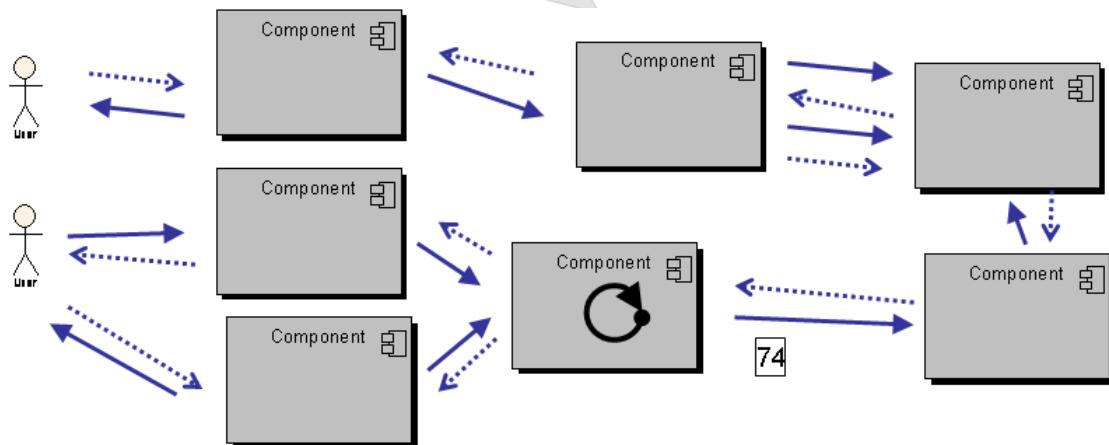
Rysunek 8-17 Asocjacja do siebie

Asocjacja klasy do samej siebie oznacza iż w kodzie wynikowym klasa będzie posiadała dwa atrybuty typu swojej klasy. W przykładzie powyżej klasa BankRegion posiada atrybuty superior i subregion, oba typu BankRegion.

8.3. Projektowanie dynamiki systemu

8.3.1. Projektowanie dynamiki na poziomie architektury

Komponenty są "czarnymi skrzynkami" zawierającymi w sobie określony kod. Kod ten jest odpowiedzialny za zachowanie komponentu widoczne dla innych komponentów oraz użytkowników. Komponent zaczyna „działać” w momencie otrzymania komunikatu z zewnątrz. Komunikat taki instruuje komponent, że ma „coś” zrobić. Po otrzymaniu komunikatu z zewnątrz, komponent zaczyna przetwarzanie, podczas którego może np. zmienić swój stan, przeprowadzić obliczenia lub wysłać komunikat do innego komponentu, jeśli jego współpraca jest konieczna do wykonania przez komponent zadania, które mu „zlecił” komunikat. Po zakończeniu przetwarzania komponent może zwrócić rezultat do obiektu, który wysłał do niego komunikat.



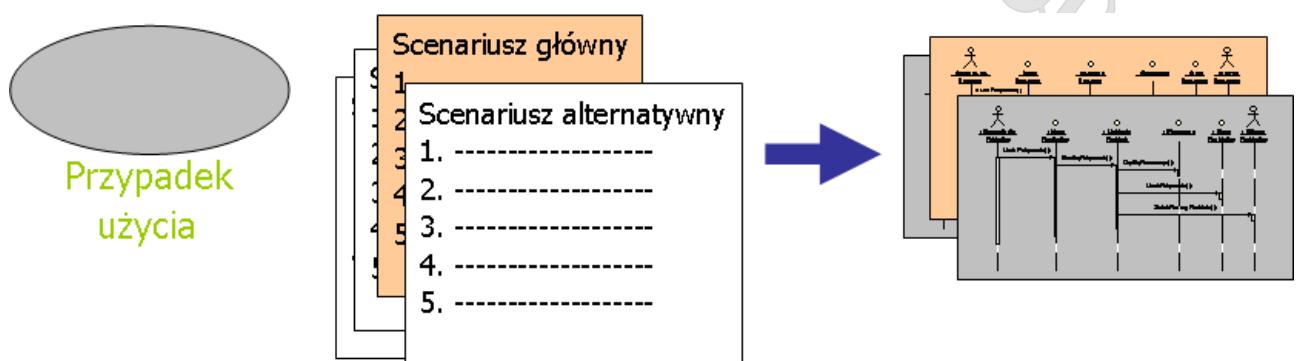
Rysunek 8-18 Komunikacja pomiędzy komponentami

Zachowanie komponentowego systemu oprogramowania opiera się na interakcji pomiędzy komponentami (i komponentami a użytkownikami). Komponenty komunikują się poprzez wysyłanie do siebie komunikatów. Wysłanie takiego komunikatu oznacza z reguły przekazanie kontroli (przetwarzania) z jednego komponentu do drugiego. Komunikaty wysyłane pomiędzy komponentami systemu mogą posiadać parametry (dane, które przekazują z jednego komponentu do drugiego) oraz mogą zwracać wartość.

Sposób komunikacji pomiędzy komponentami definiują interfejsy oraz operacje w nich zdefiniowane. Możliwości komunikacji pomiędzy poszczególnymi komponentami ograniczone są opercjami zdefiniowanymi w ich interfejsach. Oznacza to, że jeśli chcemy dodać nowy typ komunikatu pomiędzy komponentami, musimy zdefiniować nową operację w interfejsie jednego z nich (odbiorcy komunikatu).

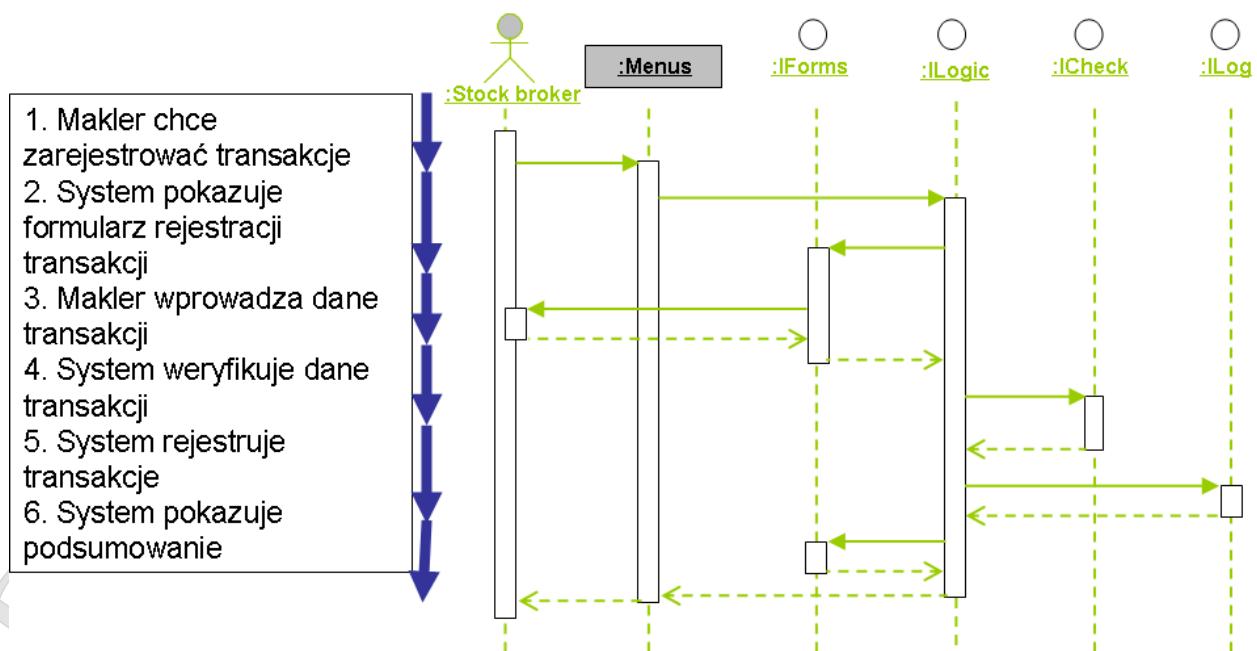
Tym, czego brakuje w statycznym opisie architektury systemu jest uporządkowanie sekwencji wysyłanych komunikatów. Cały proces przetwarzania danych przez system rozpoczyna się z reguły od interakcji użytkownika z systemem i powoduje przekazywanie komunikatów pomiędzy poszczególnymi komponentami kończące się osiągnięciem żądanego wyniku. Określenie dynamiki systemu oznacza zdefiniowanie sekwencji wymiany komunikatów prowadzących do osiągnięcia określonego rezultatu. Sekwencja taka reprezentuje zachowanie systemu obserwowane na wysokim poziomie abstrakcji (bez zagłębiania się w to co dzieje się wewnętrz komponentów).

Najbardziej naturalnym modelem języka UML przydatnym do specyfikacji takiej sekwencji wymiany komunikatów jest diagram sekwencji. Na poziomie architektonicznym, jedynymi liniami życia występującymi na takich diagramach powinny być linie aktorów, komponentów oraz interfejsów (nie powinny tam znajdować się linie życia reprezentujące poszczególne klasy, gdyż na etapie projektowania architektonicznego nie chcemy zagłębiać się w wewnętrzną dynamikę komponentów).



Rysunek 8-19 Przejście od specyfikacji wymagań do opisu dynamiki systemu

Dynamika systemu na poziomie architektury powinna odzwierciedlać interakcję użytkownika zdefiniowaną w wymaganiach funkcjonalnych za pomocą przypadków użycia i ich scenariuszy (patrz Rysunek 8-19). Architekt projektu realizację przypadków użycia poprzez naszkicowanie diagramów sekwencji dla najistotniejszych scenariuszy. Tworzenie diagramów sekwencji dla wszystkich zdefiniowanych scenariuszy nie jest konieczne do wystarczającego zdefiniowania dynamiki systemu, a bez wsparcia narzędziowego jest wręcz zbyt pracochłonne. Scenariusze przypadków użycia przedstawiają sekwencję wymiany komunikatów pomiędzy aktorem a systemem. Diagramy sekwencji na poziomie architektonicznym uszczegółowiąują te sekwencje o wymianę komunikatów pomiędzy komponentami systemu, niezbędną do osiągnięcia celu przypadku użycia.

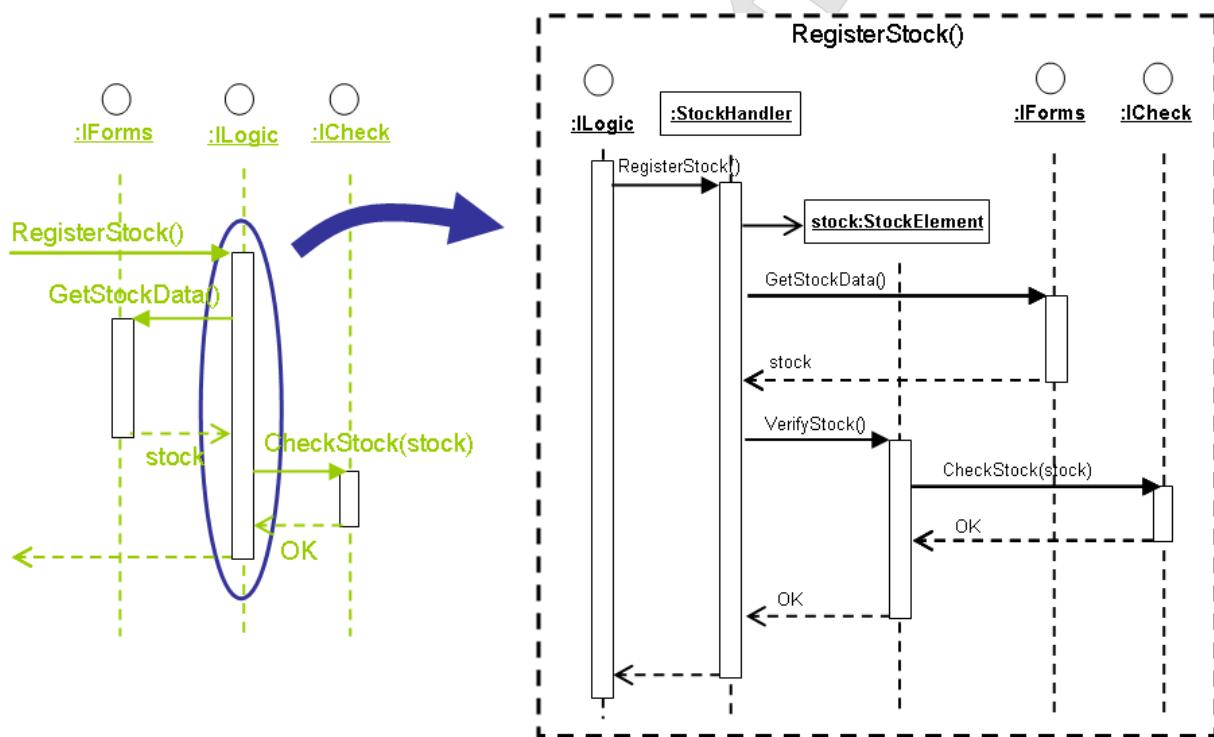


Rysunek 8-20 Odzwierciedlenie zdań scenariusza przypadków użycia na diagramie sekwencji

Każde zdanie, którego podmiotem jest system odzwierciedlane jest w takim diagramie poprzez większą liczbę komunikatów przekazywanych od warstwy najbliższej użytkownikowi (warstwy prezentacji) do warstw znajdujących się niżej (np. warstwy logiki aplikacji, logiki biznesowej i warstwy danych). W momencie interakcji użytkownika z systemem, wysyła on komunikat do komponentu w warstwie prezentacji. Warstwa ta z reguły odpowiedzialna jest za obsługę zdarzeń interfejsu użytkownika i wyświetlanie jego elementów zgodnie z komunikatami przesyłanymi od komponentów warstw niższych. Komunikat przekazany od użytkownika do komponentu warstwy prezentacji z reguły powoduje przekazanie innego komunikatu do komponentu niższej warstwy. W wypadku architektury warstwowej (więcej na ten temat w rozdziale 8.4) warstwa poniżej komunikuje się z komponentami zawartymi w warstwach jeszcze niższych (bardziej oddalonych od użytkownika). Po przetworzeniu całej sekwencji wymiany komunikatów pomiędzy komponentami różnych warstw odpowiedź systemu wraca do użytkownika poprzez warstwę prezentacji.

8.3.2. Projektowanie dynamiki na poziomie projektu szczegółowego

Dynamikę systemu na poziomie projektu szczegółowego definiujemy podobnie jak na poziomie architektonicznym za pomocą diagramów sekwencji. Diagram architektoniczny sekwencji (lewa część rysunku poniżej) przedstawia sekwencję wymiany komunikatów pomiędzy komponentami. W przedstawionym przykładzie, na interfejsie ILogic zostaje wywołana metoda RegisterStock(). Z diagramu możemy wyczytać, że wewnątrz metody RegisterStock() następuje wysłanie komunikatu do innego komponentu (metoda CheckStock z parametrem stock w interfejsie ICHECK). Na poziomie opisu dynamiki architektury nie wiemy nic więcej na temat działania tej metody.

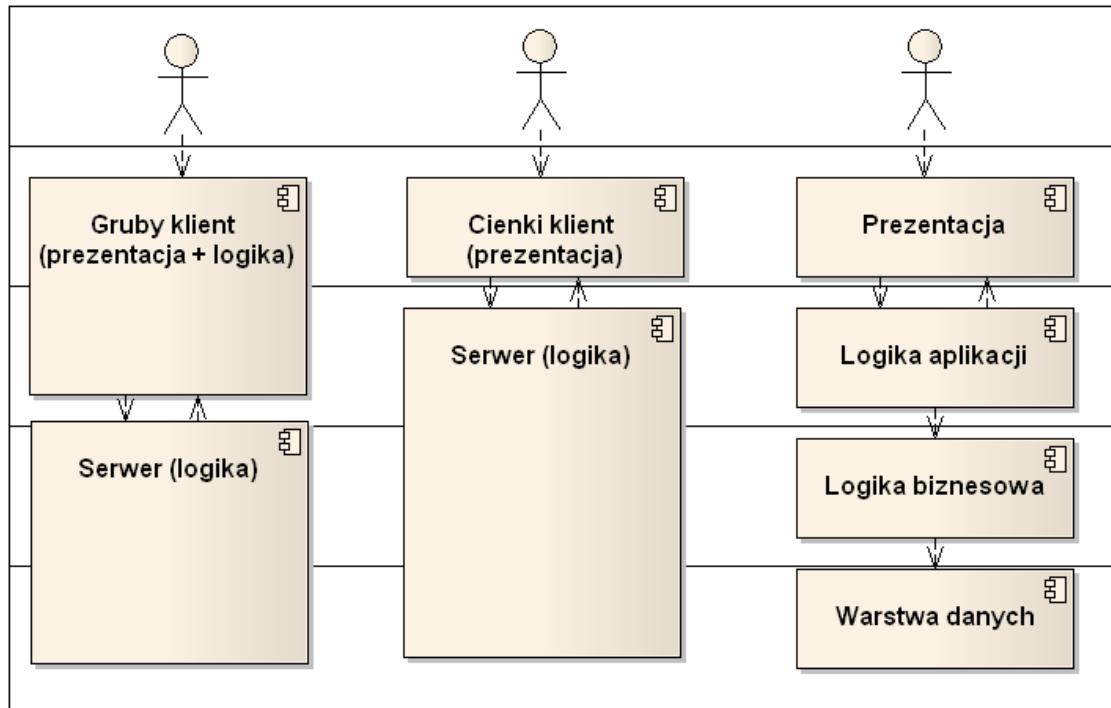


Rysunek 8-21 Diagram sekwencji na poziomie projektu szczegółowego

Zadaniem projektowania dynamiki systemu jest określenie detali tej metody na poziomie projektu szczegółowym. Na tym etapie projektowania znamy już „wnętrze” komponentu implementującego interfejs ILogic. Projekt struktury komponentu jest punktem wyjścia do opisania jego dynamiki. Prawa część rysunku powyżej przedstawia gotowy diagram sekwencji dla metody RegisterStock. Jak widać, interfejs zewnętrzny komponentu jest implementowany przez klasę StockHandler. Podczas wykonania metody RegisterStock, najpierw tworzony jest obiekt stock klasy Stock element, następnie sterowanie przekazywane jest do innego komponentu (poprzez interfejs Form) w celu pozyskania z interfejsu użytkownika danych dla nowo utworzonego elementu. Po powrocie sterowania do aktualnego komponentu, jest ono przekazywane do nowo utworzonego obiektu stock (cały czas jesteśmy wewnątrz rozważanego komponentu), który z kolei komunikuje się z innym komponentem w celu weryfikacji danych.

Jak widać na powyższym przykładzie, diagram sekwencji na poziomie projektu szczegółowego pozwala na opisanie dynamiki poszczególnych metod interfejsów publicznych komponentów. Opis taki pokazuje wewnętrzną dynamikę działania komponentu. Na diagramie sekwencji dla metody interfejsu danego komponentu pokazujemy sekwencje wymiany komunikatów wewnętrz komponentu pomiędzy klasami go implementującymi, oraz komunikację z innymi komponentami (ale już tylko na poziomie wywołania metod interfejsów publicznych tych komponentów).

8.4. Warstwowy model architektury oprogramowania.



Rysunek 8-22 Modele warstwowe architektury

Przetwarzanie w złożonych systemach z reguły rozproszone jest pomiędzy różne warstwy architektoniczne, jedne „bliszce” ostatecznemu użytkownikowi, inne bardziej od niego odległe. Najczęściej wykorzystywany modelami są architektura dwu-warstwowa (klient-serwer) lub architektury wielowarstwowe. Rysunek 8-22 przedstawia schematycznie dwa rodzaje architektury dwuwarstwowej oraz odmianę architektury wielowarstwowej – architekturę cztero warstwową. W wypadku architektury dwuwarstwowej, jedną z jego odmian może być architektura z „grubym” klientem. Przy takim podejściu, aplikacja klienta zawiera w sobie warstwę prezentacji (wszystkie elementy graficznego interfejsu wyświetlane użytkownikowi) oraz logikę aplikacji (całą „mechanikę” obsługującą dialog pomiędzy systemem a użytkownikiem oraz odpowiedzialną za odpowiednie reagowanie na interakcję użytkownika oraz odpowiedzi warstwy serwerowej). Aplikacja uruchomiona po stronie serwera odpowiedzialna jest za wykonywanie metod biznesowych wywoływanych przez logikę aplikacji zawartą w grubym kliencie.

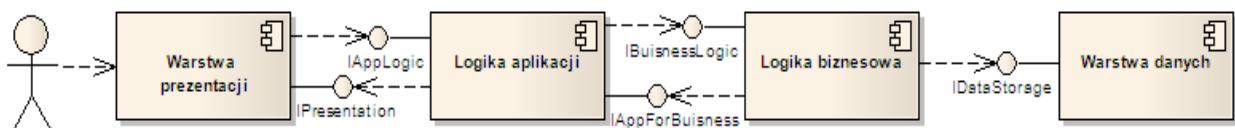
Kolejną odmianą architektury dwuwarstwowej jest architektura z „cienkim klientem”. Przy takiej architekturze, aplikacja kliencka odpowiedzialna jest jedynie za komunikację z użytkownikiem poprzez GUI. Cała mechanika tej interakcji (logika aplikacji) obsługiwana jest przez pojedynczą warstwę po stronie serwera. W obu wypadkach (cienkiego i grubego klienta) aplikacja po stronie serwera nie ma wyraźnie wydzielonych warstw architektonicznych.

W przypadku architektury wielowarstwowej zadania aplikacji są grupowane w komponentach poszczególnych warstw. Najczęściej spotykaną architekturą tego typu (szczególnie w aplikacjach Webowych dostępnych przez przeglądarkę) jest architektura czterowarstwowa.

W typowej architekturze czterowarstwowej wyróżniamy następujące warstwy (Rysunek 8-23):

- Warstwa prezentacji – zawiera elementy interfejsu użytkownika (formularze, komunikaty etc.) oraz przetwarza dane wejściowe i wyświetla dane wyjściowe użytkownikowi
- Warstwa logiki aplikacji – obsługuje wykonanie scenariuszy zgodnie z ich definicją w wymaganiach funkcjonalnych
- Warstwa logiki biznesowej – przetwarza dane zdefiniowane w wymaganiach słownikowych
- Warstwa danych – pozwala na przechowywanie i pobieranie danych przetwarzanych przez warstwę logiki biznesowej

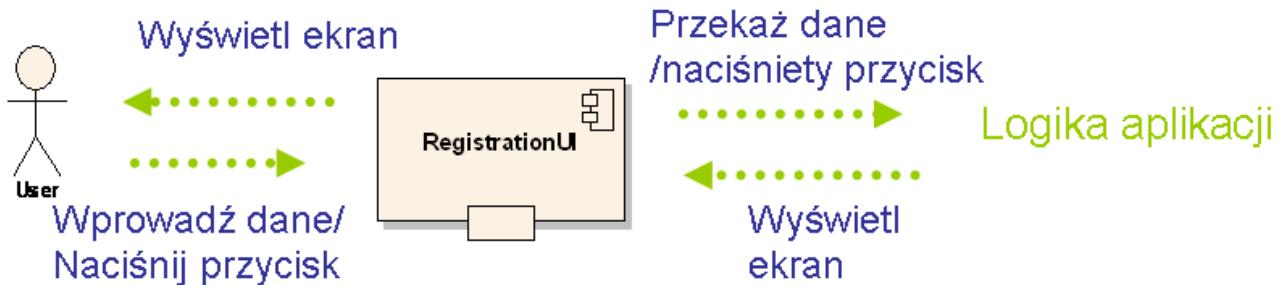
W architekturze takiej mamy przejrzyście wydzielone warstwy, każda o innej odpowiedzialności. Analogicznie do architektur z cienkim klientem, warstwą najbliższą użytkownikowi i odpowiedzialną jedynie za komunikację z użytkownikiem poprzez graficzny interfejs jest warstwa prezentacji. W odróżnieniu od architektury dwu warstwowej, gdzie logika aplikacji zaimplementowana jest w warstwie prezentacji (gruby klient) lub w pojedynczej warstwie serwerowej razem z pozostałą funkcjonalnością systemu, w architekturze czterowarstwowej logika aplikacji wydzielona jest jako oddzielną warstwę, komunikującą się jedynie z warstwą prezentacji i warstwą poniżej. Poniżej warstwy logiki aplikacji istnieje wydzielona warstwa biznesowa odpowiedzialna za realizację metod biznesowych wynikających z wymagań (ta warstwa robi to do czego system jest zaprojektowany). Warstwa logiki biznesowej korzysta z najwyższej warstwy – warstwy dostępu do danych – w celu pobrania lub przechowania danych na których operuje.



Rysunek 8-23 Warstwowy model architektury oprogramowania

Istotą architektury warstwowej jest to, iż poszczególne komponenty mogą się komunikować tylko z komponentami warstwy poniżej lub warstwy powyżej, przy założeniu że każda z tych warstw ma pewną określoną odpowiedzialność i zakres działania. Rysunek 8-1 przedstawia uproszczony model komponentowy w architekturze czterowarstwowej (istnieje tutaj tylko po jednym komponencie w każdej z warstw). Jako widać na rysunku, warstwa prezentacji komunikuje się z użytkownikiem (poprzez wyświetlanie mu komunikatów, formatek oraz innych elementów interfejsu użytkownika oraz poprzez odbieranie zdarzeń wywołanych przez użytkownika w tym interfejsie) oraz z warstwą logiki aplikacji. Warstwa logiki aplikacji komunikuje się z warstwą prezentacji „nakazując” jej odpowiednią reakcję na zdarzenie wewnętrz systemu bądź interakcję użytkownika, oraz z warstwą biznesową wywołując na niej poszczególne metody biznesowe, przekazując do tych metod dane wprowadzone przez użytkownika w warstwie prezentacji oraz odbierając wyniki operacji biznesowych. Warstwa biznesowa z reguły jest serwerem dla warstwy logiki aplikacji (warstwa biznesowa udostępnia warstwie logiki aplikacji metody biznesowej sama nie wywołując żadnych metod na warstwie logiki aplikacji) oraz klientem warstwy danych (wywołując na niej operacje i odbierając wyniki operacji). Warstwa danych, stanowiąca serwer dla warstwy biznesowej, odpowiedzialna jest za przechowywanie i dostęp do danych. Z reguły ta warstwa implementuje dostęp do bazy danych. Poniższe sekcje charakteryzują poszczególne warstwy architektury czterowarstwowej.

Warstwa prezentacji



Rysunek 8-24 Warstwa prezentacji

Warstwa prezentacji jest odpowiedzialna za prezentowanie wyników użytkownikowi i akceptowanie danych od niego.

Komponenty tej warstwy zawierają klasy obsługujące poszczególne ekranы. Klasy te generują ekranы i przetwarzają zdarzenia w tych ekranach.

Warstwa prezentacji powinna być odpowiedzialna jedynie za prezentowanie i akceptowanie danych od/do użytkownika – i nic innego. Częstym błędem popełnianym przy projektowaniu/implementacji systemu jest umieszczanie części logiki biznesowej lub logiki aplikacji w komponentach warstwy prezentacji.

Warstwa logiki aplikacji



Rysunek 8-25 Warstwa logiki aplikacji

Warstwa aplikacji kontroluje działanie całej aplikacji poprzez wywoływanie odpowiednich akcji w pozostałych warstwach zgodnie z ustaloną kolejnością – odpowiadającą wymaganiom funkcjonalnym

Komponenty tej warstwy zawierają klasy odpowiedzialne za zarządzanie odpowiednimi sekwencjami zdarzeń zdefiniowanymi w scenariuszach przypadków użycia. Klasy te „wiedzą” jak zareagować gdy zostanie naciśnięty dany klawisz w danej sytuacji. Reagują również odpowiednio na stany systemu (warstwy logiki biznesowej). Warstwa logiki aplikacji „instruuje” pozostałe warstwy.

Warstwa logiki biznesowej

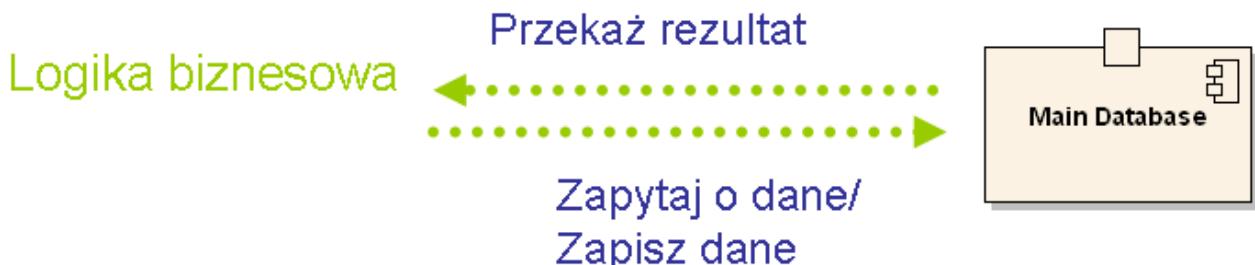


Rysunek 8-26 Warstwa logiki biznesowej

Warstwa logiki biznesowej odpowiedzialna jest za przetwarzanie danych zgodnie z żądaniami logiki aplikacji. Operacje wykonywane przez logikę biznesową na danych są zgodne z regułami biznesowymi zdefiniowanymi w wymaganiach słownikowych.

Komponenty tej warstwy zwierają klasy reprezentujące pojęcia ze słownika dziedziny. Klasy te posiadają operacje które przetwarzają dane zawarte w obiektach tych klas. Warstwa logiki biznesowej jest odpowiednim miejscem do integracji z innymi aplikacjami. Zewnętrzne aplikacje powinny integrować się z naszą poprzez wywoływanie metod biznesowych na tej warstwie.

Warstwa danych



Rysunek 8-27 Warstwa logiki biznesowej

Warstwa danych pozwala na trwałe przechowywanie i odczytywanie danych potrzebnych przez warstwę logiki biznesowej. Komponenty tej warstwy zawierają bazy danych, pliki, repozytoria danych, bazy wiedzy itd.

Komponenty relacyjnej bazy danych (najczęściej występujące) zawierają tabele i relacje. Komponenty takie obsługują zapytania języka zapytań (np. SQL)

Możemy integrować różne aplikacje dając im dostęp do tej samej warstwy danych. Nie jest to jednak dobry pomysł, gdyż struktura bazy danych może się często zmieniać.

8.5. Podstawy wzorców projektowych.

Wzorzec projektowy jest formalnym sposobem opisywania poprawnych rozwiązań często pojawiających się problemów projektowych. Koncepcja wzorca projektowego wywodzi się z wzorców projektowych w architekturze zaproponowanych przez architekta Christophera Alexandra. Koncepcja ta nie przyjęła się w architekturze, ale została zaadaptowana w różnych innych dziedzinach, m.in. w inżynierii oprogramowania. Wzorce projektowe w inżynierii oprogramowania zostały spopularyzowane przez „Bandę Czterech” (Gamma, Helm, Johnson, oraz Vlissides) w ich książce „Inżynieria oprogramowania: Wzorce projektowe”

Użycie wzorców projektowych w inżynierii oprogramowania czyni kod prostszym, bardziej wydajnym, czytelniejszym i w uogólnieniu poprawia jego jakość. Wzorzec projektowy oparty jest o diagram klas przedstawiający kilka klas oraz relacje pomiędzy nimi i przedstawia pewien sposób działania. Wzorzec jako taki jest abstrakcyjny (oderwany od Konkretnego zastosowania), ale każdy wzorzec posiada wiele przykładów konkretnego użycia.

Wzorce projektowe możemy poklasyfikować według ich przeznaczenia na trzy grupy:

- wzorce konstrukcyjne - opisujące proces tworzenia nowych obiektów; ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów
- wzorce strukturalne - opisujące struktury powiązanych ze sobą obiektów
- wzorce czynnościowe - opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów

Poniżej przedstawiamy przykłady wzorców poszczególnych typów wraz z ich krótką charakterystyką.

Wzorce konstrukcyjne:

- Abstract factory (fabryka abstrakcji) – dostarcza interfejsu umożliwiającego tworzenie rodzin spokrewnionych lub uzależnionych od siebie obiektów bez konieczności określania ich konkretnych klas
- Builder (budowniczy) – oddziela konstruowanie złożonych obiektów od ich reprezentacji, w celu umożliwienia budowania różnych reprezentacji obiektów za pomocą tych samych zasad konstrukcyjnych
- Factory method (Metoda fabrykująca) – klasa decyzyjna, która w zależności od otrzymanych danych, zwraca obiekt jednej z wielu klas pochodnych abstrakcyjnej klasy podstawowej
- Prototype (prototyp) – pozwala na klonowanie istniejących instancji klas. Właściwości instancji mogą następnie być modyfikowane za pomocą metod publicznych
- Singleton – klasa zezwalająca na utworzenie tylko jednego jej obiektu o dostępie publicznym

Wzorce strukturalne:

- Adapter – używany do stworzenia nowego interfejsu klasy, taka by dostosować go do interfejsu innej klasy
- Bridge (most) – oddziela interfejs obiektu od jego implementacji, dzięki temu mogą być niezależnie modyfikowane
- Composite (kompozyt) – tworzy obiekt, który jest kompozycją obiektów. Każdy z tych obiektów może być obiektem prostym lub kompozytem
- Decorator (dekorator) – pozwala dynamicznie dodawać nowe cechy do obiektu
- Façade (fasada) – używany do utworzenia pojedynczej klasy reprezentującej złożony system
- Flyweight (waga piórkowa) – Stosowany dla obiektów współdzielonych.
- Proxy (pośrednik) – tworzy prosty obiekt zastępujący obiekt bardziej złożony, który może być dostępny w późniejszym czasie.

Wzorce czynnościowe:

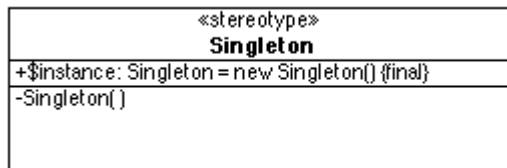
- Chain of responsibility (łańcuch odpowiedzialności) – uwalnia nadawcę żądania od konieczności powiązania żądania z jednym odbiorcą. Obiekty łańcucha odpowiedzialności przekazują kolejno między sobą żądanie do czasu, aż zostanie ono rozpoznane i obsłużone przez jeden z nich
- Command (polecenie) – nadaje żądaniu formę obiektu, pozwala na zrealizowanie dziennika poleceń i wycofanie wykonanych operacji
- Interpreter – definiuje jak wprowadzić do programu składowe języka na podstawie jego gramatyki i opisów semantycznych
- Iterator – formalizuje sposób poruszania się po dowolnej kolekcji
- Mediator – definiuje w jaki sposób można uprościć komunikację pomiędzy obiektami z użyciem osobnego obiektu, zapobiegając jawnym odwołaniom między nimi.
- Memento - udostępnienia stan wewnętrznego obiektu innym obiektom w taki sposób, aby nie naruszyć jego hermetyzacji. Działanie takie umożliwia odtworzenie stanu obiektu
- Observer (obserwator) – definiuje sposób powiadomienia o zmianach stanu obiektu
- State (stan) – umożliwia obiektowi zmianę jego zachowania w wyniku zmiany jego stanu wewnętrznego
- Strategy (strategia) – definiuje rodzinę obudowanych algorytmów i umożliwia ich wymianę

- Template (szablon) – dostarcza abstrakcyjnej definicji algorytmu
- Visitor (wizytator) – pozawala na zdefiniowanie nowej operacji bez zmian w klasach elementów, do których operacja się odnosi

Następne sekcje charakteryzują bardziej szczegółowo kilka często wykorzystywanych wzorców.

Singelton

Wzorzec singelton jest wzorcem konstrukcyjnym. Zadaniem tego wzorca jest zapewnienie, że istnieje tylko jedna instancja klasy oraz dostarczenie globalnego punktu dostępu do tej klasy. Dodatkowo, wzorzec Singelton gwarantuje utworzenie obiektu dopiero w momencie jego pierwszego użycia.



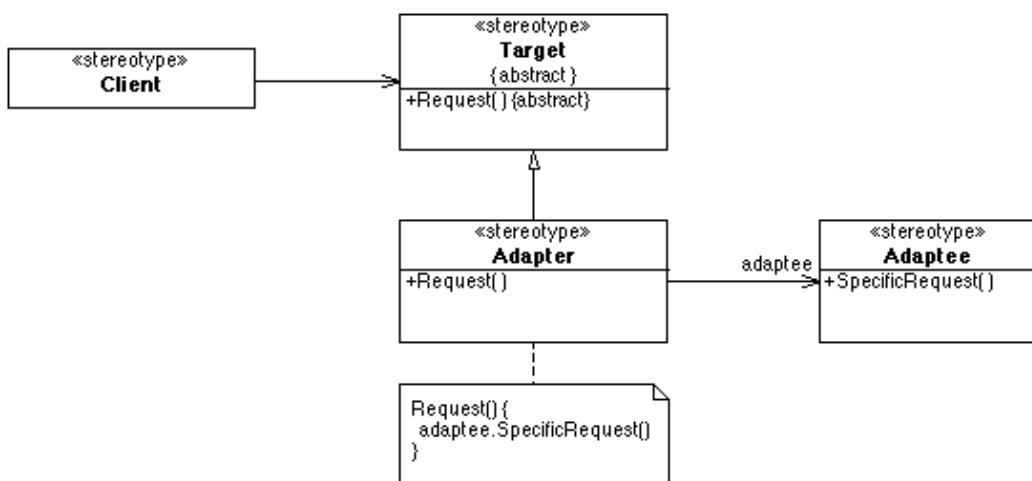
Rysunek 8-28 Wzorzec singelton

Singelton implementuje się poprzez stworzenie klasy posiadającej prywatny lub chroniony konstruktor i statyczną metodę, która najpierw sprawdza czy instancja tej klasy już istnieje i zależnie od wyniku sprawdzenia zwraca istniejącą instancję lub ją. Instancję przechowuje się w prywatnym lub chronionym, statycznym polu, do którego dostęp ma tylko opisana wyżej metoda, która jest jedyną drogą pozyskania instancji obiektu singletonu. Cały proces jest niewidoczny dla użytkownika. Nie musi on wiedzieć, czy instancja już istnieje czy dopiero jest tworzona.

Adapter

Wzorzec Adapter konwertuje interfejs jednej klasy na interfejs innej. Wzorzec adaptera stosowany jest najczęściej w przypadku, gdy wykorzystanie istniejącej klasy jest niemożliwe ze względu na jej niekompatybilny interfejs. Drugim powodem użycia może być chęć stworzenia klasy, która będzie współpracowała z klasami o nieokreślonych interfejsach.

Istnieją dwa sposoby realizacji wzorca Adapter: poprzez dziedziczenie i poprzez kompozycję. W pierwszym przypadku klasę adaptera dziedziczymy z klasy która posiada niekompatybilny interfejs oraz dopisujemy metody spełniające wymagany interfejs. W przypadku tego adaptera wywołanie funkcji jest przekierowywane przez jego interfejs do interfejsu obiektu adaptowanego. W drugim przypadku klasa adaptera implementuje żądany interfejs oraz zawiera w sobie klasę adaptowaną. W tym wypadku żądanie klienta adapter konwertuje na jedno bądź więcej wywołań skierowanych do obiektu adaptowanego.

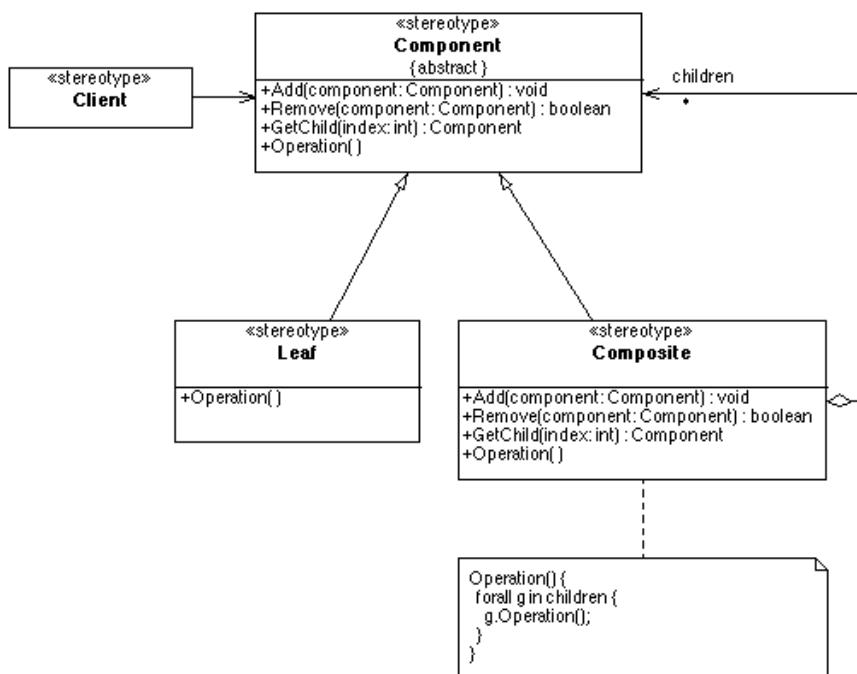


Rysunek 8-29 Wzorzec Adapter

Przykład adaptera opartego na kompozycji przedstawia Rysunek 8-29. Klasa Adapter implementuje żądaną interfejs Target dziedzicząc z niego metodę request oraz zawiera obiekt adapter typu adaptowanej klasy. W wypadku wywołania metody request na obiekcie typu Adapter, ten wywołuje metodę SpecificRequest na prywatnym obiekcie klasy Adaptee a wynik przekazuje do klasy Client.

Kompozyt

Kompozyt jest jednym ze strukturalnych wzorców projektowych, którego celem jest składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt. Kompozyt jest kolekcją obiektów z których każdy może być kompozytem lub pojedynczym obiektem. Kompozyt jest wykorzystywany wszędzie tam gdzie reprezentujemy strukturę hierarchiczną lub w uogólnieniu strukturę drzewiastą.

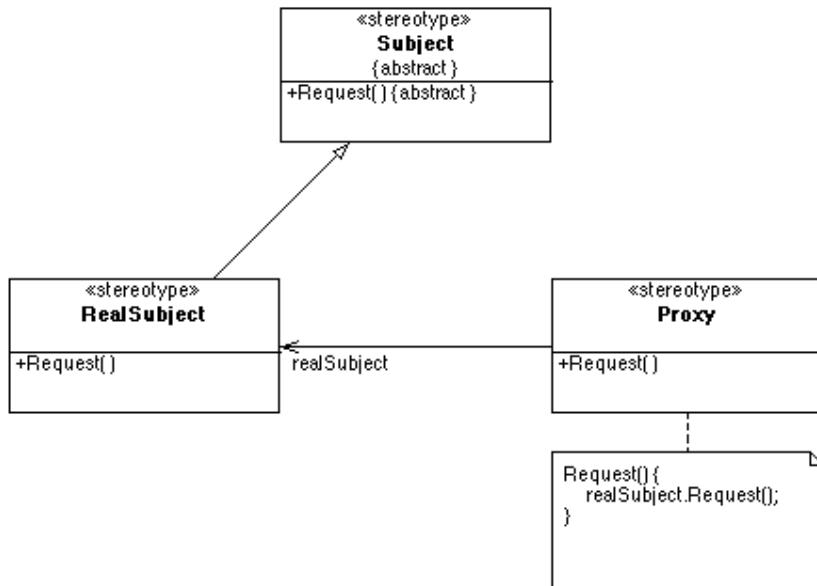


Rysunek 8-30 Wzorzec Composite

Jednym z możliwych rozwiązań implementacji wzorca kompozyt jest utworzenie klasy abstrakcyjnej lub interfejsu definiującego wspólne dla węzłów i liści metody. Rysunek 8-30 przedstawia diagram klas takiego rozwiązania. Elementy kompozytu są dostępne dla klas korzystających z nich poprzez interfejs Component, zawierający metody wspólne dla liści i węzłów. Z tego interfejsu dziedziczy klasa Leaf dostarczająca funkcjonalność liścia kompozytu oraz klasa Composite implementująca metody dostępu do zawartych w niej komponentów oraz zawierająca w sobie zero lub więcej komponentów (komponentem może być kolejny kompozyt lub liść). Podczas wykorzystywania wzorca kompozyt, należy mieć na uwadze fakt, iż nie zabezpiecza on przed występowaniem cykli w strukturze.

Proxy (pośrednik)

Wzorzec Proxy jest wykorzystywany do reprezentowania obiektów które są skomplikowane, wymagają dużego nakładu czasu na utworzenie lub takich do których np. chcemy ograniczyć prawa dostępu. Jeśli utworzenie obiektu jest kosztowne, wykorzystanie wzorca Proxy pozwala na odłożenie utworzenia obiektu do momentu w którym będzie on faktycznie wykorzystany. W przypadku potrzeby ograniczenia dostępu do obiektu, Proxy może sprawdzać uprawnienia użytkownika przed wywołaniem żądanej metody na prawdziwym obiekcie.

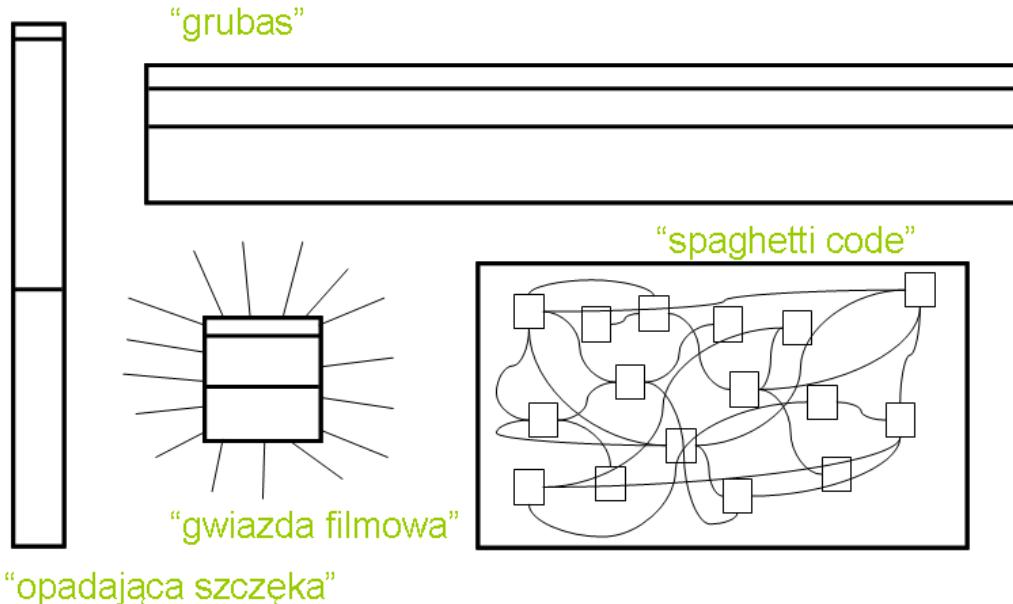


Rysunek 8-31 Wzorzec Proxy

Proxy posiada takie same metody jak obiekt który reprezentuje. W momencie żądania wywołania metody rzeczywistego obiektu, Proxy odwołuje się do instancji prawdziwego obiektu którą posiada. Wywołanie to może być wewnątrz metody Proxy opakowane np. w kod sprawdzający poziom uprawnienia dostępu do wywołania żądanej metody.

Anty-wzorce

Wzorce projektowe opisują rozwiązanie dobrze zidentyfikowanych problemów pojawiających się w trakcie projektowania systemów. Tym samym stanowią pewne dobre praktyki do których powinniśmy się stosować. Analogicznie zidentyfikowano wiele złych praktyk, tzw. anty-wzorców, będących często powtarzanymi w projektach informatycznych błędami. Katalog zidentyfikowanych antywzorów rozciąga się od anty-wzorców w obszarze projektowania architektonicznego i szczegółowego, poprzez anty-wzorce w programowaniu po anty-wzorce organizacyjne i metodologiczne. Przegląd istniejących antywzorów znacznie wykracza poza zakres tego podręcznika, jednakże postaramy się przybliżyć kilka przykładowych złych praktyk, które dość łatwo zidentyfikować i ich uniknąć



Rysunek 8-32 Przykład anty-wzorców łatwych do zidentyfikowania „gołym okiem”

Część złych praktyk projektowych jest wręcz widoczna po pierwszym rzucie okiem na projekt systemu bądź diagram klas odtworzony z kodu poprzez inżynierię odwrotną (reverse engineering –

stworzenie diagramu klas poprzez zaimportowanie istniejącego kodu do narzędzia CASE). Przykład takich anty-wzorców przedstawia Rysunek 8-32. Najczęściej spotykanym błędem jest tzw. „Spaghetti code”. Jest to sytuacja, w której klasy powiązane są między sobą nadmierną ilością relacji. Powoduje to, iż już samo czytanie i rozumienie diagramu klas takiego fragmentu jest prawie niemożliwe, nie mówiąc już o zrozumieniu tych zależności na podstawie kodu źródłowego. Kolejnym przykładem podobnego błędu jest „gwiazda filmowa” – klasa posiadająca tak dużo zależności, że aż od nich „promieniuję” na diagramie. Często popełnianym błędem w fazie projektowania systemu jest umieszczenie zbyt dużej lub zbyt małej odpowiedzialności w jednej klasie. Powoduje to powstawanie klas posiadających jedynie 1-2 metody lub klas, którym na diagramie „opada szczęka” od ilości metod, które zawiera. Innym błędem projektowym jest projektowanie sygnatur metod ze zbyt dużą ilością parametrów. Taka „gruba” klasa wręcz nie mieści się na diagramie.

Dobrą praktyką stosowaną zarówno na etapie projektowania architektonicznego/szczegółowego, jak i na etapie specyfikacji wymagań jest stosowanie się do zasad 7+-/2. Liczba ta została zidentyfikowana przez amerykańskiego psychologa George'a A. Millera w jednym z najczęściej cytowanych artykułów z zakresu psychologii: „The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information”. Oznacza ona liczbę informacji, jaką człowiek może efektywnie przetworzyć na raz. Wynika to z ograniczeń naszej pamięci krótkotrwałej. Stosowanie „magicznej liczby” w projektowaniu oprogramowania pozwala na tworzenie czytelnych i łatwych do rozumienia projektów oraz uniknięcie najczęstszych anty-wzorców projektowych. Jeżeli np. w naszym projekcie istnieją klasy posiadające dziesiątki metod, to nie tylko oznacza to, że klasa taka posiada zbyt dużą odpowiedzialność, ale również znacznie utrudnia zrozumienie takiego projektu. Dlatego w takich wpadkach warto rozdzielić funkcjonalność takiej klasy na kilka mniejszych o rozsądnej liczbie metod. Podobnie jeżeli moduł (komponent) zawiera zbyt dużo składowych (klas implementujących), być może oznacza to, iż powinien on być podzielony na dwa lub więcej mniejszych modułów o logicznie określonym zakresie odpowiedzialności. Problem ten można rozciągnąć na ilość atrybutów w klasach, poziom złożoności relacji pomiędzy klasami lub komponentami (patrz „spaghetti code” i „gwiazda filmowa”) lub na logiczną strukturę projektu.

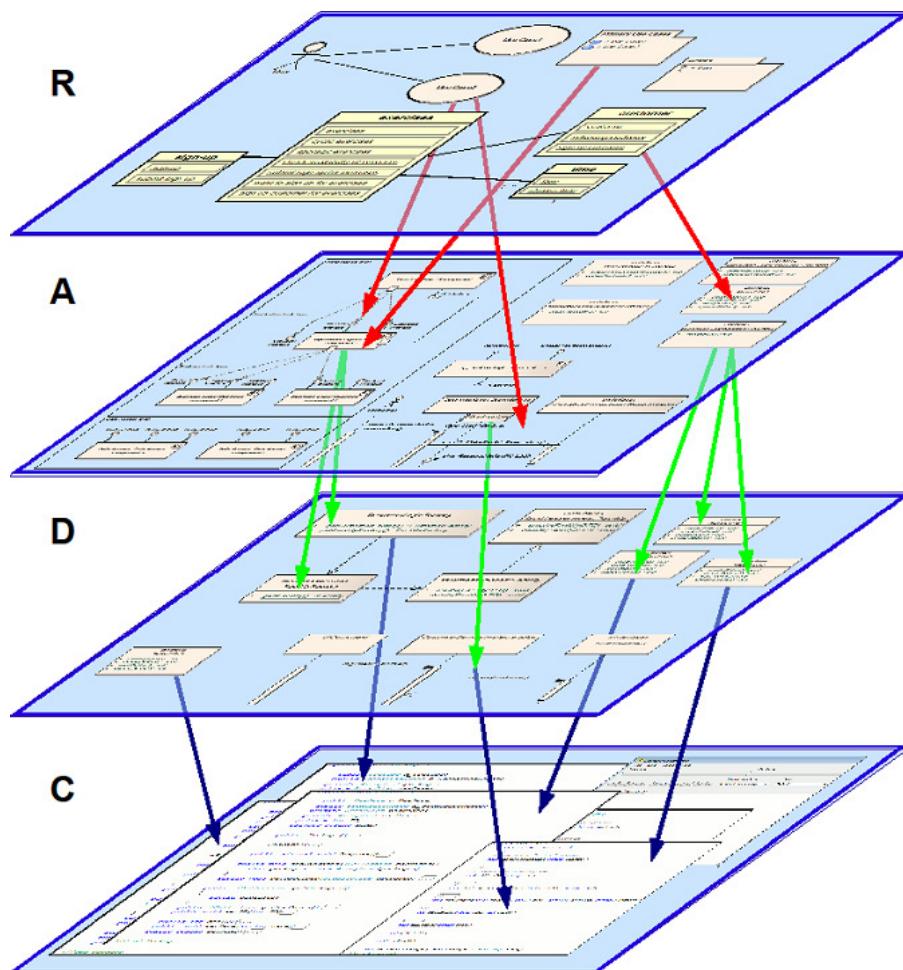
Podobnie jak znajomość wzorców projektowych ułatwia wytwarzanie oprogramowania i powoduje, że tworzone oprogramowanie jest lepszej jakości. Również zapoznanie się z najczęściej spotykanymi złymi praktykami powinno ten proces usprawnić oraz pozwoli uniknąć wielu niepotrzebnych błędów i zaoszczędzić wiele czasu.

8.6. Zachowanie spójności specyfikacji projektowej i zgodności z wymaganiami.

Zapewnienie spójności i zgodności specyfikacji projektowej (zarówno projektu architektonicznego jak i szczegółowego) wymaga zapewnienia możliwości dotarcia do elementów modeli wyższych poziomów z których dane elementy modelu pochodzą oraz modeli niższych poziomów które te elementy realizują.

Rysunek 8-33 przedstawia zależności pomiędzy modelami na różnych poziomach abstrakcji. Dla wymagań zdefiniowanych w specyfikacji wymagań, istnieją fragmenty modelu architektonicznego odpowiedzialne za realizację tych wymagań. Np. dla przypadku użycia będzie to interfejs w warstwie logiki aplikacji odpowiedzialny za realizację danego przypadku oraz elementy warstwy prezentacji wykorzystywane do komunikacji z użytkownikiem podczas wykonania przypadku. Z elementami tymi będą również powiązane komponenty i ich interfejsy w warstwie logiki biznesowej i warstwie danych biorące udział w wykonaniu danego przypadku użycia. Możliwość odnalezienia tych śladów pozwala na śledzenie zależności od wymagań do architektury oraz analizę zmian poszczególnych fragmentów architektury na wypadek zmiany w poszczególnych wymaganiach.

Analogicznie, projekt architektoniczny ma swoje zależności w projekcie szczegółowym, tj. fragmenty projektu szczegółowego przedstawiające realizację danego elementu modelu architektonicznego. Dla komponentu będą to np. klasy zawarte wewnątrz tego komponentu, realizujące zadaną funkcjonalność komponentu. Klasy zawarte w projekcie szczegółowym mapują się z kolei na konkretne jednostki kodu odpowiedzialne za realizację danej klasy. Podobne zależności istnieją również dla opisy dynamiki systemu.



Rysunek 8-33 Zależności pomiędzy modelami wymagań, projektowymi i kodem

Poszczególnym scenariuszom przypadków użycia w specyfikacji wymagań odpowiadają konkretne diagramy sekwencji na poziomie architektury przedstawiające sekwencje wymiany komunikatów niezbędną do realizacji danego scenariusza. Wywołania poszczególnych metod interfejsów w tych diagramach sekwencji będą uszczegółowiana na poziomie projektu szczegółowego przez konkretne diagramy sekwencji, a te z kolei znajdą swoje odbicie w konkretnych metodach zaimplementowanych w jednostkach kodu źródłowego.

Zapewnienie spójności specyfikacji projektowej wymaga zapewnienia synchronizacji poszczególnych elementów na różnych poziomach abstrakcji na wypadek zmian. Synchronizację taką pomiędzy modelem szczegółowym a kodem źródłowym (do pewnego stopnia) zapewniają narzędzia CASE oraz mechanizm inżynierii odwrotnej.

Problem stanowi synchronizacja modeli która musi być dokonywana ręcznie. Aby ułatwić ten proces, powinniśmy zadbać o to aby każde wymaganie w specyfikacji wymagań posiadało unikalny identyfikator pozwalający na śledzenie tego wymagania oraz jego zależności. Ważne jest też dobrze udokumentowanie projektów architektonicznego i szczegółowego pozwalające na łatwe zidentyfikowanie elementów źródłowych oraz wynikających z danych modeli.

Kolejnym problemem jest zapewnienie zgodności z wymaganiami. Aby nasz system spełniał zakładaną funkcjonalność, niezbędny jest mechanizm (w miarę możliwości zautomatyzowany) pozwalający na weryfikację działania systemu względem wymagań. Takim mechanizmem są testy akceptacyjne, o których więcej dowiemy się w rozdziale 11. Testy takie budowane są bezpośrednio na bazie wymagań (scenariuszy przypadków użycia) oraz weryfikują czy system spełnia funkcjonalność opisaną przez te scenariusze.

8.7. Podsumowanie

Celem projektowania architektonicznego i szczegółowego jest zapanowanie nad złożonością budowanego systemu oprogramowania. Dekompozycja systemu na mniejsze jednostki logiczne (komponenty) pozwala projektantom skupić się na pracy nad fragmentem systemu o rozsądnej wielkości.

Projekt architektoniczny i szczegółowy stanowią dokumentację podjętych decyzji i sposobu realizacji systemu. Z tego powodu przejrzysty projekt pełni istotną rolę nie tylko przez cały cykl wytwórczy projektu, ale również w procesie pielęgnacji systemu oprogramowania podczas jego użytkowania.

9. Metamodelowanie MOF i transformacje

9.1. Metamodelowanie

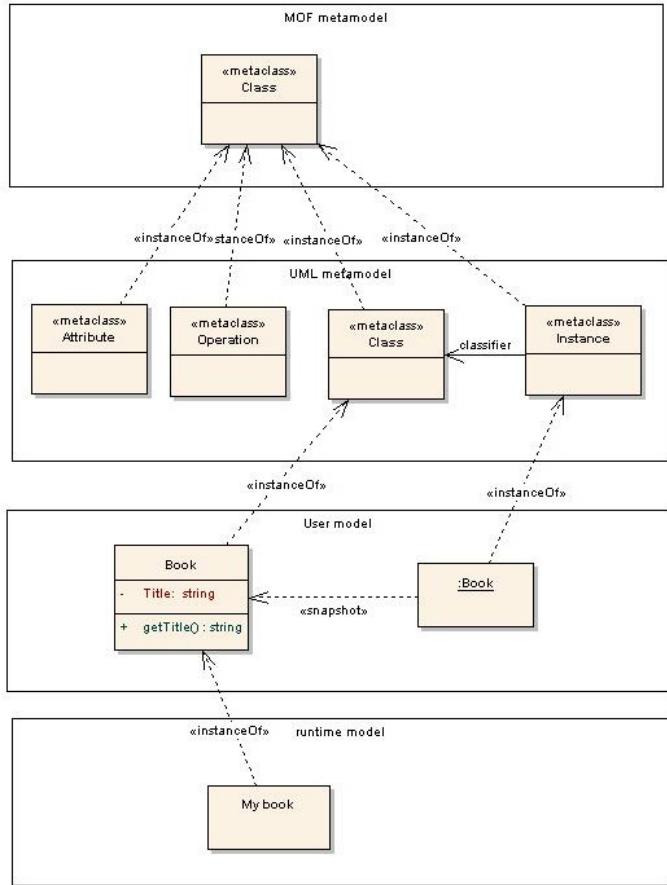
Metamodelowanie jest procesem podobnym do modelowania - różnica polega na przedmiocie modelowania. Podczas gdy modelowanie pozwala na abstrakcyjną reprezentację rzeczywistych procesów i zjawisk, metamodelowanie reprezentuje proces tworzenia innych modeli. Metamodel stanowi definicję języka modelowania wyrażoną w sposób graficzny (tzw. składnia abstrakcyjna) oraz uzupełnioną opisem słownym lub innymi technikami (semantyka, definicja składni konkretnej, ograniczenia narzucone na model wyrażone w języku OCL).

Model jest instancją pewnego metamodelu i sam w sposób rekursywny może stanowić metamodel dla innego modelu. Poprzez rekursywne stosowanie tego schematu możemy otrzymać nieskończoną liczbę warstw.

9.1.1. Czterowarstwowa hierarchia metamodeli

To, co w jednym kontekście jest modelem w innym staje się metamodelem. Tak jest również z językiem UML. UML jest specyfikacją języka (metamodelu), z którego użytkownicy mogą definiować własne modele. Podobnie MOF (Meta Object Facility – język, za pomocą którego zdefiniowany jest metamodel UMLa) jest metamodelu, z którego użytkownicy mogą definiować własne modele. Z perspektywy języka MOF, UML jest specyfikacją użytkownika (modelem), opartym o MOF, jako specyfikację języka. Możemy wyróżnić trzy warstwy definiujące języki modelowania: specyfikacja języka (metamodel), specyfikacja użytkownika (model), obiekty w modelu.

OMG zaproponowało czterowarstwową hierarchię metamodeli. Warstwa meta-metamodelu stanoi fundament hierarchii metamodelownia. Definiuje ona język służący specyfikacji modeli. Meta-metamodel powinien być bardziej spójny od metamodeli, które są za jego pomocą definiowane. MOF jest przykładem meta-metamodelu leżącego u podstaw języka UML.



Rysunek 9-1 Czterowarstwowa hierarchia metamodeli zaproponowana przez OMG

Rysunek 9-1 przedstawia tę hierarchię. Metamodel jest instancją meta-metamodelu. OMG dostarcza kilka metamodeli opartych o MOF, np. UML i CWM. UML jest bardziej skomplikowany od swojego meta-metamodelu i stanowi język specyfikacji modeli użytkownika

Model, będący instancją metamodelu definiuje język opisu dziedziny semantycznej problemu. Model pozwala użytkownikowi reprezentować problemy z różnych dziedzin.

Na samym dole hierarchii leżą instancje elementów zdefiniowanych w modelu użytkownika.

Cechą charakterystyczną hierarchii metamodeli jest możliwość projektowania języków refleksyjnych, tzn. takich, które mogą definiować same siebie. MOF jest językiem refleksyjnym, ponieważ zawiera w sobie wszystkie metaklasy potrzebne do zdefiniowania samego siebie. Oznacza to, że nie jest potrzebna warstwa powyżej warstwy, w której zdefiniowany jest MOF.

Ta czterowarstwowa architektura stanowi infrastrukturę dla zdefiniowania meta-metamodeli, metamodeli i modeli oraz dostarcza podstaw do modelowania rozszerzeń języka UML jak i również innych, zupełnie nowych języków modelowania opartych na metamodelach.

9.1.2. Składnia i semantyka języka

Aby stworzyć język modelowania za pomocą metamodelu, musimy zdefiniować jego składnię i semantykę.

Składnia definiuje wszystkie konstrukcje występujące w języku oraz sposób, w jaki te konstrukcje są tworzone w relacji z innymi konstrukcjami języka. W momencie, gdy język ma graficzną reprezentację, konieczne jest zdefiniowanie języka w sposób niezależny od notacji (składnia abstrakcyjna) oraz mapowania notacji na składnię abstrakcyjną (składnia konkretna).

Semantykę języka możemy podzielić na statyczną i dynamiczną. Statyczna semantyka języka definiuje sposób łączenia konstrukcji języka z innymiinstancjami. Osiąga się to poprzez narzucone pewnych warunków i ograniczeń na model powstający w projektowanym języku. Ograniczenia takie mogą być wyrażone za pomocą opisu słownego, lub bardziej precyzyjnie z wykorzystaniem

języka OCL (Object Constraint Language). Semantyka dynamiczna nadaje znaczenie konstrukcjom powstającym zgodnie z ograniczeniami semantyki statycznej. W celu zdefiniowania języka UML (zdefiniowania jego składni i semantyki) posłużyono się podzbiorem samego języka UML (język MOF), językiem OCL oraz językiem naturalnym.

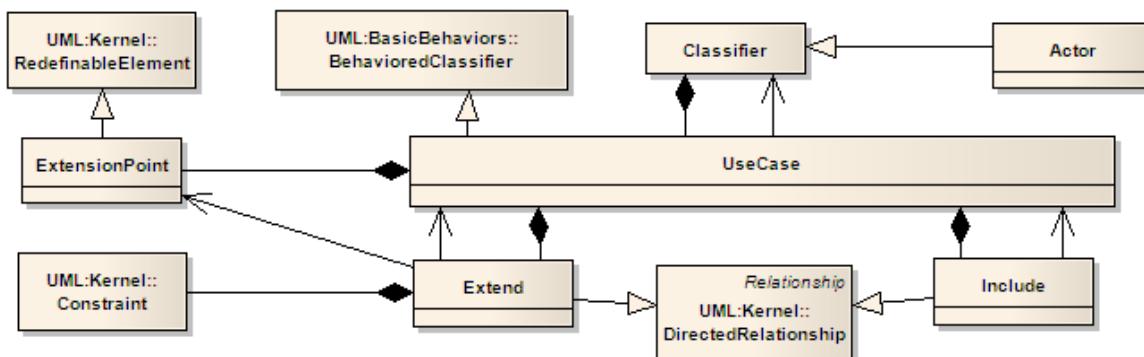
9.1.3. Język MOF

Język MOF, będący podstawą metamodelowania jest w zasadzie podzbiorem diagramu klas. Pozwala on na zdefiniowanie pojęć występujących w języku oraz precyzyjne wyrażenie relacji pomiędzy tymi pojęciami. Podobnie jak w modelu klas języka UML, w MOFie możemy korzystać z relacji takich jak dziedziczenie, asocjacje, agregacja i kompozycja. Metaklasy zaprojektowanie w MOFie mogą również posiadać metaatrybuty. Język MOF został sformalizowany w postaci dwóch odmian: CMOF (Complete MOF) i EMOF (Essential MOF). Pierwszy z nich posiada bogatszą składnię, pozwalającą na wyrażanie modeli łatwiejszych do przyswojenia przez ludzi. Definicja języka w CMOFie jest przydatna na wczesnym etapie projektowania języka, kiedy konstrukcje języka są dopiero „wymyślone” i istotne jest dobre zrozumienie i czytelność metamodeli dla ludzi zaangażowanych w tworzenie języka. Wadą użycia CMOFa dla precyzyjnego zdefiniowania metamodelu jest jego zbyt duża niejednoznaczność. CMOF posiadając bogatą składnię, jest zbyt skomplikowany do precyzyjnego wyrażenia składni abstrakcyjnej koniecznego do zaimplementowania jej w narzędziu wspierającym budowanie modeli w projektowanym języku. Dlatego też twórcy narzędzi wykorzystują do specyfikacji języka na potrzeby implementacji narzędzi bardziej restrykcyjną odmianę MOFa – EMOF. Metamodele wyrażone w EMOFie są trudniejsze do zrozumienia dla ludzi, ale dużo łatwiejsze do zrealizowania w narzędziu.

9.2. Fragment metamodelu na przykładzie UMLa

Aby zobrazować sposób definiowania języka modelowania za pomocą języka MOF skorzystamy z fragmentu specyfikacji języka UML. Rozważanym fragmentem UMLa będzie model przypadków użycia. Jest on zdefiniowany poprzez dość prosty metamodel w porównaniu z innymi fragmentami UMLa.

9.2.1. Metamodel przypadków użycia w języku UML



Rysunek 9-2 Fragment metamodelu przypadków użycia

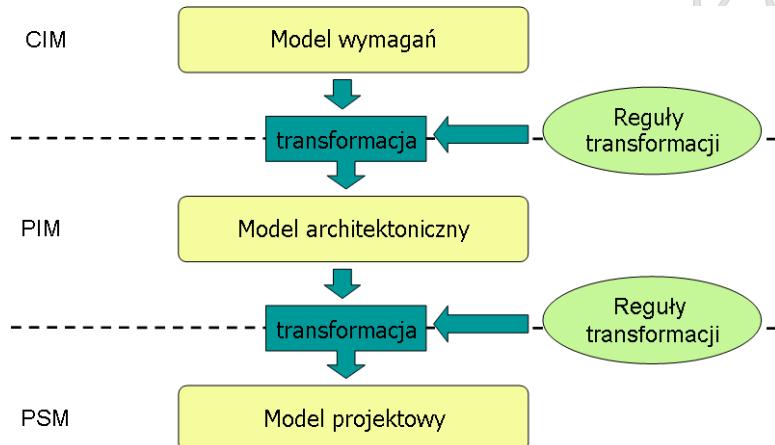
Specyfikacja języka UML definiuje pakiet przypadków użycia jak przedstawiono na rysunku powyżej. Diagram ten stanowi specyfikację składni abstrakcyjnej modelu UML wyrażoną za pomocą języka MOF. Zdefiniowane są tu elementy oraz relacje występujące w modelu przypadków użycia. Metaklasa **UseCase** pochodzi od **BehavioredClassifier**. Aktor, dziedziczący z **Classifiera** rozszerzonego o zdolność posiadania **UseCase-a**, reprezentuje podmiot do którego przypadek użycia się odnosi. Aktor posiada zachowanie zdefiniowane poprzez posiadane przypadki użycia. Przypadki użycia mogą być powiązane dwoma rodzajami relacji: **extend** i **include**. Obie te relacje pochodzą od metaklasy **DirectedRelationship** (relacja skierowana), a więc posiadają kierunek. Dodatkowo relacja **extend** może być dołączona do przypadku użycia w kilku punktach (**extensionPoints**). Przyjrzyjmy się dokładniej sposobowi definicji relacji w tym metamodelu. Jak widzimy, metaklasy **Extend** i **Include** definiujące relacje o takich samych nazwach połączone są z metaklasą **UseCase** przy pomocy agregacji i asocjacji. Metaklasa **UseCase**, będąca w składni konkretnej elipsą, może posiadać relację **Extend** bądź **Include** (agregacje tych metaklas), a te z kolei wskazują asocjacją z powrotem na meta-

klasę UseCase. W składni konkretnej relacje te będą strzałkami z odpowiednim stereotypem. Strzałka reprezentująca relację rozpoczyna się w elipsie reprezentującej przypadek użycia który agreguje tą relację (źródło relacji) a kończy się na elipsie reprezentującej przypadek użycia wskazywany przez asocjację (cel relacji). Jak łatwo się domyślić, składnia abstrakcyjna języka definiuje sposób, w jaki język jest reprezentowany w pamięci komputera wewnętrz narzedzia (np. narzędzia CASE).

Aby specyfikacja języka była kompletna, oprócz składni abstrakcyjnej musi posiadać również składnię konkretną oraz semantykę. Składnię konkretną modelu przypadków użycia poznaliśmy w rozdziale 6. Semantyka modelu przypadków użycia określa, co oznaczają poprawnie sformułowane konstrukcje tego modelu.

9.3. Podstawy transformacji modeli

W rozdziale 3 została wprowadzona koncepcja MDD (Model Driven Development) oraz idea transformacji w procesie wytwarzania oprogramowania. W tym rozdziale dowiemy się więcej na temat sposobu definiowania transformacji oraz narzędzi (języków transformacji), które takie transformacje pozwolą nam zaimplementować i wykonywać.



Rysunek 9-3 Transformacje pomiędzy modelami w kontekście MDA

Rysunek 9-3 przedstawia ogólną ideę transformacji w kontekście MDD. Model MDD definiuje trzy warstwy modelowania oprogramowania. Warstwa CIM (Computation independent model) jest warstwą modeli przedstawiających system w sposób niezależny od jego realizacji w oprogramowaniu. Warstwa ta najczęściej odpowiada modelowi wymagań. Model wymagań definiuje funkcjonalność systemu (co ma robić system) bez określania sposobu osiągnięcia tego celu. Kolejną warstwą jest warstwa PIM (Platform independent model). Modele zdefiniowane w tej warstwie określają, w jaki sposób system realizuje swoją funkcjonalność, ale w sposób niezależny od platformy technologicznej, która posłuży do zaimplementowania tej funkcjonalności. Modelem niezależnym od platformy jest projekt architektoniczny. Ostatnią, najniższą warstwą jest warstwa PSM (Platform specific model), czyli warstwa zawierające modele określające realizację systemu przy wykorzystaniu konkretnej platformy technologicznej. Modelem takim jest projekt szczegółowy systemu.

Zgodnie z duchem idei MDD, przejścia pomiędzy tymi warstwami (sposób tworzenia modeli niższych warstw na podstawie modeli wyższych warstw) powinien być zautomatyzowany poprzez transformację modeli. W celu zrealizowania takich transformacji, musimy najpierw zdefiniować reguły takiej transformacji (reguły określające, w jaki sposób z modeli wyższych warstw generowane są modele niższych warstw). Posiadając ściśle zdefiniowane reguły transformacji możemy przystąpić do implementacji takiej transformacji korzystając z wybranego języka transformacji.

Następny podrozdział charakteryzuje najpopularniejsze dostępne języki transformacji oraz przedstawia bardziej szczegółowo jeden z nich – język MOLA. Kolejny podrozdział omawia transformację pomiędzy modelami na różnych poziomach abstrakcji na przykładzie reguł transformacji modelu wymagań na model statyczny architektury.

9.4. Języki specyfikacji transformacji

W tym rozdziale poznamy kilka przykładowych języków transformacji wraz z ich krótką charakterystyką. Dla lepszego zrozumienia idei transformacji modeli, jeden z tych języków (język MOLA) zostanie przedstawiony bardziej szczegółowo. Na przykładzie prostej transformacji poznamy podstawowe konstrukcje języka MOLA.

9.4.1. Przegląd specyfikacji języków transformacji

Cechą wspólną większości języków transformacji jest podstawowa idea ich działania. Z reguły mamy do czynienia z modelem źródłowym i docelowym. Elementy modelu źródłowego podlegające transformacji, dopasowywane są do pewnego zdefiniowanego wzorca. Te elementy, które zostaną dopasowane do zadanego wzorca podlegają transformacji (na nich wykonywana jest pewna określona akcja).

Języki transformacji możemy skategoryzować według różnych kryteriów. Jednym z kryteriów podziału języków może być sposób definiowania wzorców wykorzystywanych w transformacji. Zgodnie z tym kryterium języki możemy podzielić na języki tekstowe i graficzne. W językach tekstowych wzorzec będzie definiowany za pomocą kodu w języku tekstowym, w językach graficznych wzorzec będzie wyrażany przy pomocy graficznego modelu. Bardziej intuicyjne i czytelniejsze wydaje się drugie podejście. Dla niektórych, bardziej skomplikowanych sytuacji, graficzne zdefiniowanie wzorca może być trudne, dlatego też niektóre języki stanowią hybrydę obu podejść pozwalając na wyrażanie wzorców zarówno w sposób tekstowy, jak i graficzny.

Kolejnym kryterium podziału języków transformacji jest podział na języki deklaratywne i imperatywne. W uogólnieniu, język deklaratywny (nie tylko język transformacji) to taki, który opisuje CO ma być wynikiem działania programu (jakie warunki musi spełniać) nie określając JAK ma to być osiągnięte. Przykładem takich języków jest np. HTML, opisujący jak ma wyglądać strona internetowa, lub język SQL, w którym definiujemy, jaki widok ma być wyselekcjonowany z bazy danych. W językach imperatywnych, w przeciwieństwie do deklaratywnych, programista instruuje maszynę JAK osiągnąć założony cel (definiuje algorytmy prowadzące do osiągnięcia zamierzonego rezultatu). Paradygmat imperatywny spełnia większość popularnych języków programowania takich jak np. Java bądź C. Dla definiowania transformacji bardziej naturalnym jest wyrażenie ich w sposób deklaratywny, dlatego większość języków spełnia ten paradygmat. Istnieją jednak sytuacje, w których transformacja jest niezmiernie trudna lub wręcz niemożliwa do wyrażenia w sposób deklaratywny. Rozwiązaniem tego problemu są języki hybrydowe wspierające obie metody definiowania transformacji.

ATL

Język ATL (ang. Atlas Transformation Language) został stworzony na Uniwersytecie w Nantes przy współpracy z narodowym francuskim instytutem badawczym INRIA (fr. Institut national de recherche en informatique et en automatique). Powstanie języka było odpowiedzią na ogłoszenie przez instytucję OMG zapotrzebowania na specyfikację standardu języka QVT (RFP – Request For Proposal) i miał on znaczący wpływ na ostateczny kształt standardu QVT.

ATL charakteryzuje się hybrydową, tekstowo-graficzną notacją i umożliwia wygenerowanie wielu modeli docelowych na podstawie zbioru modeli wejściowych. ATL wspiera tworzenie transformacji zarówno w sposób imperatywny, jak i deklaratywny. Podejście deklaratywne uznawane jest za podstawowe, a imperatywne za pomocnicze.

Transformacje modeli w języku ATL definiuje się w tak zwanych modułach. Każdy moduł zapisany jest przeważnie w oddzielnym pliku wejściowym, ale nie jest to obowiązkowe. Moduł składa się z czterech części: nagłówka, importów, pomocników (ang. helpers) oraz reguł. Najistotniejsza jest sekcja definiowania reguł.

W języku ATL występują dwa rodzaje reguł: reguły dopasowujące (ang. matched rules) - obsługujące programowanie deklaratywne oraz reguły wołające (ang. called rules) - obsługujące programowanie imperatywne.

Reguła dopasowująca rozpoczyna się od słowa kluczowego rule i składa z dwóch obowiązkowych sekcji, w których następuje definicja wzorca źródłowego (from) i docelowego (to), oraz dwóch opcjonalnych zawierających definicje zmiennych lokalnych (using) oraz elementy programowania imperatywnego (distinct). Poniższy fragment kodu przedstawia prosty przykład reguły dopasowującej:

```
rule Author {
    from
    a : MMAuthor!Author
    to
    p : MMPerson!Person (
        name <- a.name,
        surname <- a.surname
    )
}
```

W momencie deklarowania zmiennej projektant musi określić model oraz meta-model, do którego się ona odnosi.

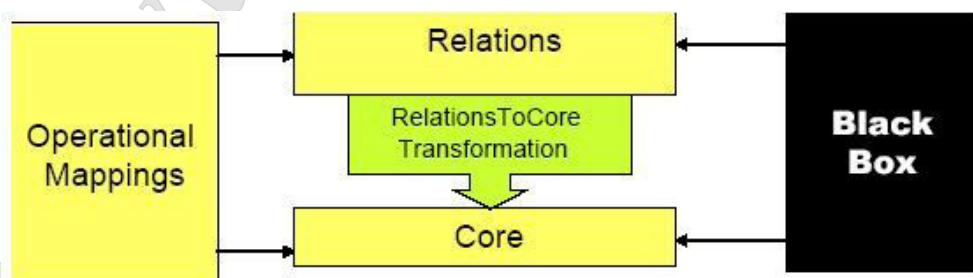
Reguła wołająca również rozpoczyna się od słowa kluczowego rule lecz na tym podobieństwa się kończą. Nie zawiera sekcji źródłowej, a jedynie docelową (to), która jest opcjonalna. Imperatywne polecenia są umieszczone tak jak w regule dopasowującej w sekcji imperatywnej, rozpoczyna się ona od słowa kluczowego do. Reguły wołające mogą przyjmować parametry (tak jak procedury), oraz mogą być wywołane tylko z wewnętrz inną reguły (podczas transformacji jedynie reguły dopasowujące są wywoływane domyślnie). Poniższy fragment kodu przedstawia przykład reguły wołającej.

```
rule NewPerson (param1: String, param2: String) {
    to
    p : MMPerson!Person (
        name <- param1
    )
    do {
        p.surname <- param2
    }
}
```

QVT

QVT (Query/View/Transformation), jest standardem języka transformacji zaproponowanym przez organizację OMG (Object Management Group). Istnieje kilka niezależnych implementacji tego standardu. Jedną z nich jest Eclipse M2M (Model to model).

Język QVT jest językiem hybrydowym imperatywno-deklaratywnym. Część deklaratywnej zbudowana jest w architekturze dwupoziomowej i stanowi podstawę do wykonywania części imperatywnej.



Rysunek 9-4 Architektura języka QVT

Poziomy części deklaratywnej to:

- Warstwa Relations – stanowiąca najważniejszą część standardu jest przyjazną dla użytkownika warstwą umożliwiającą intuicyjne definiowanie relacji zachodzących pomiędzy

modelami w sposób graficzny i tekstowy. Język ten wspiera dopasowywanie złożonych wzorców oraz śledzenie źródeł zmian w modelu docelowym.

- QVT Core stanowiąca szkielet do definiowania transformacji na wyższym poziomie abstrakcji. Warstwa ta dostarcza podobnej funkcjonalności jak oparte na niej QVT Relations, lecz jest bardziej skomplikowana i przeznaczona dla szerokiego grona specjalistów. Składnia języka jest stosunkowo prosta (w porównaniu z QVT Relations), jednak zmusza to użytkownika definiującego transformację do zagłębiania się w szczegóły techniczne takie jak klasy pośrednie. Czyni to definiowanie transformacji bardziej skomplikowanym, a powstałe definicje są rozbudowane i tym samym bardziej trudne do interpretacji przez człowieka. Definicja transformacji przy użyciu języka QVT Core może być stworzona bezpośrednio, lub pośrednio przy użyciu QVT Relations, a następnie zmapowana na język QVT Core..

Poza częścią deklaratywną, która umożliwia deklarowanie transformacji na dwóch różnych poziomach abstrakcji istnieją dwa mechanizmy wywoływanie imperatywnej implementacji transformacji: standardowy język – mapowanie relacyjne (ang. Operational Mappings), a także niestandardowy zrealizowany na zasadzie „czarnej skrzynki” – MOF Operations. Język mapowania relacyjnego został stworzony jako standardowa droga dostarczania imperatywnych implementacji i pokrywa się funkcjonalnością z QVT Relations.

MOLA

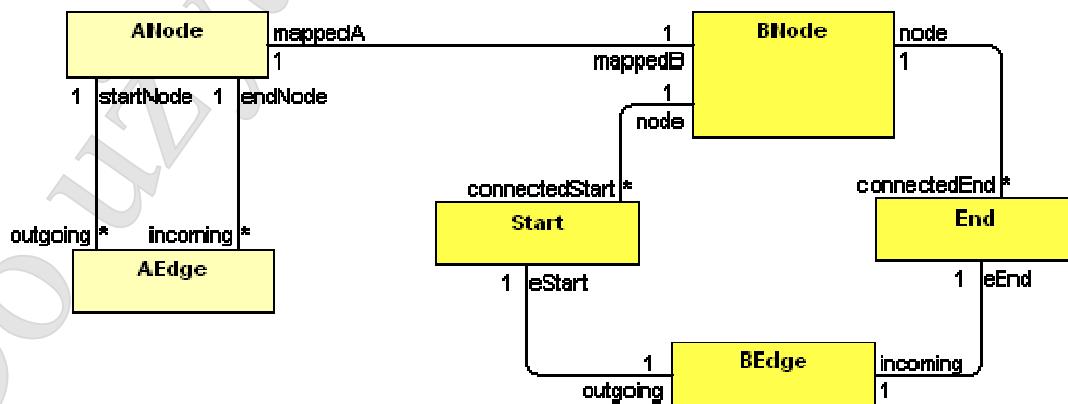
MOLA jest językiem transformacji rozwijanym na Uniwersytecie Łotewskim w Rydze. MOLA to skrót od angielskiego wyrażenia MOdel transformations LAngue, co w dosłownym tłumaczeniu oznacza język transformacji modeli. Głównym celem języka MOLA jest dostarczenie łatwego i czytelnego sposobu zapisu transformacji modeli. Zapis oparty jest na połączeniu tradycyjnych języków strukturalnych z graficzną formą zapisu opartą na regułach i wzorcach.

Programy w języku MOLA używane są do transformacji modelu źródłowego zgodnego ze źródłowym metamodelu na model docelowy zgodny z metamodelem docelowym. Transformacje same w sobie są modelami i tworzone są w sposób graficzny. Transformacje w języku MOLA to zbiór reguł transformacji. Każda taka reguła składa się ze wzorca i akcji. Kiedy wzorzec zostanie odnaleziony (dopasowany) w zbiorze instancji źródłowych, na dopasowanych instancjach zostaje wykonana zdefiniowana akcja. Następny podrozdział przedstawia na przykładzie podstawową składnię języka MOLA.

9.4.2. Podstawy języka MOLA

Język transformacji MOLA transformuje instancje metamodelu źródłowego w instancje metamodelu docelowego. Podstawowymi elementami języka MOLA umożliwiającymi taką transformację są reguły, wzorce i akcje.

Aby zrozumieć podstawowe konstrukcje języka MOLA posłużymy się prostym przykładem transformacji pomiędzy dwoma sposobami zapisu grafu skierowanego.



Rysunek 9-5 Przykład metamodelu źródłowego i docelowego oraz mapowania

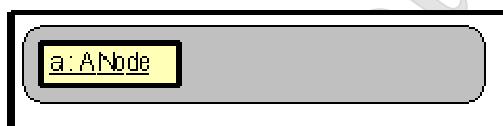
Pierwszy zapis (metamodel po lewej stronie rysunku powyżej) zawiera węzły (Nodes) i krawędzie. Każda krawędź posiada dokładnie jeden węzeł początkowy i końcowy. Drugi sposób zapisu (prawa strona rysunku) zawiera dodatkowe punkty połączenia typu Start i End. Każda krawędź posiada swój początek i koniec (punkty połączenia Start i End). Tylko te punkty są połączone bezpośrednio z węzłami. W dalszej części tego rozdziału poznamy podstawowe konstrukcje języka MOLA poprzez zdefiniowanie transformacji zamieniającej jeden sposób zapisu grafu na drugi.

Pierwszym krokiem w zdefiniowaniu transformacji w języku MOLA jest wprowadzenie metamodeli obu języków (źródłowego i docelowego) do narzędzia MOLA. Rysunek powyżej jest przykładem takich metamodeli wprowadzonych do narzędzia MOLA. Dodatkowa asocjacja łącząca oba metamodely jest mapowaniem obu metamodeli i jest pierwszym etapem definiowania transformacji MOLA (mapowanie to będzie później wykorzystane w procesie transformacji).



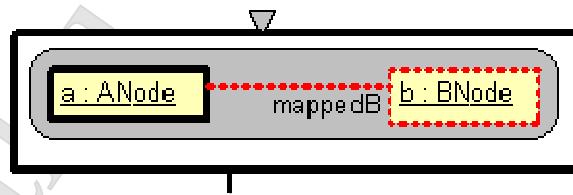
Rysunek 9-6 Wzorzec w języku MOLA

Najprostszą konstrukcją języka MOLA jest wzorzec (Rysunek 9-6). Składa się on z klasy (w przykładzie powyżej klasa ANode z modelu źródłowego) oraz nazwy zmiennej. Fragment transformacji powyżej odnajdzie po wszystkie instancje żądanej klasy (ANode) po kolei przypisując je do zmiennej a.



Rysunek 9-7 Pętla w języku MOLA

Przypasowanie pokazane w poprzednim przykładzie jest niedeterministyczne (nie wiemy kiedy i w jakiej kolejności silnik języka MOLA przypasuje elementu modelu do zmiennej a). Częściej spotkaną konstrukcją będzie wykorzystanie pętli. Pętla oznaczana jest grubą czarną obwódką wokół reguły. Wzorzec okolony podobną obwódką wewnętrz reguły jest zmienną pętli. Wykonanie takiego fragmentu transformacji spowoduje wywołanie wnętrza pętli tyle razy ile elementów żądanego typu znajduje się w modelu źródłowym, za każdym razem przypisując przypasowany element do zmiennej a.



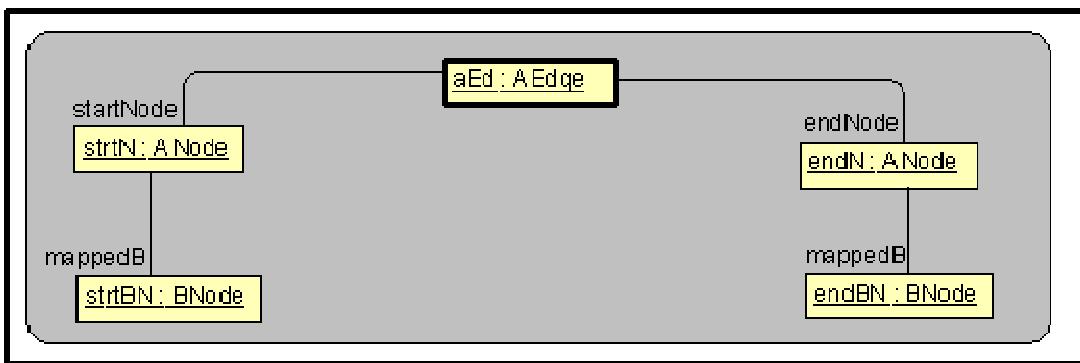
Rysunek 9-8 Reguła z akcją

Reguła w MOLA zwykle nie tylko przypasowuje się do zadanego wzorca, ale również wykonuje jakąś akcję, np. tworzy instancję klasy bądź relację, usuwa klasę bądź relacje albo modyfikuje wartości atrybutów.

W MOLA pętla zawiera jedną główną regułę zawierającą zmienną pętli oraz może zawierać dodatkowe reguły, które również są wykonywane w każdej iteracji. Rysunek 9-8 przedstawia pętle zawierającą tylko jedną (główną) regułę a w niej jedną akcję. Za każdym razem, kiedy reguła dopasuje instancję żądanej klasy (albo pętla wykona iteracje dla kolejnego znalezionej elementu) akcja jest wykonywana. W przykładzie powyżej, każda iteracja pętli stworzy w modelu docelowym instancję klasy BNode oraz relację pomiędzy elementem źródłowym i docelowym odpowiadającą asocjacji mapującej jeden metamodel na drugi.

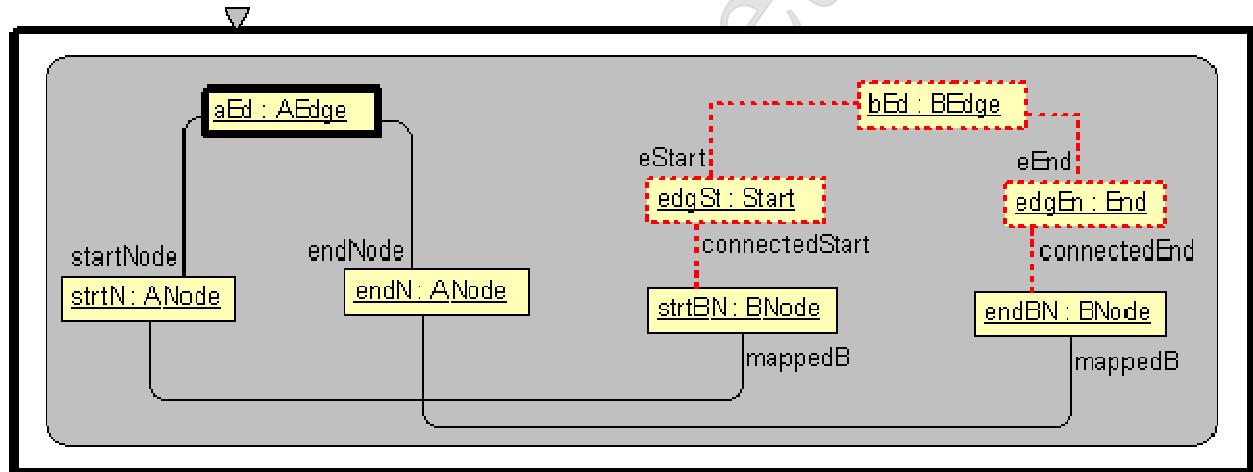
Tworzenie instancji w MOLA zaznaczane jest poprzez umieszczenie wzorca klasy w czerwonej przerywanej obwódkie. Utworzenie relacji dokonywane jest poprzez umieszczenie czerwonej prze-

rywaną linie pomiędzy elementami, które ma łączyć relacja. Typ relacji określany jest poprzez umieszczenie nazwy asocjacji przy końcu (przynajmniej jednym) linii tworzącej relacje.



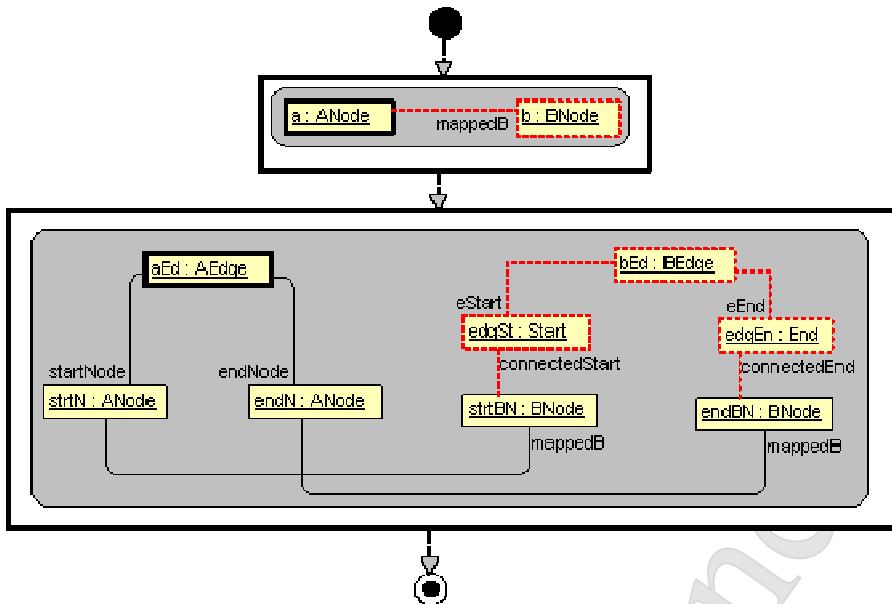
Rysunek 9-9 Przykład bardziej złożonego wzorca

Kolejny przykład (Rysunek 9-9) przedstawia pętle z bardziej skomplikowanym wzorcem. W tym wzorcu, zmienna pętli (`aEd:AEdge`) iteruje po wszystkich instancjach klasy `AEdge` w modelu źródłowym (wszystkich krawędziach grafu zapisanego w pierwszy typie notacji). Jednak wzorzec musi się dopasować również do pozostałych czterech klas zawartych we wzorcu. Dlatego też iteracja całej pętli wykona się tylko dla tych krawędzi `AEdge`, dla których cały wzorzec pasuje – muszą dla takiej krawędzi istnieć dwie instancje węzła (`ANode`) połączone za pomocą relacjami odpowiednio `startNode` i `endNode`, oraz instancja klasy `BNode` odpowiadająca danemu węźlowi `ANode`.



Rysunek 9-10 Złożony wzorzec wraz z akcją

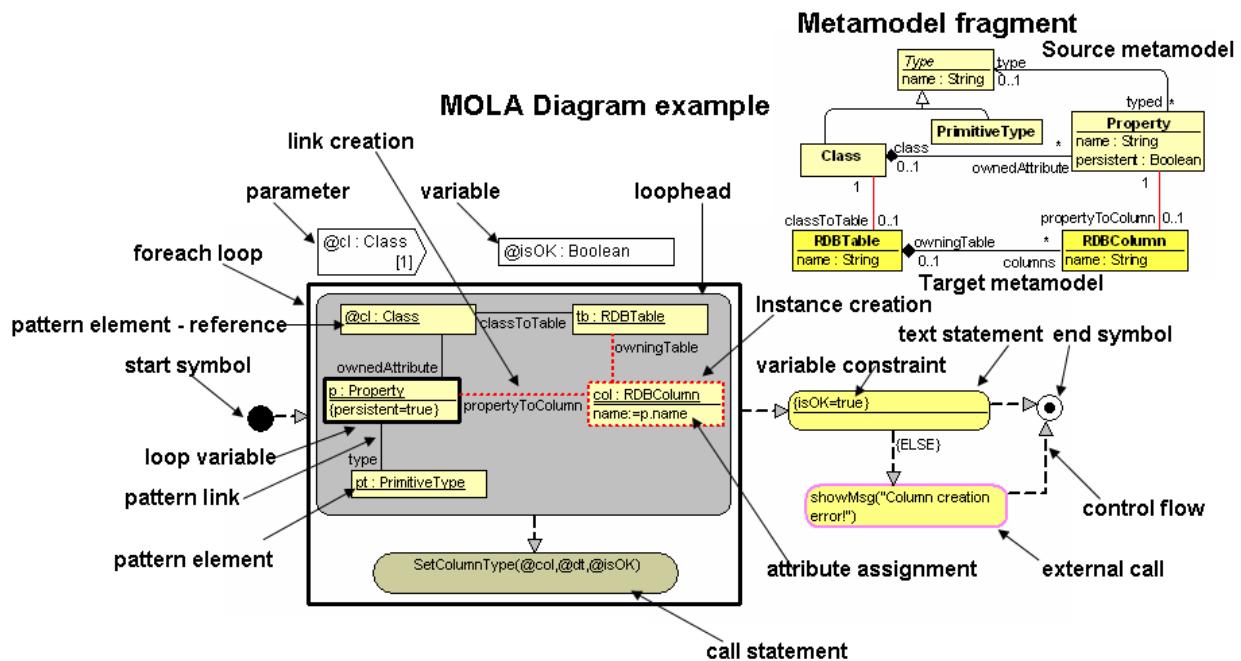
Rysunek powyżej przedstawia kompletną regułę uzupełnioną o akcję pętli. Akcja wewnętrz pętli przy każdej iteracji (dla każdego dopasowanego elementu spełniającego wzorzec) tworzy trzy instancje i cztery relacje – dla każdej krawędzi typu `ANode` w modelu źródłowym, w modelu docelowym tworzona jest instancja krawędzi typu `BNode` wraz zinstancjami punktów łączących `Start` i `End` oraz relacjami łączącymi krawędź z punktami połączenia, a te z węzłami `BNode` odpowiadającymi węzłem `ANode` z modelu źródłowego. W przykładzie tym wykorzystane są węzły typu `BNode` utworzone wcześniej przez pierwszą zaprezentowaną pętlę.



Rysunek 9-11 Kompletny program MOLA

Kompletny program MOLA transformujący graf skierowany zapisany przy pomocy metamodelu źródłowego w graf zapisany przy pomocy metamodelu wynikowego przedstawia Rysunek 9-11.

Elementami dodanymi względem poprzedniego fragmentu są węzły początkowy i końcowy (podobne do tych znanych nam z modelu aktywności) i przepływ sterowanie (strzałki). W najprostszych przypadkach, przepływ sterowania przechodzi po kolej od jednego elementu programu mola (pętli) do następnego. W MOLA można zdefiniować również bardziej złożone reguły posiadające dwa wyjścia. Kolejność wykonania programu jest następująca: najpierw wykonywana jest w całości pierwsza pętla (dla każdego węzła typu ANode tworzony jest odpowiadający mu węzeł typu BNode), dopiero po jej ukończeniu wykonywana jest pętla druga (która może korzystać i korzysta z wyniku działania pętli pierwszej).



Rysunek 9-12 Konstrukcje języka MOLA (na podstawie podręcznika języka)

Konstrukcje użyte w powyższym przykładzie nie wyczerpują wszystkich konstrukcji dostępnych w języku MOLA, jednakże pozwalają na zorientowanie się w ogólnej zasadzie tworzenia transformacji z wykorzystaniem tego języka jak i również rozpoczęcie samodzielnego eksperymentów z tym językiem. Bardziej kompletny zestaw konstrukcji języka przedstawia Rysunek 9-12.

9.5. Transformacja między modelami na różnych poziomach abstrakcji.

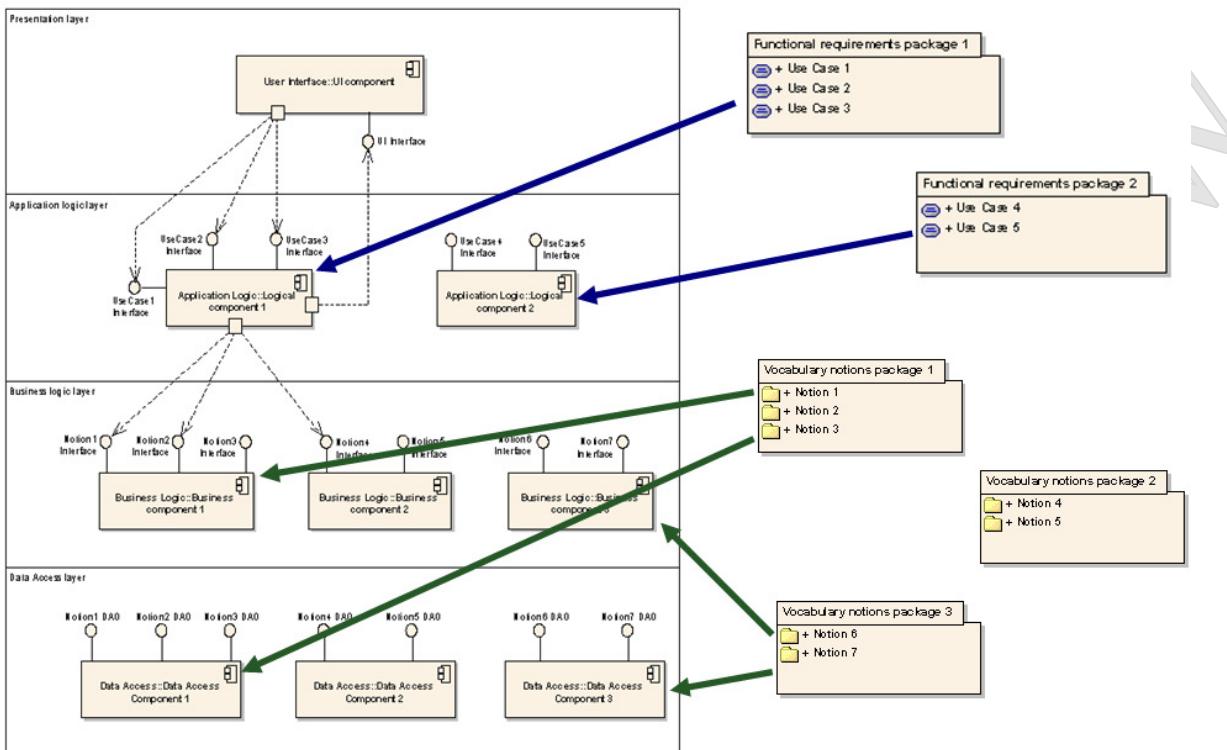
Transformacje umożliwiają zautomatyzowania przejścia pomiędzy modelami na różnych poziomach abstrakcji, wszędzie tam gdzie nie jest wymagane podejmowanie decyzji przez architekta/projektanta, lub decyzje takie mogą być zaimplementowane wewnętrz takiej transformacji. Jako przykład dobrego „materiału” na transformację weźmy definiowanie modelu statycznego architektury (stworzenie modelu komponentowego oraz wyodrębnienie interfejsów za pomocą których te komponenty się komunikują).

Aby zrealizować i z sukcesem korzystać z takiej transformacji, musimy poczynić pewne założenia wstępne oraz zdefiniować (na razie w sposób nieformalny) reguły transformacji. Reguły te określą, z jakich elementów modelu wymagań generowane są jakie elementu modelu statycznego architektury.

Nasze założenia wstępne będą następujące:

- Modelem wejściowym transformacji jest dobrze zdefiniowany model wymagań funkcjonalnych w postaci kompletnego modelu przypadków użycia podzielonych na pakiety stanowiące logiczne jednostki funkcjonalności. Model przypadków użycia uzupełniony jest o kompletny model słownika pojęć dziedzinowych zawierający wszystkie pojęcia występujące w modelu wymagań funkcjonalnych. Pojęcia w modelu słownika również muszą być podzielone na pakiety zawierające w sobie zestaw logicznie powiązanych pojęć. Aby słownik mógł być wykorzystany w procesie transformacji musi być zdefiniowany w postaci modelu (niech będzie to model klas).
- Docelową architekturą dla której chcemy wygenerować model statyczny jest architektura czterowarstwowa, którą poznaliśmy w rozdziale 8.
- Projektowana transformacja nie uwzględnia generowania komponentów warstwy prezentacji, gdyż model wymagań sformułowany w taki sposób nie dostarcza wystarczającej informacji do zautomatyzowania tego procesu. Zamiast tego wygenerujemy pojedynczy komponent warstwy prezentacji, dostarczający pojedynczego interfejsu, z którym będą komunikować się wszystkie komponenty warstwy logiki aplikacji.
- Transformacja nie uwzględnia również wymagań pozafunkcjonalnych zdefiniowanych w modelu wymagań, oprócz tych, które z założenia wbudowane są w transformację (np. decyzja o zaprojektowaniu architektury komponentowej czterowarstwowej).

Po określeniu wstępnych założeń dla projektowanej transformacji, czas na określenie reguł określających, w jaki sposób konkretne elementy modelu architektonicznego będą generowane z elementów modelu wymagań.



Rysunek 9-13 Transformacja modelu wymagań do statycznego modelu architektury

Zadaniem transformacji jest wygenerowanie komponentów logiki aplikacji, logiki biznesowej, warstwy danych oraz ich interfejsów i oraz modelu danych wymienianych pomiędzy poszczególnymi komponentami (tzw. DTO – Data Transfer Objects). Ogólną koncepcję takiej transformacji przedstawia Rysunek 9-13.

Jak wiemy z poprzedniego rozdziału, logika aplikacji jest odpowiedzialna za realizację poszczególnych przypadków użycia. Przy założeniu, że przypadki użycia są pogrupowane w pakiety zawierające logicznie powiązane ze sobą jednostki funkcjonalności, naturalnym będzie generowanie komponentów odpowiadających pakietom przypadków użycia, odpowiedzialnych za realizację przypadków użycia zawartych w tych pakietach. Aby uporządkować komunikację komponentu odpowiadającego pakietowi przypadków użycia, zamiast wystawiania pojedynczego interfejsu odpowiedzialnego za dostarczenie usług związanych ze wszystkimi przypadkami, wygenerujemy po jednym interfejsie dla każdego przypadku zawartego w rozważanym pakiecie.

Logika biznesowa w architekturze czterowarstwowej odpowiedzialna jest za przetwarzanie danych zdefiniowanych w słowniku dziedziny. Dlatego też struktura tej warstwy powinna odzwierciedlać logiczną strukturę słownika. Z pakietów pojęć słownikowych zostaną wygenerowane komponenty warstwy biznesowej. Dla każdego pojęcia zawartego w takim pakiecie zostanie w odpowiednim komponencie wygenerowany interfejs, który będzie grupował metody biznesowe związane z tym pojęciem.

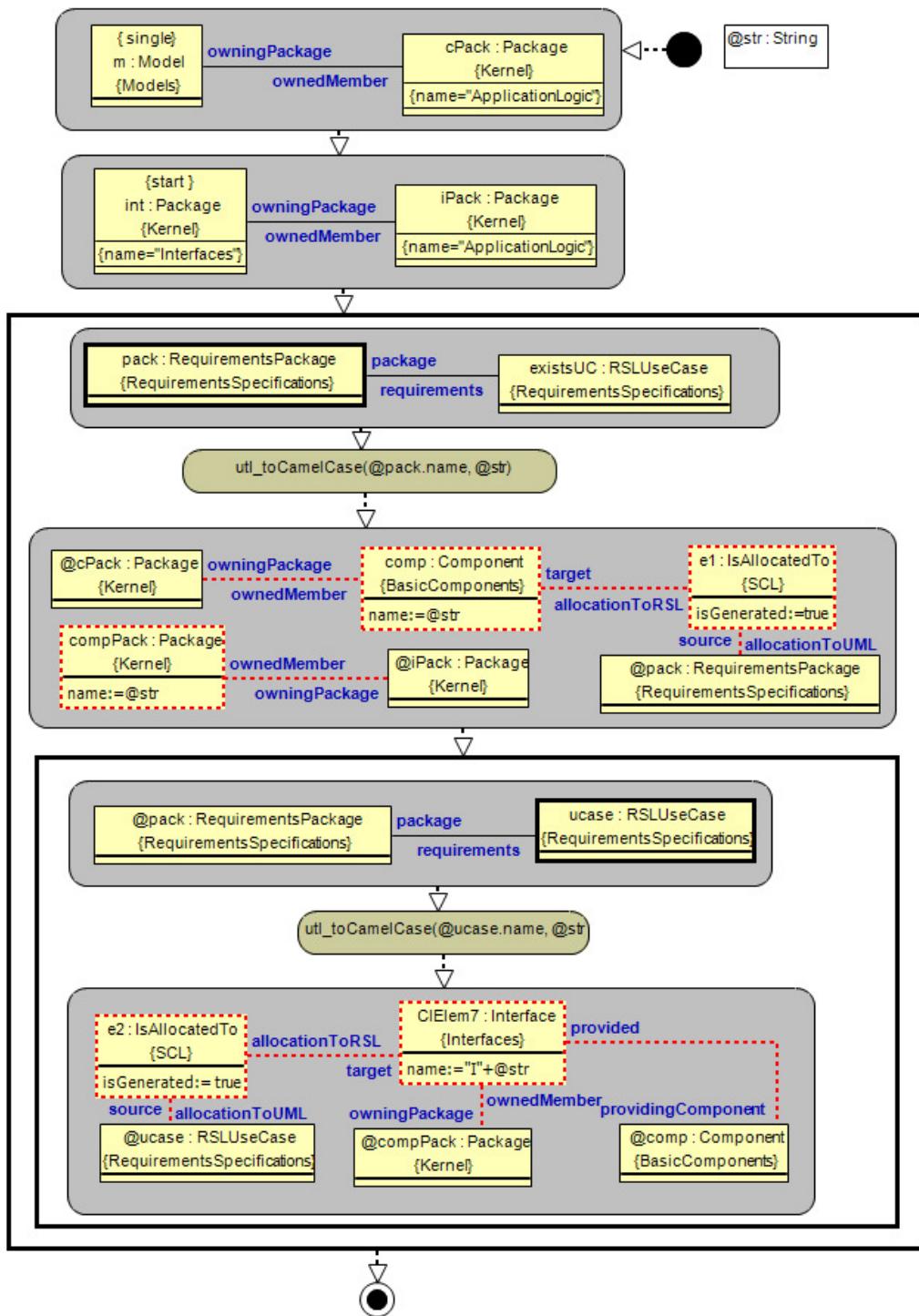
Warstwa danych odpowiedzialna jest za przechowywanie i udostępnianie danych przetwarzanych przez warstwę biznesową, dlatego struktura tej warstwy może być wygenerowana w sposób analogiczny do warstwy biznesowej. Z każdego pakietu słownika pojęć zostanie wygenerowany jeden komponent warstwy danych odpowiedzialny za przechowywanie i udostępnianie pojęć zawartych w tym pakiecie. Dla każdego pojęcia zostanie w odpowiednim komponencie wygenerowany interfejs grupujący metody służące do manipulowania danym pojęciem. Możemy od razu założyć, że w interfejsach tych wygenerujemy dla pojęć metody CRUD (Create-Read-Update-Delete – stwórz, czytaj, aktualizuj, usuń). Dodatkowo, jeśli uważamy to za stosowne, możemy również generować domyślnie metodę zwracającą wszystkie obiekty danego typu.

Kolejnym etapem będzie zdefiniowanie generowania modelu danych. Tutaj posłużymy się wprost słownikiem dziedziny. Dla każdego pojęcia słownika wygenerujemy klasę DTO. Jeśli w słowniku analityk określił atrybuty bądź relacje pomiędzy pojęciami i ich krotności to te atrybuty i relacje wygenerujemy również w modelu klas transferu danych.

Jak widać zaimplementowanie transformacji zgodnej z powyższymi regułami, powinno oszczędzić dużo żmudnej pracy architektowi. Uzupełnienie takiego modelu będzie wymagało zdefiniowania metod poszczególnych interfejsów na podstawie treści scenariuszy przypadków użycia. Informacja zawarta w scenariuszach przypadków użycia nie może być wykorzystana w procesie transformacji, gdyż scenariusze z reguły wyrażone są poprzez tekst. Posłużenie się informacją zawartą w scenariuszach wymagałoby sformułowania ich w sposób bardziej formalny w postaci modelu. Takiej możliwości nie daje nam UML, ale możliwe jest zaprojektowanie języka modelowania wymagań specyfikującego scenariusze w postaci precyzyjnie określonych modeli. Wykorzystanie takiego języka pozwoliłoby na jeszcze dalsze uszczegółowienie wygenerowanego modelu architektonicznego (np. poprzez degenerowanie metod interfejsów lub wygenerowanie diagramów sekwencji na podstawie sekwencji zdań scenariuszy). Kolejnym aspektem nieuwzględnionym w powyższej transformacji jest warstwa prezentacji, którą architekt musi zaprojektować samodzielnie. Jednakże nawet bez wykorzystania wiedzy zawartej w scenariuszach oraz uwzględnienia warstwy prezentacji, zdefiniowana powyżej transformacja pozwala na automatyczne wygenerowanie szkieletu architektury logicznej będącej dobrym punktem startowym do dalszej pracy.

Aby transformację przedstawioną powyżej móc wykorzystać w praktyce, należy ją zaimplementować w wybranym języku transformacji. Fragment implementacji powyższej transformacji w języku MOLA przedstawia Rysunek 9-14. Jest to fragment odpowiedzialny za wygenerowanie komponentów warstwy logiki aplikacji na podstawie pakietów przypadków użycia oraz ich interfejsów na podstawie samych przypadków użycia.

Poniższy diagram stanowi jedynie mały fragment całej transformacji. Implementacja kompletnej transformacji wymaga sporego doświadczenia w posługiwaniu się językiem transformacji i dużego nakładu pracy. Jednak czas zaoszczędzony przez architekta w perspektywie wielu projektów, w których taka transformacja może być wykorzystana powinien ten wysiłek wynagrodzić.



Rysunek 9-14 Fragment transformacji zaimplementowanej w MOLA – generowanie komponentów logiki aplikacji i ich interfejsów na podstawie struktury modelu przypadków użycia

9.6. Podsumowanie

Metamodelowanie i transformacje modeli są nowoczesnymi narzędziami inżynierii oprogramowania dopełniającymi techniki poznane w poprzednich rozdziałach. Wytwarzanie oprogramowania sterowane modelami wymaga inwestycji w implementację transformacji, lecz w perspektywie wielu projektów zasoby zainwestowane w opracowanie transformacji powinny zwrócić się na zaoszczędzonym czasie architektów i projektantów.

10. Implementacja systemu, zarządzanie konfiguracją i zmianami

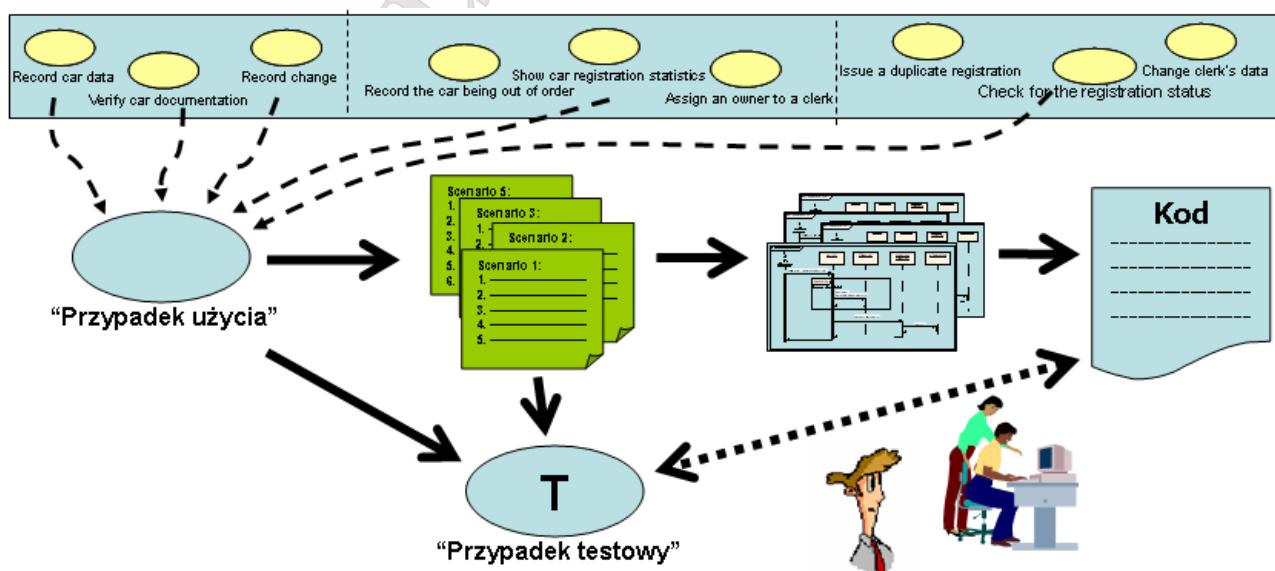
Implementacja systemu informatycznego polega na wykorzystaniu wiedzy zgromadzonej podczas tworzenia modeli wymagań, architektury i projektu szczegółowego. Celem implementacji jest zapisanie funkcjonalności systemu w obrębie wybranego środowiska implementacyjnego, przy zachowaniu wymagań jakościowych i przestrzeganiu ograniczeń środowiskowych. Czynność ta polega na tworzeniu kodu w wybranym języku lub zbiorze języków programowania oraz przygotowaniu artefaktów wpływających na działanie kodu. Tworzony kod powinien wypełniać ramy nakreślone projektem szczegółowym, jednocześnie go uzupełniając. Poza realizacją zapisów wymagań nie-funkcjonalnych, ograniczeń zewnętrznych czy zasad ustalonych dla komunikacji między fragmentami systemu, kod i jego powstawanie podlega wytycznym obowiązującym w danej organizacji, a także, w sposób mniej ścisły, w środowisku inżynierów oprogramowania.

Z pojęciem implementacji systemu nierozerwalnie związane są zagadnienia zarządzania konfiguracją i zmianami. Istotność tych zagadnień wynika z dynamicznego i często wielotorowego charakteru rozwoju systemu informatycznego. Prawidłowe planowanie zmian oraz odpowiednia polityka dotycząca konfiguracji ma istotny wpływ na sukces projektu informatycznego.

10.1. Kodowanie systemu na podstawie projektu

Opisane w poprzednich rozdziałach tworzenie wymagań oraz opisu środowiska klienta i rozwijanego systemu informatycznego to próby zrozumienia prawdziwych potrzeb zamawiającego, a także sposobu funkcjonowania jego otoczenia. Kolejny krok – utworzenie modeli architektonicznych i projektu szczegółowego – to z kolei przełożenie uzyskanej podczas analizy wymagań wiedzy na język zrozumiały dla osób ze środowiska technicznego. Język ten jest w założeniu jednoznaczny dla programistów, którzy w kooperacji z projektantami, „tłumaczą” projekt na kod wykonywalny zapisując za pomocą wybranych konstrukcji języków programowania informacje zawarte w modelu projektowym.

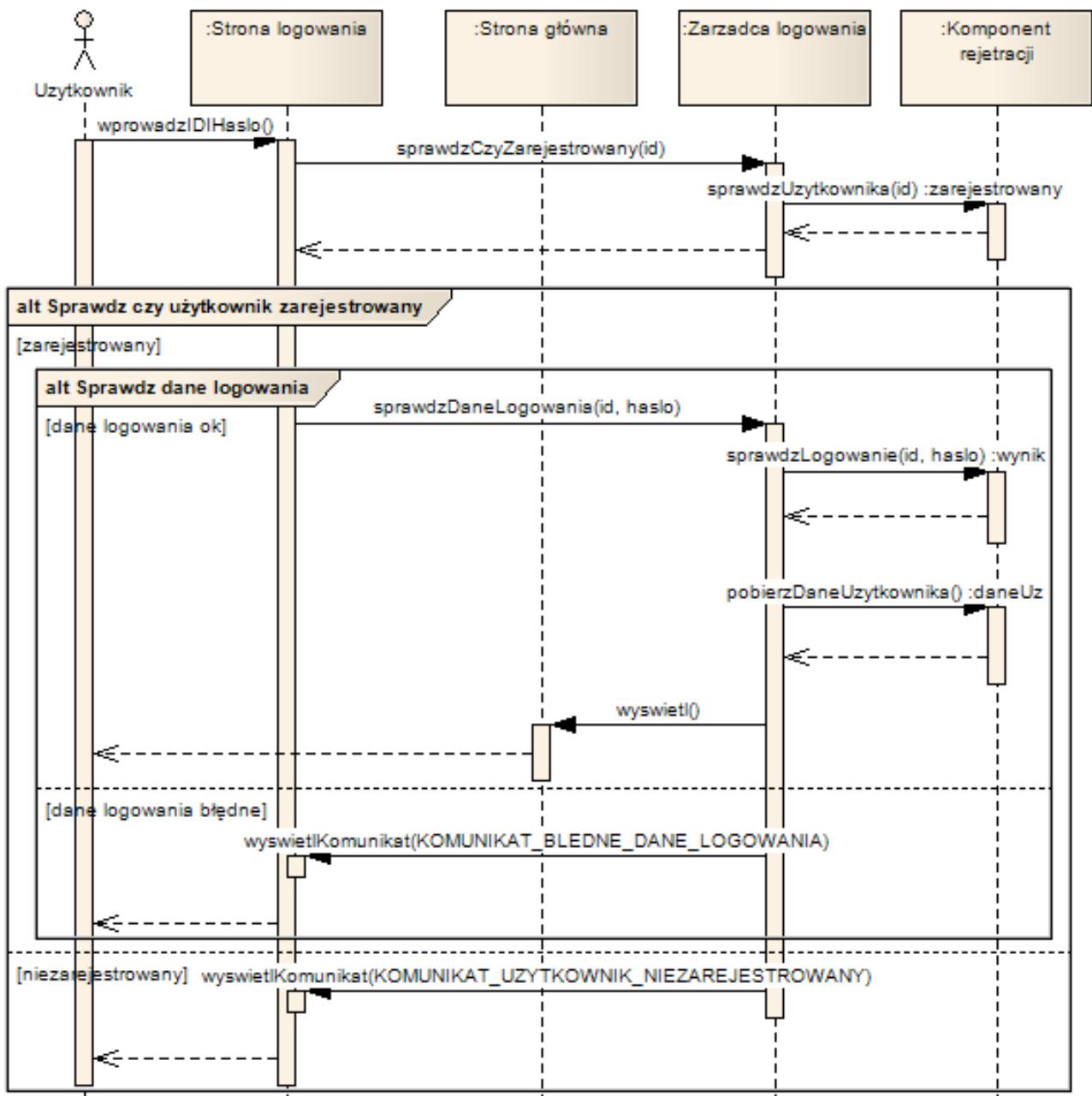
Rys. 10.1 ilustruje czynności na ścieżce „wiedza klienta – wymagania na system – modele systemu – kod”. O ile część tych operacji można zautomatyzować, to ich sens pozostanie niezmienny i polega na przenoszeniu wiedzy uzyskanej w poprzednim etapie do produktów etapu kolejnego: po zrozumieniu potrzeb klienta i utworzeniu wymagań, w postaci scenariuszy i scenopisów, na ich podstawie tworzone są diagramy interakcji, a następnie kod. Na bazie przypadków użycia pisane są przypadki testowe (patrz: sekcja 11.3), aby formalnie zweryfikować, czy system (działający kod) jest zgodny z wymaganiami oprogramowania.



Rys. 10.1: Proces „pisanie wymagań - tworzenie projektu – kodowanie i przygotowanie testów”

Przenoszenie do kodu rozwiązań architektonicznych i projektowych spełniających wymagania polega na stworzeniu ścieżek wykonania operacji opisanych modelami. Kluczowy jest tu aspekt dynamiki implementowanego systemu (której sposób modelowania opisano w rozdziale 6). Dla każdego komunikatu z diagramu interakcji (patrz: sekcja 6.2.1) w kodzie tworzone są odpowiednie wywołania metody lub funkcji. Zachowana jest kolejność wywołań. Typy parametrów wywołań odpowiadają typom określonym w modelu projektowym. Mogą one pochodzić z języka modelowania, docelowego języka programowania systemu lub być zdefiniowane jako klasy w „słowniku” projektu. Parametry mają takie wartości, jak określono na diagramach interakcji lub w innej dokumentacji projektu. Fragmentom włączonym na diagramach interakcji odpowiadają instrukcje przepływu sterowania na poziomie kodu.

Diagram interakcji na Rys. 10.2 przedstawia uproszczony fragment dynamiki działania systemu zamodelowanego na poziomie projektu szczegółowego. Poniżej diagramu znajduje się kod w języku Java będący odzwierciedleniem komunikatów opisujących zachowanie metody sprawdzDaneLogowania(...).



Rys. 10.2: Przykładowy diagram interakcji

Ciało metody sprawdzDaneLogowania wypełniają kolejno operacje:

- wywołanie metody sprawdzLogowanie(...) z KomponentRejestracji implementującą odpowiedni komunikat na diagramie
- sprawdzenie wyniku wywołania metody sprawdzLogowanie(...) – odpowiada to fragmentowi typu alternatywa o nazwie „Sprawdz dane logowania”
- w przypadku gry powyższy warunek jest spełniony („dane logowania ok”) wykonywane są operacje wywołania metod pobierzDaneUzytkownika (z KomponentRejestracji) oraz wyświetl (z komponentu StronaGlowna)
- w przypadku niespełnienie warunku („dane logowanie błędne”) wywoływana jest metoda wyświetlKomunikat z komponentu StronaLogowania

```
1 public void sprawdzDaneLogowania(long id, String haslo) {  
2     int wynik = komponentRejestracji.sprawdzLogowanie(id, haslo);  
3     if(wynik == LOGOWANIE_OK) {  
4         DaneUzytkownika daneUz  
5             = komponentRejestracji.pobierzDaneUzytkownika();  
6         stronaGlowna.wyswietl();  
7     }  
8     else if(wynik == LOGOWANIE_BLAD) {  
9         stronaLogowania  
10            .wyswietlKomunikat(KOMUNIKAT_BLEDNE_DANE_LOGOWANIA);  
11    }  
12 }
```

Taka postać kodu jest niesatisfakcyjną z punktu widzenia jakości implementowanego systemu – kod ten powstał przez mechaniczne przeniesienie informacji projektowej zawartej na jednym diagramie (z wyjątkiem wykorzystania stałych i pól klas), bez uwzględnienia wymagań niefunkcjonalnych i innych ograniczeń. Docelowy kod powinien zostać przez programistów przystosowany wg. zasad panujących w danej organizacji, a także zgodnie prawidłami sztuki, przykładowo: dane wejściowe metody poddaje się zazwyczaj walidacji, wyniki wywoływanych metod powinny zostać sprawdzone, dostęp do pól klas odbywa się za pośrednictwem metod dostępowych, wyjątki i/lub komunikaty o błędach o ścisłe określonej hierarchii nie są w tym przykładzie uwzględnione, w prezentowanym kodzie brakuje komentarzy itd. Zaproponowane zmiany powinny zostać nieniesione w modelu projektowym (np. wprowadzenie wywołania walidujDaneLogowania(id, haslo) w linii 2, uzupełnia diagram interakcji o dodatkowy komunikat).

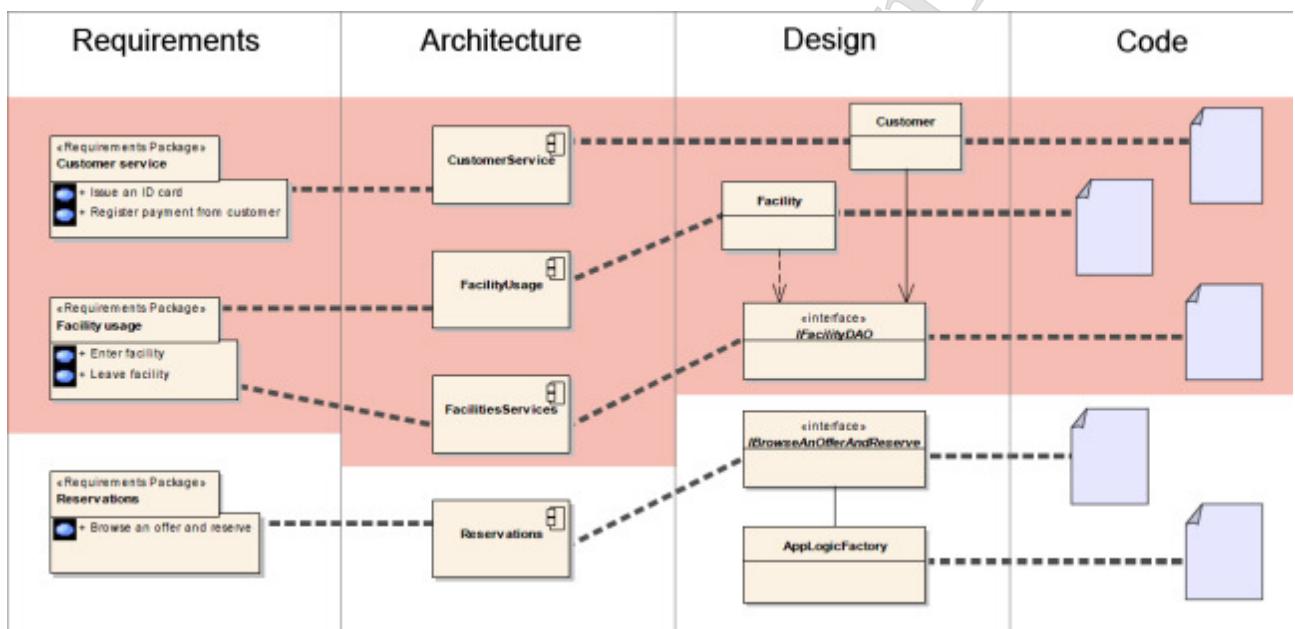
W rzeczywistym projekcie każda inna metoda z diagramu Rys. 10.2 powinna zostać opracowana wg. powyżej zaprezentowanego opisu. Wypełniając wnętrza metod tworzy się de facto wnętrza klas budujących komponenty. Wykorzystany tutaj przykład pomija zagadnienia udostępniania przez komponenty interfejsów, kontraktów² i komunikacji między interfejsami a komponentami. W przykładzie uproszczono lub zaniedbano także inne kwestie związane z komponentową organizacją kodu, jak realizację komponentów przez klasy, wewnętrzne zależności między klasami w obrębie

² Kontrakt jest to formalna, precyzyjna specyfikacja warunków wstępnych i końcowych dotyczących usług realizowanych przez interfejsy

bie komponentu, a także artefakty zewnętrzne w stosunku do kodu, wpływające na jego działanie (np. pliki konfiguracji).

Jak już wspomniano część operacji związanych z tworzeniem kodu można zautomatyzować. Stosując transformacje możemy wygenerować szkielet modelu architektury i projektu szczegółowego oraz szkielet kodu na podstawie wymagań (Rys. 10.3). Jest to uproszczona realizacja koncepcji MDD (Model-Driven Development, patrz: sekcja 3.3). Modele pośrednie między kodem a wymaganiami mogą podlegać ręcznym modyfikacjom – operacja wymagania → kod nie jest wykonywana w jednym kroku. Zagadnienia związane z generacją kodu z modeli zostały opisane w sekcji 10.6.

Wartym uwagi jest fakt, że wynikające z transformacji modele i kod są niejako parametryzowane wymaganiami i sposobem przeprowadzenia transformacji, co pozwala na podstawie jednego modelu wymagań i zestawu transformacji tworzyć systemy w różnych środowiskach technicznych (patrz: sekcja 3.3). Model wymagań, poddany opartym o różne szablony architektoniczne transformacjom, pozwala uzyskać szereg odpowiadających mu modeli architektury, z których każdy można przekształcać na różne, znowu zależne od wybranego profilu transformacji, modele projektu szczegółowego.



Rys. 10.3: Propagacja informacji zawartej w wymaganiach poprzez transformacje

Zagadnienia związane z generacją kodu zostały opisane dokładniej w sekcji 10.6. Szczegóły procesu powstawania testów opisano w rozdziale 11. Sekcja kolejna zawiera wskazówki dotyczące pewnych aspektów tworzenia kodu.

10.2. Dobre praktyki w zakresie kodowania

Jak można się zorientować nawet z prostego przykładu zawartego w poprzedniej sekcji, nakład pracy związany z kodowaniem jest znaczący, a sama czynność implementacji ma istotny wpływ na ostateczny kształt systemu. Z tego powodu powstawanie kodu w ramach każdego projektu informatycznego powinno odbywać się z troską o wysoką jakość wytwarzanych produktów. Nie da się tego uzyskać, o ile nie uwzględnili się pewnych czynników natury ogólnej, nie związanych konkretnym projektem informatycznym.

Kod, tak jak inne artefakty procesu wytwarzania oprogramowania, jest tworzony i czytany przez osoby o różnych kwalifikacjach, doświadczeniu, nawykach itp. Aby zapobiec trudnościom podczas zespołowej pracy nad kodem środowisko inżynierów oprogramowania, twórców języków i programistów wypracowało szereg zasad, zwanych „dobrymi praktykami kodowania”. Praktyki te, oparte na wieloletnim doświadczeniu branży informatycznej, w obiektywny sposób opisują pewne pożą-

dane cechy kodu – niezależnie od języka programowania, rozwiązań technicznych, charakteru organizacji informatycznej czy dziedziny projektu, w obrębie którego kod powstawał.

Pierwszą przesłanką, którą każdy programista powinien uwzględnić podczas pracy z kodem jest fakt, że nie pisze on kodu dla siebie. Troska kodera o przyszłego czytelnika objawia się w szeroko rozumianej łatwości przyswajania znaczenia kodu. Kod powinien być czytelny i zrozumiałym przy minimalnym nakładzie pracy, a także, na ile to możliwe, prosty do zmiany dla kompetentnej osoby w obrębie danej organizacji czy środowiska.

Warto przy tej okazji zauważać istnienie wśród programistów pewnego przesądu: otóż niektórzy z nich uważają, że pisanie nieczytelnego kodu podnosi ich wartość dla organizacji, ponieważ stają się oni jedynymi ekspertami w zakresie takiego zagmatwanego i niepotrzebnie skomplikowanego fragmentu aplikacji. Owa „niezbędność” to jednak broń obosieczna: inni pracownicy w obrębie organizacji szybko przyzwyczają się, że każdą, najmniejszą pracę, nawet tylko luźno związaną ze „specjalnie skomplikowanymi” modułami musi wykonać ich „właściciel” (i to wykonać jak najszybciej – nikt nie jest przecież w stanie, nie rozumiejąc zawartych w sztucznie nieczytelnym kodzie rozwiązań, ocenić nakładu pracy wymaganego dla zmian). Innym, może nawet istotniejszym argumentem jest problematyczna konserwacja własnego kodu po upływie dłuższego czasu: pamięć ludzka jest zawodna i powrót do pracy nad kodem zawierającym „haczyki” i sztuczne utrudnienia może zaowocować frustracją nawet u oryginalnego implementatora takich fragmentów.

Drugą ważną wskazówką dla programisty jest fakt, że kod, który pisze, powstaje w ścisłe określonym celu jakim jest realizacja projektu, a nie jedynie po to, by był czytany przez współpracowników – kolegów z pracy. Jak już wyjaśniono – należy dbać o kolejnego czytelnika kodu, ale nie wolno przytaczać prawdziwego przeznaczenia implementowanych fragmentów własną osobowością, w celu wywarcia na ich czytelniku „dobrego” wrażenia. Przykładami takich niepożądanych działań jest praktykowane przez niektórych koderów niepotrzebne popisywanie się wiedzą i umiejętnościami. Może się to objawiać poprzez upychanie zbyt wielu operacji w jednej linii programu, stosowanie zaawansowanych algorytmów czy bibliotek w bardzo prozaicznych celach, realizowanie ambicji niespełnionych projektantów, którzy przy prostych zadaniach tworzą niezrozumiałe struktury programistyczne itd.

Jak wynika z powyższych rozważań, kodu nie pisze się ani dla siebie, ani dla swoich współpracowników; kod powstaje by realizować założenia projektu w taki sposób, aby „jutro” nieznana nam osoba mogła z nim komfortowo pracować. Z tego względu „dobre praktyki kodowania” można podzielić na praktyki „edytorskie”, „merytoryczne” oraz związane z nawykami codziennej pracy.

Wśród praktyk związanych z kodem, traktowanym jako tekst, można wyróżnić:

- formatowanie: już przykład z poprzedniej sekcji zawiera praktyczną ilustrację kilku zasad. Kod powinien zawierać wcięcia i odstępy organizujące go w logiczne bloki. Linie nie powinny być zbyt długie, aby można je było łatwo objąć wzrokiem (warto zwrócić uwagę na charakterystyczne złamanie linii 9 w przykładzie)
- odpowiednie komentowanie: jest o tyle istotne, że często kod nie mówi sam za siebie – może być nieoczywisty, zawierać pozornie niejednoznaczne odwołania, mieć nieewidentne motywacje. Komentarze mogą występować w kodzie na wielu poziomach abstrakcji – mogą służyć nawiązaniu roli modułu (często jest to tekst przeniesiony z dokumentacji projektowej), opisaniu kontraktu dla metody/funkcji (także, przynajmniej częściowo opisanego w projekcie), a także wyjaśnieniu działania konkretnej operacji. Współczesne języki programowania oferują parę typów komentarzy właśnie w celu wyróżnienia komentarzy wysokiego poziomu od tych bardziej szczegółowych, lokalnych. Komentarze wyjaśniające działanie kodu zgrubnie, mają często swoją własną składnię (jak JavaDoc dla języka Java, czy XML dla C#) i pozwalają na automatyczne przetwarzanie – w celu generacji dokumentacji czy „podglądu” metod znanego z wielu IDE³. Komentarze lokalne (dla linii instrukcji przepływu sterowania, przypisań i wywołań procedur/funkcji) mogą mieć postać pojedynczych linii lub blokową – należy umiejętnie korzystać z obu form. Sama zawartość komentarzy powinna w sposób wystarczający i zwięzły wyjaśniać

³ Zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*), więcej na ten temat w sekcji 12.3

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 odpowiadając sobie na pytanie „co jest oczywiste (dla mnie/dla innych)?”. Dobrą wskazówką do poprawy lub uściślenia konkretnych komentarzy (albo ogólnie: stylu ich pisania), jest sytuacja, gdy powrocie do pracy nad danym kodem po dłuższym czasie, jego zrozumienie sprawia trudność, mimo istnienia komentarzy.

- przestrzeganie konwencji 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.

Od strony „merytorycznej” dobre praktyki pisania kodu obejmują:

- projektowanie także podczas pisania kodu: projekt oddawany w ręce programistów nie jest „martwy”, czy „zaplombowany” – koderzy opracowując komponenty projektują ich wnętrze dokonując dekompozycji klas realizujących interfejs, rozbijając metody zmniejszając ich poziom komplikacji itd. Wprowadzając takie zmiany w kodzie, powinni jednocześnie nanosić je w modelu projektowym.
- zapewnienie możliwie wysokiej parametryzowalności kodu: wszystkie używane w kodzie stałe (napisy, liczby, symbole itp.) powinny być definiowane poza kodem, który się do nich odwołuje. Dobre, gdy stałe wykorzystywane globalnie (np. kody błędów) określa się w sposób skonsolidowany (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 permanentnych 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 związane z omówioną możliwością parametryzacji kodu. Unikanie formuł przekazywanych bezpośrednio, a nie określonych symbolicznie wiąże się 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 PROC_PRIORITY_HIGH = 5.
```

Unikanie „magicznych wartości” ogranicza też szanse pomyłek związanych z literówkami (które nie są wskazywane jako błędy przez kompilatory).

- posługiwanie się wzorcami projektowymi: programista, podobnie do projektanta, często ma możliwość zastosowania ogólnie przyjętych i sprawdzonych rozwiązań. Wśród najważniejszych wzorców używanych przez programistów należy wymienić (Abstract)Factory, Singleton, Command [Gamm08], a także Lazy Initialization (ang. leniwa inicjalizacja: technika powodująca wypełnianie obiektów danymi tylko, gdy zaistnieje taka potrzeba), Private Constructor (ang. prywatny konstruktor: przesłanianie pewnych konstruktorów przed dostępem „z poza” klasy). Warto zauważać, że każda rozbudowana technologia obiektowa posiada swoje własne wzorce np. [Stel05]. Zapoznanie się z nimi ułatwia programistom zrozumienie modelu projektowego, a także tworzenie kodu opartego na dobrych rozwiązaniach. Więcej informacji o wzorcach projektowych zawiera sekcja 8.5.
- unikanie antywzorców: tak jak środowisko inżynierów oprogramowania stworzyło wzorce projektowe, tak samo rozpoznano szereg powtarzalnych błędów popełnianych przez projektantów i programistów. Są one dość dobrze opisane w literaturze. Każdy programista powinien zapoznać się z nimi i unikać sytuacji, w których ich stosowanie może obniżać jakość jego kodu (bo nie zawsze też antywzorce są „szkodliwe”). Przykładowymi antywzorcami są *Interface for constants* w języku Java (interfejs nie definiujący funkcjonalności, a jedynie stałe), korzystanie z

globalnych zmiennych w językach proceduralnych), czy tworzenie klas danych typu *Bean* (z bezparametrowym konstruktorem i wieloma metodami dostępowymi do pól).

- przywiązywanie uwagi do testów związanych z kodem: moment powstawania testów jest zależny od metodyki i procesu wytwórczego przyjętego w projekcie, ale przyjmuje się testy powinny powstawać, w oparciu o wymagania użytkownika, mniej więcej równocześnie z kodem, który testują. Ułatwia to pracę programistów o tyle, że mogą wcześnie testować swój kod w warunkach zbliżonych do określonych przez użytkownika, co pozwala na wyłapywanie poważnych błędów zanim implementacja wejdzie w zaawansowane stadium.
- przeprowadzanie walidacji parametrów wejściowych metod: każdy programista powinien pilnować kontraktów określonych dla implementowanych przez siebie modułów sprawdzając nie tylko, czy zwraca poprawne dane, ale także walidując wartości parametrów z jakimi wywoływane są jego metody (szczególnie te publiczne, będące elementami kontraktu). Nigdy nie należy zakładać, że procedura albo funkcja zostanie wywołana z „dobrymi” danymi – walidację należy zacząć od sprawdzania pustych wskaźników i referencji, 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 zawartych w dokumentacji technicznej projektu (wyjątki i logowanie): koordynatorzy projektu powinni przygotować dokumenty określające aspekty techniczne pisania kodu. Programista powinien zapoznać się z tymi tekstami, szczególną uwagę przywiązuje do realizacji zapisu wykonywanych operacji w dzienniku (poziomy logowania, format wpisów, pliki docelowe, użyta technologia dziennika) oraz do hierarchii wyjątków (przestrzeganie polityki reakcji na błąd, unikanie niepotrzebnego filtrowania wyjątków, a także zasypywania nimi klas wywołujących)
- dbałość o dane przetwarzane przez aplikację: historia inżynierii oprogramowania uczy, że dane aplikacji trwają najczęściej dłużej niż czas „życia” tychże aplikacji. Często aplikacje tworzone są w celu przetwarzania danych istniejących od wielu lat, przy założeniu, że za parę lat zostaną zastąpione przez kolejne systemy pracujące na tych samych danych (przykładami są tu systemy banków, czy firm ubezpieczeniowych). Programiści muszą tworzyć kod szanujący „odziedziczone” dane (ang. *legacy data*) i nie generujący w obrębie starych schematów bazodanowych „śmieci”. Z zagadnieniem dbałości o dane łączy się problem szczególnie istotne w środowiskach wielowątkowych: zabezpieczenie danych przed jednoczesnym dostępem do nich przez różne procesy czy aplikacje. Programista powinien sprawdzać możliwość „zamykania” zestawów informacji przed ich odczytem lub zapisem, uważając jednak na możliwość zmniejszenia wydajności całego systemu spowodowanej blokowaniem dostępu do danych.
- dbałość o zasoby maszyn: programista powinien pamiętać, że jego kod nie będzie jedynym działającym na komputerach służących aktualnie rozwijanemu systemowi i dlatego powinien zapoznać się z metodami odzyskiwania pamięci, wielowątkowości czy optymalizacji kodu dla technologii, z których korzysta.
- branie pod uwagę możliwej przenośności: czasami (i jest tak coraz częściej) wymogiem stawianym przez użytkownika jest przenośność aplikacji. Rynek obfituje w technologie i rozwiązania wspierające przenośność, także projektanci dysponują doświadczeniem pozwalającym ją uzyskać, jednak często programiści, podczas implementacji niskopoziomowych funkcji rozwijanego systemu, zapominają o fakcie, że ich 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 możliwej skalowalności: wielu programistów uważa, że *skalowalność* to termin przynależny dziedzinie architektów i twórców sprzętu. Rzeczywistość pokazuje niestety, że często za słabą skalowalność systemów dopowiadają błędy programistów. Czasami piszą oni swój kod nieświadomi, że będzie on wywoływany wielokrotnie w krótkich odstępach czasu i nawet drobne błędy (popularny przykład: konkatenacja „surowych” łańcuchów w języku Java, a nie wykorzystanie rozwiązania typu StringBuffer) mogą powodować problemy stabilności całej

aplikacji. Dobrym rozwiązaniem jest tutaj kooperacja programistów z architektami, którzy powinni informować koderów, które miejsca w systemie będą szczególnie obciążane. Także tworzenie odpowiednich testów obciążeniowych dla konkretnych fragmentów kodu pozwoli wcześniej rozwiązać wspomniane problemy.

- pisanie prawdziwego kodu zorientowanego obiektywnie: popularnym błędem (i jednocześnie antywzorcem) jest pisanie kodu klas w stylu wywodzącym się z języków proceduralnych, polegającym na niejawnych założeniach co do kolejności wykonania metod. Instancje klas mogą mieć zdefiniowany jakiś charakterystyczny cykl życia, ale tylko w bardzo charakterystycznych przypadkach, przy dobrym udokumentowaniu tego faktu, a także przy obwarowaniu wykonania metod walidacją już nie tylko parametrów wejściowych, ale także kontrolą stanu całego obiektu. Inny „proceduralny” błąd spotykany w kodzie zorientowanym obiektywnie, to korzystanie z pól statycznych nie będących stałymi (jest to odpowiednik zmiennych globalnych w językach proceduralnych).

Dobre praktyki związane z pewnymi nawykami, które powinien mieć każdy programista dotyczą głównie pracy zespołowej i systemów kontroli wersji (opisanych dokładnie w sekcji 10.3).

Ważnym jest aby kod podlegający kontroli wersji był „czysty” – pozbawiony błędów komplikacji, ostrzeżeń (ang. *warnings*), a także wad 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.

Także przed rozpoczęciem pracy z kodem pochodząącym z repozytorium warto sprawdzić, że lokalna jego kopia jest „czysta” przez uaktualnienie jej w całości, a następnie uruchomienie testów.

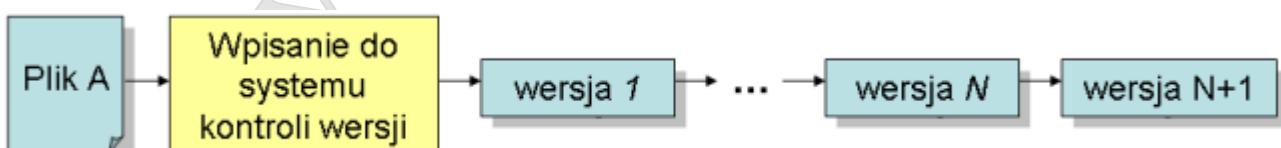
Innym dobrym nawykiem jaki powinien sobie wyrobić programista jest umiejętne korzystanie z jego IDE – bardzo pomocne w pracy są funkcje takie jak auto-uzupełnianie, fragmenty-szablony kodu (ang. *snippets*) gotowe do użycia. Korzystanie z tego typu funkcji i dobrze skonfigurowanego środowiska znaczco podnosi produktywność kodera.

Dobrą radą pozwalającą poprawić jakość pisanej kodu jest obserwacja fragmentów programów dostępnych w repozytoriach projektów typu Open Source – sztandardowe projekty tego typu mają bardzo dobre wyniki jeśli chodzi o zachowanie standardów i „dobrych praktyk” kodowania.

Badanie poziomu przestrzegania reguł kodowania ustalonych dla danego przedsięwzięcia informatycznego może podlegać znacznej formalizacji – patrz sekcja 11.3.

10.3. Kontrola wersji

Kontrola wersji (ang. *revision control*) to sposób zarządzania historią artefaktów i ich zbiorów związanych z danym projektem informatycznym lub organizacją. Artefakty (kod, dokumenty, modele, dane binarne itp.) utożsamiane są z plikami – tekstowymi i binarnymi. System kontroli wersji pozwala na przywołanie dla każdego pliku jego poprzednich stanów, a także określenie parametrów zmian dla niego.

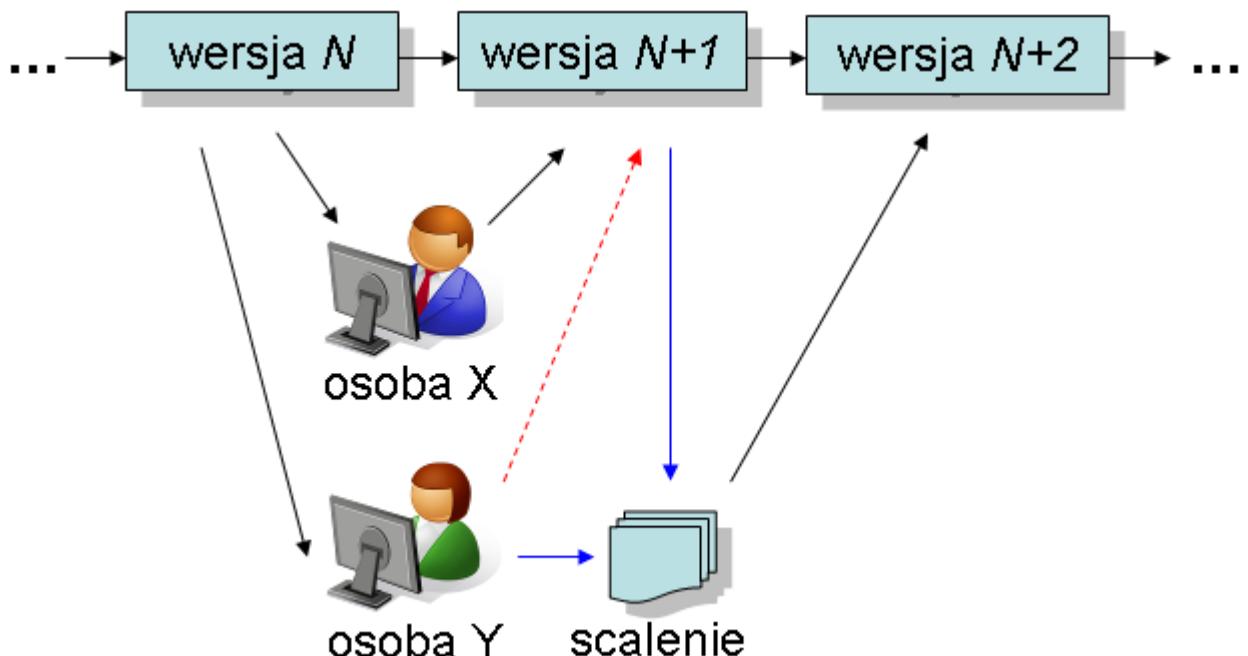


Rys. 10.4: Kolejne stany pliku poddanego kontroli wersji

Istnieją dwa modele systemów kontroli wersji: scentralizowane (klient-serwer) oraz rozproszone. Komercyjną popularność zyskały te pierwsze, zakładające istnienie centralnego repozytorium i synchronizujących z nim swoje informacje użytkowników. Centralizacja zarządzania wersjonowaniem pozwala na typowe dla systemów zunifikowanych zorganizowanie pracy grupowej, zarządzanie kopiami zapasowymi, importowanie i eksportowanie danych, czy użycie systemów ciągłej integracji, komplikacji i testowania.

W obrębie modelu scentralizowanego istnieją dwa rodzaje strategii zarządzania zrównoległonym procesem zmian artefaktów: poprzez blokady (ang. *lock*) i poprzez scalenia (ang. *merge*). Pierwsza strategia zakłada możliwość zablokowania przez użytkownika wersjonowanego zasobu, by uniemożliwić innym zapis – stworzenie nowej wersji (odczyt jest cały czas dopuszczalny). Blokowanie wykorzystuje się, aby zawartość zasobu nie była modyfikowana jednocześnie przez wielu użytkowników. Strategia polegająca na scaleniach zakłada, że istnieje mechanizm stworzenia spójnego i użytecznego artefaktu w oparciu o jego wersję N, wersję N+1 oraz zawartość bazującą na wersji N, ale posiadającą różnice (powodujące nawet sprzeczność) z wersją N+1. Rys. 10.5 przedstawia sytuację, gdy osoby X i Y pracują na tej samej wersji zasobu, a osoba „X” zapisuje swoje zmiany jako pierwsza: osoba „Y” otrzymuje informacje o nieaktualności swojego zasobu podczas próby zapisu do repozytorium. System umożliwia scalenie zmian i zapis w postaci kolejnej rewizji.

Modele kontroli wersji typu rozproszonego są rzadsze w przemyśle i opierają się na założeniu równorzędności praw uczestniczących w procesie twórczym (ang. *peer-to-peer, p2p*). Pozwalają one na większą niezależność partycypujących osób, lecz jednocześnie (i jest to główny powód odrzucenia tej strategii przez przemysł) utrudniają zarządzanie zarówno kierunkiem procesu, jak i jego przeszłyimi efektami.



Rys. 10.5: Scalenie zmian w systemie kontroli wersji

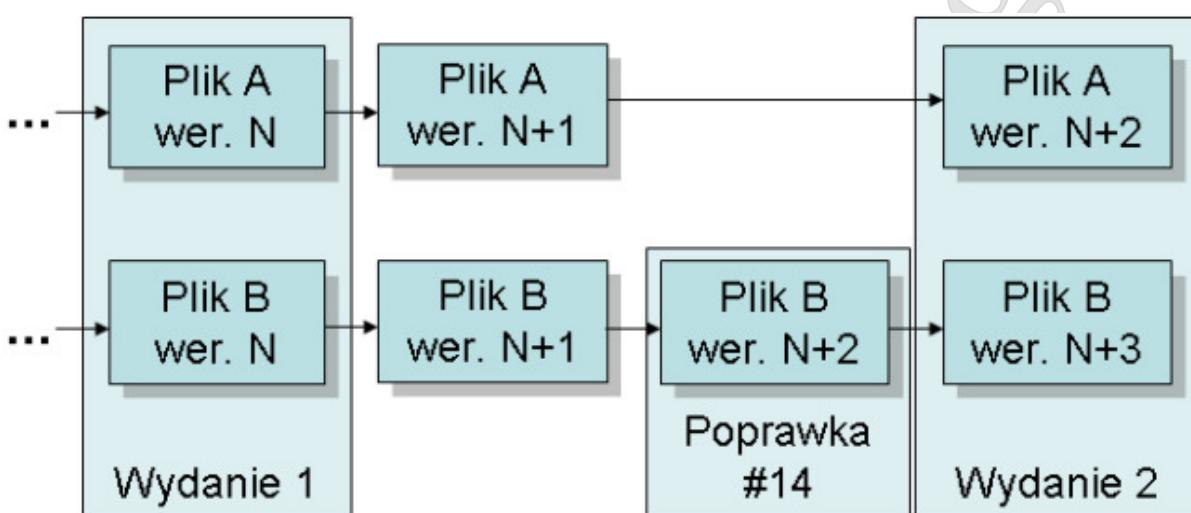
Kontrola wersji łączy się z paroma zjawiskami i pojęciami, których zrozumienie jest niezbędne w celu pracy w obrębie takiego systemu. Każdemu zarchiwizowanemu stanowi artefaktu (zasobu, pliku) – wersji – odpowiada opis, obejmujący najczęściej:

- 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 *delta* (zmiana) w stosunku do rewizji ją poprzedzającej. Często delta jest kompresowana w celu dalszego ograniczenia potrzebnych na jej przechowywanie i przesyłanie zasobów.

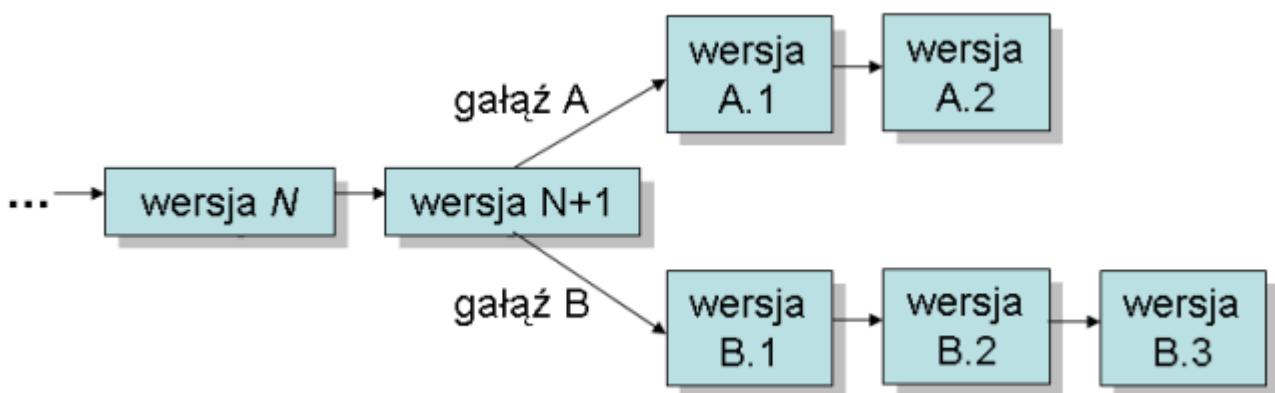
Inne istotne pojęcia dotyczące kontroli wersji:

- głowa (ang. *head*): ostatnia, najnowsza wersja
- znacznik, etykieta (ang. *tag*): symboliczny opis (nazwa) numeru wersji ułatwiający oznaczenie istotnych rewizji (np. wydań). W systemach, gdzie każdy plik ma osobną numerację może dotyczyć wielu wersji różnych plików. Przykłady znaczników znajdują się na Rys. 10.6: etykieta „Poprawka #14” dotyczy pojedynczej wersji wybranego pliku, etykiety „Wydanie 1” i „Wydanie 2” odnoszą się do zbiorów plików. „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



Rys. 10.6: Znaczniki dla plików w systemie kontroli wersji

- gałąź (ang. *branch*): zasób będący pod kontrolą wersji może posiadać wiele, rozwijanych z różną prędkością wariantów (gałęzi) – Rys. 10.7
- pień (ang. *trunk*): główna ścieżka („gałąź”) rozwoju pliku, w stosunku do której wprowadzane są zmiany



Rys. 10.7: Schemat wersjonowania dla zasobu o dwóch gałęziach

- kopia robocza (ang. *working copy*): lokalna kopia wersji zachowanych w systemie kontroli wersji
- *check out*: stworzenie kopii roboczej. Nie jest to jedynie eksport danych z repozytorium, ale także pobranie dodatkowych informacji usprawniających pracę z systemem kontroli wersji.
- *check in*: wpisanie do systemu kontroli wersji stanu wybranych zasobów z kopii lokalnej
- aktualnienie (ang. *update*): aktualizacja plików w kopii lokalnej

- konflikt: sytuacja, w której pliki kopii roboczej są w niej opisane jako wynikające z wersji N, natomiast w repozytorium istnieje wersja N+1. Rozwiążanie konfliktu polega na scaleniu zmian repozytorium z lokalnymi (Rys. 10.5)
- diff (ang. *difference* - różnica): sposób (a także narzędzie) porównywania zawartości dwóch rewizji pliku, bądź rewizji i danego stanu zasobu

Wśród popularnych systemów scentralizowanej kontroli wersji możemy wyróżnić:

- oparte na „otwartej” licencji (GPL, BSD, Apache itp.)
 - Concurrent Versions System (CVS) – oparty na fundamentalnym Revision Control System (RCS), zorientowany na przechowywanie plików tekstowych
 - Subversion (SVN) – „odpowiedź” na problemy związane z użytkowaniem CVS – posiada wsparcie dla zasobów binarnych, repozytorium oparte jest o bazę danych (charakteryzuje ją dobra wydajność, system jest transakcyjny, pozwala na przechowywanie meta-danych)
- komercyjne
 - Microsoft Visual SourceSafe – system przechowujący dane w plikach, do których dostęp kontrolowany jest poprzez system blokad; aktualnie zastępowany przez funkcjonalność Team Foundation Server
 - Perforce – system oparty o bazę danych pozwalający na transakcyjną pracę z dowolnymi typami zasobów

Wśród systemów rozproszonych najpopularniejsze są Bazaar, SVK i Code Co-op.

Schemat numeracji wersji nie jest w wymienionych systemach jednakowy. CVS numeruje kolejne wersje dla każdego pliku wg. szablonu **x.y**, gdzie x i y to liczby naturalne (numeracja zaczyna się od wersji „1.1”). SVN z kolei nadaje kolejny numer (startując od 0) zbiorowi plików objętych nowym wpisem do repozytorium (kolejne numery rewizji stają się de facto liczbowymi znacznikami dla jednego lub więcej plików).

Większość popularnych systemów kontroli wersji, poza narzędziami dostarczonymi przez producenta, posiada wiele dodatkowych „nakładek”, „wtyczek” itp. Szeroko rozpowszechnione IDE, zarządcy plików, aplikacje śledzenia błędów (ang. *bug trackers*), narzędzia wspierające ciągłą integrację, analizatory kodu itp. posiadają najczęściej wsparcie pozwalające na integrację z systemami kontroli wersji. Wpisanie kontroli rewizji w centrum procesu wytwarzania oprogramowania pozwala na usprawnienie pracy zespołów, a także na efektywne zarządzanie zmianami.

10.4. Zarządzanie zmianami

Każdy system informatyczny, niezależnie od fazy w jakiej się znajduje, podlega mniej lub bardziej istotnym zmianom. Są one wynikiem okoliczności związanych ze sposobem rozwoju oprogramowania, a także czynników zewnętrznych: zmian prawnych, organizacyjnych, cyku koniunkturalnego, potrzeb użytkowników, rozwoju infrastruktury, wyników analizy działania systemu w obrębie wspieranego biznesu itd.

Proces nieuniknionych zmian tłumaczony jest prawem ciągłych zmian Lehmana [Lehm80]:

System, z którego korzystają użytkownicy, podlega ciągłej zmianie lub obniża swoją wydajność.

Efektem ewolucji oprogramowania [Lehm80] jest narastająca komplikacja systemów. Opisuje to prawo rosnącej złożoności Lehmana:

Program komputerowy, który jest zmieniany, staje się coraz mniej uporządkowany. Zmiany powodują wzrost entropii i złożoności programów.

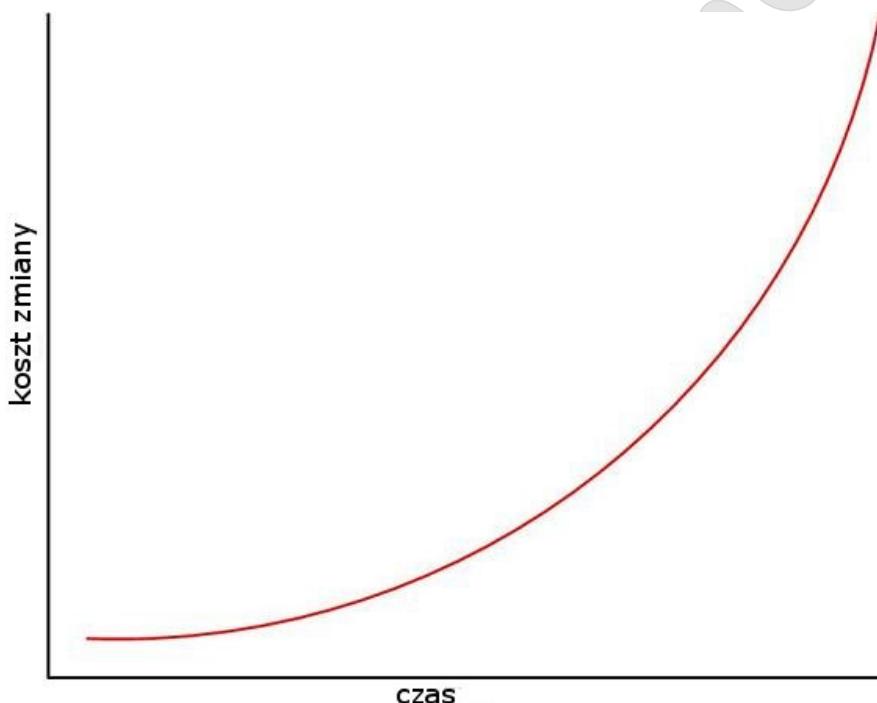
Poziom złożoności aplikacji rośnie nie tylko z powodu dodawanych do niego funkcjonalności, co powoduje wzrost liczby interakcji między komponentami. Często wprowadzone zmiany nie były, nawet hipotetycznie, brane pod uwagę w oryginalnym projekcie, a nowe elementy pojawiają się w

programie w sposób sztuczny, niejako wymuszony warunkami zewnętrznymi – stąd „spadek uporządkowania” takiego programu.

Jak już wielokrotnie wspominano rozwój oprogramowania jest złożony: w procesie twórczym 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 znaczną liczbę interakcji obserwowanych podczas produkcji systemu. Gdy weźmiemy pod uwagę dodatkową ilość czynników wynikających ze zmian i pogłębiających poziom komplikacji oprogramowania, odpowiednie zarządzenie zmianami i ich wpływem na proces twórczy wydaje się nieodzowne.

Analizując zagadnienie zmian od strony ekonomii projektu widać, jak istotnym jest wczesne, zdecydowanie tańsze wprowadzanie zmian (Rys. 10.8). Jednak wykorzystanie odpowiedniego zarządzania zmianami zabezpiecza projekt przed nieoczekiwanyimi, późnymi zmianami.

W celu sprawnego zarządzania zmianami niezbędne jest wykorzystanie systemu kontroli wersji, pozwalającego na śledzenie historii zmian zasobów powiązanych z projektem, a także utrzymywanie relacji między wersjami plików w repozytorium a dokumentacją zmian.



Rys. 10.8: Koszt zmiany w zależności od czasu [Ambl06]

Niezmierne istotne jest również dokumentowanie zmian, na wszystkich poziomach abstrakcji (od wymagań do kodu). Każda rewizja powinna posiadać szereg atrybutów, np.:

- źródło zmiany
- opis zmiany
- opis dodatkowych efektów zmiany (np. wymaganych zmian w innych niż zmieniany komponentach); w tym podkreślenie 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.)

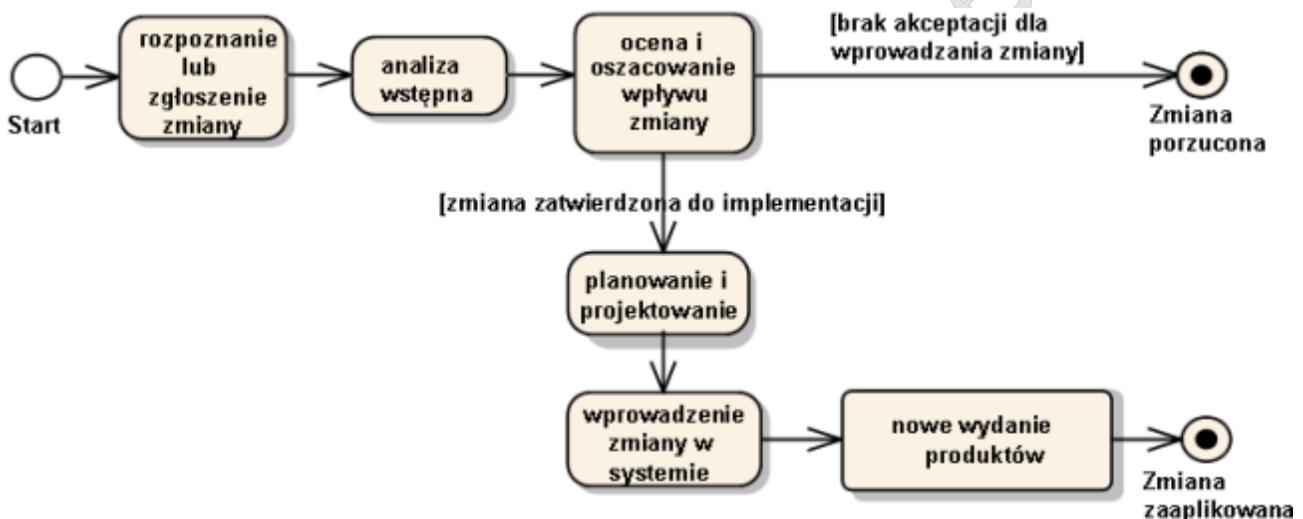
Zmiany powinny być wprowadzane wg. ściśle określonej procedury. Przykładowy proces aplikacji zmiany może obejmować następujące fazy (zilustrowane na Rys. 10.9):

- rozpoznanie lub zgłoszenie zmiany: źródeł 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 przeprowadzonej w dwóch powyższych punktach podejmowana jest decyzja co do ewentualnej aplikacji zmiany. W przypadku, gdy decyzja jest pozytywna przeprowadzane są następujące kroki:

- 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.



Rys. 10.9: Uproszczony proces aplikacji zmiany w postaci diagramu aktywności

O ile nie należy utożsamiać zmiany z poprawką (zmianami w systemie wynikającymi z jego wadliwego, niezgodnego z wymaganiemi funkcjonowania), to proces wprowadzania poprawek przypomina przedstawiony powyżej.

Z zagadnieniem wprowadzania zmian i tworzenia na ich bazie nowych produktów łączy się pojęcie zarządzania konfiguracją.

10.5. Zarządzanie konfiguracją

Nieuchronność zmian powodujących wzrost stopnia komplikacji systemu informatycznego wymaga stosowania pewnych procedur i zasad pozwalających na opanowanie złożoności kodu i modeli oraz możliwości powstających wersji produktów. Ustalenie i aplikację zbioru takich standardów postępowania określa się zarządzaniem konfiguracją.

Jak wspomniano w poprzedniej sekcji wprowadzanie zmian powoduje powstawanie nowych wydań produktów systemowych, jednakże pewna ilość wersji oprogramowania może wynikać już ze specyfikacji wymagań (np. przez określenie pewnej ilości docelowych platform sprzętowych czy systemowych). Każda z takich wersji może wymagać innych wartości parametrów określających jej 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 powinny pozwalać 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 [Somm03]:

- artefakty procesu wytwórczego podlegające zarządzaniu, a także wzorzec identyfikacji tych artefaktów
- definicję osób odpowiedzialnych za zarządzanie konfiguracją i komunikację z zespołami analityków, projektantów, architektów i innych tworzących elementy systemu
- schemat wersjonowania, pozwalający na jednoznaczne umiejscowienie artefaktu procesu wytwórczego w czasie, a także jego identyfikację w zależności od innych założonych parametrów (platforma na jaką jest przeznaczony, zamawiający klient itp.)
- formalny schemat opisujący atrybuty artefaktów takie jak miejsce danego elementu w procesie wytwórczym czy status (np. „planowany”, „implementowany”, „stabilny”)
- opis dokumentacji tworzonej w ramach zarządzania konfiguracją
- opis narzędzi wspierających zarządzanie konfiguracją
- strategię postępowania z wykorzystywanym oprogramowaniem pochodząącym od zewnętrznych dostawców

Wspomniane artefakty procesu wytwórczego podlegające zarządzaniu, odpowiednio grupowane, określa się elementami konfiguracji. Zbiory elementów konfiguracji tworzą kolejne wersje całego systemu, wśród których wyróżnia się wydania – wersje przekazywane klientom. Wydanie systemu to wykonywalny kod aplikacji, a także dotyczącego pliki konfiguracyjne, dane z których korzysta program, dokumentacja (użytkownika, techniczna, marketingowa) oraz program instalacyjny.

Dobrze 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.

10.6. „Inżynieria w obie strony” (ang. round-trip engineering) – synchronizacja kodu z projektem

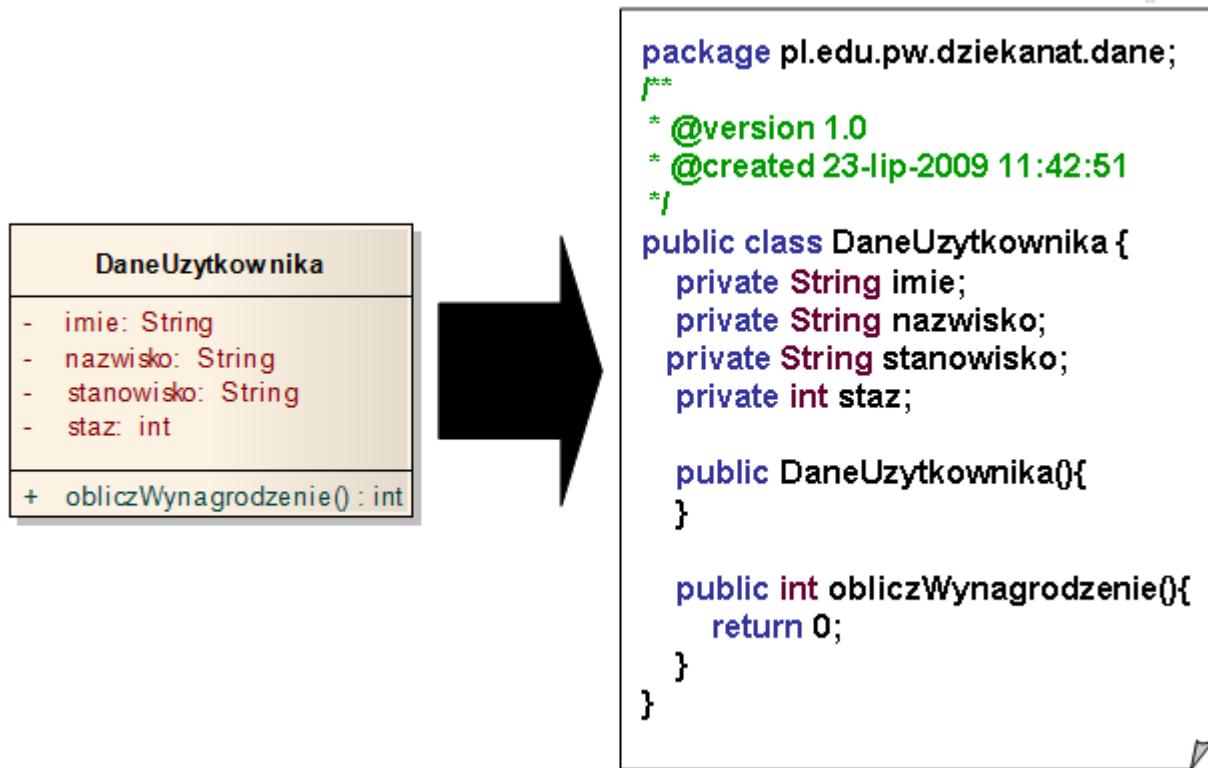
W sekcji 10.1 wspomniano, że programista powinien samodzielnie stworzyć lub otrzymać od projektantów szkielet kodu. Podstawowa struktura kodu może zostać wygenerowana przy pomocy powszechniej w narzędziach CASE funkcji generacji kodu (zwanej także „inżynierią w przód” od ang. *forward engineering*). Funkcja ta pozwala na stworzenie na podstawie modeli UML kodu w danym języku programowania. Przykładowo, dla modelu klas tworzone są katalogi odpowiadające pakietom w których znajdują się klasy oraz umiejscowione w nich pliki, wypełnione kodem odzwierciedlającym atrybuty, operacje i inne elementy dla każdej z klas z modelu. Przykład generacji kodu zilustrowany jest na Rys. 10.10.

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 ostereotypowanych czy też interpretację relacji kompozycji. *Forward engineering* dotyczy najczęściej modeli statycznych UML (diagramów klas, które transformowane są w pliki opisujące klasy lub schematy baz danych).

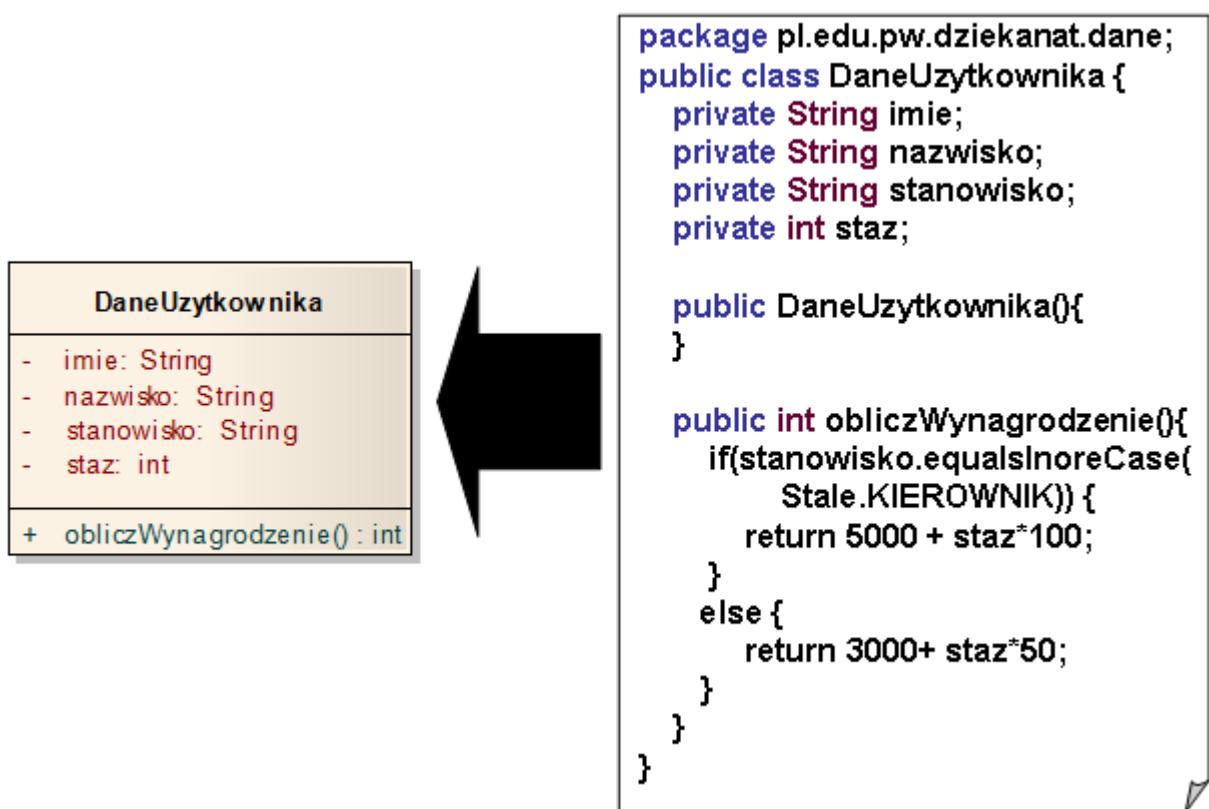
Dostępną w narzędziach CASE funkcją jest także import kodu – „inżynieria odwrotna” (ang. *reverse engineering*). Pozwala ona na wykorzystanie istniejącego kodu w celu stworzenia na jego podstawie modelu. Przykład „odtwarzania” elementu modelu UML został przedstawiony na Rys. 10.11.

Tak samo, jak „inżynieria w przód”, proces *reverse engineering* podlega dość złożonej konfiguracji umożliwiającej uzyskanie satysfakcyjnych efektów – czytelnych modeli odpowiadających znaczeniowo kodowi, z którego powstały.

Aby zachować spójność dokumentacji technicznej i kodu wykonywalnego należy, jak wspomniano w sekcji 10.2 („Dobre praktyki...”), uaktualniać model projektowy podczas kodowania. Niestety, w większości przypadków proste wygenerowanie kodu z modelu, a następnie zaktualizowanie modelu ew. zmianami w kodzie nie jest wystarczające: programy powstają przyrostowo, a także podlegają wielu zmianom wynikającym z czynników zewnętrznych (patrz: sekcja 10.4). W trakcie rozwoju produktów informatycznych pojawia się konieczność wielokrotnego generowania kodu i jednoczesnego uzupełniania modelu zmianami pochodząymi z kodu. Pewien poziom synchronizacji między kodem a modelem uzyskuje się przez zastosowanie techniki łączącej „inżynierii w przód” i „inżynierii odwrotnej” zwanej „inżynierią kodu w obie strony” (ang. *round-trip engineering*).



Rys. 10.10: Przykład generacji kodu dla klasy: źródła zapisanego w języku UML i wyniku w języku Java



Rys. 10.11: Przykład importu kodu (plik .java ⇒ klasa UML)

Większość narzędzi CASE udostępnia funkcjonalność *round-trip engineering* pozwalając na generację kodu bez nadpisywania w nim zmian jeszcze nie obecnych w modelu i, na odwrót, odtwarzając w „bezpieczny” sposób modele z kodu. Czasami w plikach z kodem znajdują się komentarzeznaczniki pozwalające oznaczyć fragmenty poddane synchronizacji z modelem.

Modele uzyskiwane technikami „inżynierii odwrotnej” i „inżynierii w obie strony” są dokładnym (na ile to możliwe) odzwierciedleniem kodu. Należy pamiętać o tym korzystając z tych modeli, ponieważ w niektórych przypadkach z uwagi na swoją zbyt dużą szczegółowość nie stanowią one dobrej „mapy” kodu. Rozwiązaniem jest tworzenie w modelu opartym na kodzie przejrzystych diagramów pomijających część informacji zgromadzonej w modelu, zorientowanych na prezentacje wybranych, istotnych aspektów implementacji.

10.7. Podsumowanie

Powyżej omówiono istotne problemy związane z przejściem ze „świata modeli” do „świata wykonywalnego kodu”. Przedstawiono sposoby w jakie można wykorzystać projekt szczegółowy – czy to zdając się na automatyczne transformacje i generatory kodu, czy też ręcznie tworząc kod aplikacji. Czynności te powinny podlegać odpowiedniej kontroli pozwalającej śledzić zmiany zachodzące w projekcie.

Przedstawione w tym rozdziale zagadnienia są elementami zarządzania jakością w projektach informatycznych. Odpowiednie zarządzanie konfiguracją, oparte o system zarządzania zmianami i kontrolę wersji wraz ze stojącym na wysokim poziomie sposobie przenoszenia wiedzy z modeli projektowych do kodu są fundamentami zapewnienia dobrej jakości produktów w ramach określonego budżetu. Kolejnym, dopełniającym wyżej wymienione zagadnieniem jest testowanie oprogramowania, opisane w następnym rozdziale.

11. Podstawy testowania

Testowanie oprogramowania jest to dział szeroko rozumianej analizy i inżynierii oprogramowania zajmujący się określaniem metod i wyników ich działania w odniesieniu do poprawności programów, algorytmów i systemów informatycznych.

Testowanie oprogramowania stało się oddzielną dziedziną sztuki programistycznej. Rozwinięto wiele metod, które pozwalają zespołom testującym nadążyć za coraz szybciej powstającymi programami, a także opanować coraz bardziej złożone interakcje między systemami informatycznymi. Te wysiłki mają spowodować podniesienie jakości wytwarzanego oprogramowania – zarówno postrzeganej z perspektywy doskonałości technicznej kodu, jak i zgodności funkcjonalności aplikacji z wymaganiami użytkownika.

Podczas powstawania systemu informatycznego niezmiernie istotne jest stwierdzenie czy jego działanie nie zwiera 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 oprogramowania⁴ i obejmują one sytuacje jak opisano w [Patt02]:

- oprogramowanie nie wykonuje czegoś, co powinno wykonywać
- oprogramowanie robi coś, czego nie powinno robić
- oprogramowanie nie wypełnia warunków związanych z szybkością działania, łatwością użycia itp.
- oprogramowanie wykonuje nieprzewidziane specyfikacją operacje

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 projektu. Podobnie jest z błędami wynikającymi z wadliwej specyfikacji. Te dwa źródła defektów oprogramowania są jeszcze niebezpieczne z powodu paradoksu: projekt realizowany według niewłaściwych założeń, mimo że w odniesieniu do nich jest poprawny, w rzeczywistości jest nie do przyjęcia.

Część błędów stanowią tzw. pomyłki – 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 możliwe wcześnie znajdowanie nieprawidłowości działania aplikacji, a także odpowiednie opisywanie ich w celu ułatwienia programistom dokonania poprawek. Także 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 system) oraz zróżnicowany (zawierać odpowiednią ilość „typowych” informacji, ale nie pomijać nawet „egzotycznych” wyjątków).

11.1. Ustalenie ilości zadań testowych

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

⁴ „Nieprawidłowość działania”, której poszukują testerzy, może być nazywana także błędem, defektem, niezgodnością ze specyfikacją (wymagań), problemem, pomyłką, incydentem, awarią, „pluską” (ang. *bug*) – wszystkie te pojęcia opisują niepożądane cechy oprogramowania, które powinny zostać odkryte w procesie testowania, a następnie poprawione lub skompensowane.

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) – Rys. 11.1.

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 umieszczonych błędów, jest oceniany poprzez odpowiednie oszacowania bazujące na liczbie wykrytych przez niego usterek.

Jeżeli:

$$B - \text{liczba wykrytych błędów}$$

$$b - \text{liczba posianych błędów, które zostały wykryte}$$

$$P - \text{liczba posianych błędów}$$

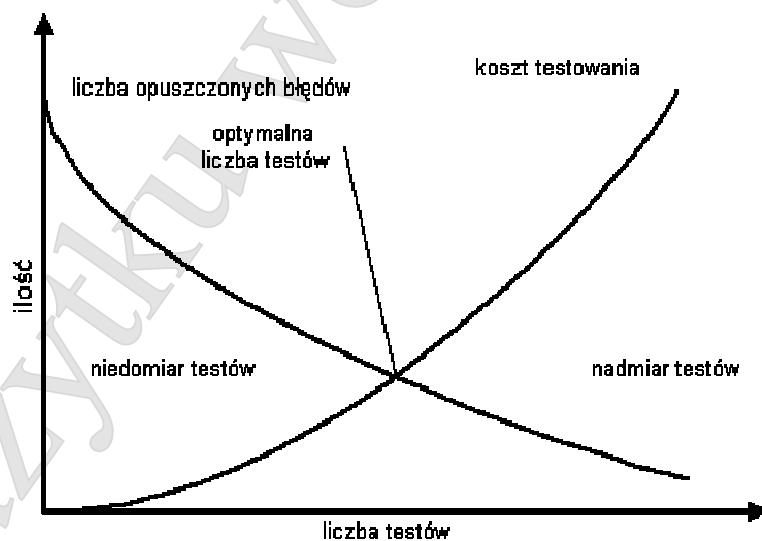
to szacunkowa liczba błędów przed wykonaniem testów wynosi

$$(B - b) * \frac{P}{b} \quad (11-1)$$

a szacunkowa liczba błędów pozostałych po wykonaniu testów

$$(B - b) * \frac{P}{b-1} \quad (11-2)$$

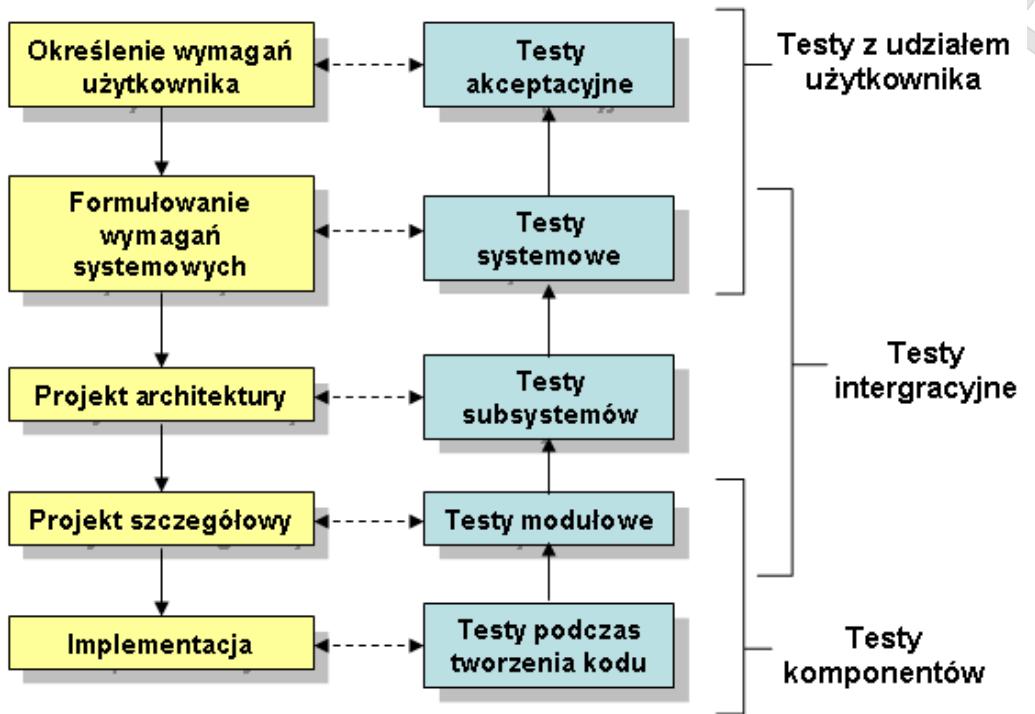
Mimo że walidacja oprogramowania często dotyczy procesów zewnętrznych względem głównego „pnia” rozwoju projektu (kontrole, audyty, przeglądy, standaryzacja – dotyczy to głównie projektów rządowych i militarnych), to testowanie jest wpisane w cykl wytwarzczy oprogramowania (patrz: sekcja 11.2).



Rys. 11.1: Ustalenie odpowiedniej ilości zadań testowych

11.2. Różne poziomy testowania systemu: od testów modułowych do testów akceptacyjnych

Podczas przygotowywania systemów informatycznych znaczną część czasu (czasem nawet do 80%) poświęca się testom powstającego produktu. Testy towarzyszą tworzeniu programu we wszystkich fazach jego rozwoju – Rys. 11.2.



Rys. 11.2: Różne poziomy testowania systemu

Wśród faz procesu testowania możemy wyróżnić (klasyfikacja wg. [Somm03]):

- testowanie jednostek (ang. *unit testing*, testy podczas implementacji): testowaniu podlegają poszczególne klasy, ew. ich zbiory. Drobne elementy systemu testowane są niezależnie od innych (w przypadku potrzeby komunikacji testowanego kodu z innymi komponentami należy stosować „zaślepki” – obiekty imitujące ich zachowanie)
- testowanie modułowe: moduły (komponenty) to wynikające z modelu projektowego zbiory jednostek takich jak klasy; w tej fazie testowane są niezależnie od działania i testów innych modułów
- testowanie subsystemów: testowaniu podlegają zbiory komponentów, które wg. projektu architektury podlegają integracji w większe całości. W tej fazie nacisk kładzie się na testowanie przeprowadzanej przez interfejsy komunikacji między modułami.
- testowanie systemu: faza mająca zidentyfikować problemy dotyczące współpracy zintegrowanych podsystemów, a także zachowania systemu jako całości w odniesieniu do wymagań nie-funkcjonalnych (bada się m.in. jego wydajność i skalowalność)
- testowanie akceptacyjne (odbiorcze): w ostatniej fazie testowania określa się na ile system spełnia stawiane mu wymagania funkcjonalne. Testy akceptacyjne budowane są w oparciu o scenariusze przypadków użycia (patrz: sekcja 11.4), dane testowe dostarczone przez użytkownika oraz przy udziale przedstawicieli sponsora projektu (użytkowników końcowych).

Wartym podkreślenia jest fakt, że w kolejnych fazach testowania biorą udział różne osoby. Efekty implementacji na najniższym poziomie testują programiści (weryfikując tym samym własny kod). Biorą oni także udział w testach integracyjnych, jednak stopniowo (wraz z wzrostem granularności testowanych elementów) odpowiedzialność za proces ten przejmują testerzy. Opis ról występujących w procesie testowania znajduje się w sekcji 11.6.

Jak już wspomniano ostatnia faza testowania przebiega przy udziale użytkowników końcowych systemu, którzy często uzupełniani są osobami określonymi jak „niedoświadczeni użytkownicy”: osoby niepowiązane z klientem i wydawcą oprogramowania 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”. Inne metody testowania przedstawiono w sekcji kolejnej.

11.3. Podstawowe metody testowania

W czasie wczesnego rozwoju inżynierii oprogramowania rozwijano teoretyczne metody testowania (na przykład formalne dowodzenie poprawności algorytmów Djiksty), jednak nie sprawdzają się one w warunkach zasobów pieniężnych i czasowych jakimi dysponują współczesne zespoły programistyczne. Popularne obecnie techniki testowania można podzielić na metody nieinwazyjne („czarnej skrzynki” od ang. *black box*), metody związane z analizą powstającego kodu („szklana skrzynka” - *glass box*) lub specyfikacji (testy statyczne). Wymienione podejścia do testowania, niezależnie od ich rodzaju, można przeprowadzać na wielu poziomach abstrakcji. Poniżej przedstawiono krótki opis podstawowych metod testowania.

Metoda „czarnej skrzynki” polega 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 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 oczekiwani użytkownika, niż z perspektywy zawiłości technicznych pewnych rozwiązań. Testerzy nie koncentrują się na tym jak program działa, ale czy działa. Nie ma też mowy 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 „programistom się nie śniło”, 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 idei nieingerencji i nieznajomości kodu programu. Należą do nich metody:

- tworzenia klas równoważności
- warunków granicznych
- zmian stanów

- „niedoswiadczonego” 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, zeroowych, początkowych i innych przypadków, gdy przyszły użytkownik nie wprowadzi do programu oczekiwanych danych. Inną klasą równoważności mogą stanowić wartości z 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 dilemma 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:

$$\begin{aligned}
 x &= a - 1 \\
 x &= a \\
 x &= a + 1 && (11-3) \\
 x &= b - 1 \\
 x &= b \\
 x &= b + 1
 \end{aligned}$$

Jeśli program odpowiednio reaguje na takie dane, to najprawdopodobniej (o ile przedział został wyznaczony poprawnie) będzie działał prawidłowo również dla danych ze środka przedziału. Właśnie 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. Z pośród 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 określone 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 $(0, 1, 7, 15, 127, 255, 511, 1023, \dots)$ 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 ilość 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, przekroczyć 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, można odkryć wiele nieoczekiwanych błędów.

acjach (które przecież mogą zaistnieć podczas rzeczywistego użytkowania) można uzyskać nową wiedzę na temat pozostawionych przez programistów „niedociągnięć”.

Metoda testowania zmian stanów szczególnie dobrze wpisuje się w ideę testowania „czarnej skrzynki”, ponieważ na przejście między trybami działania programu najlepiej jest spojrzeć z punktu widzenia użytkownika, bez wnikania w wartości zmienny czy używane funkcje. Także z perspektywy odbiorcy oprogramowania (dość specyficznego) pozwala spojrzeć na błędy w programie metoda „niedoświadczonego użytkownika”, omówiona w poprzedniej sekcji.

Metoda „czarnej skrzynki” jest przeciwniągą dla **metody „szklanej (białej) skrzynki”** (ang. *glass-, white-box testing*) wykorzystującej wiedzę o kodzie programu w celu podniesienia skuteczności testów. Testy bazujące na oglądzie 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 (patrz sekcja 10.2) 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).

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.

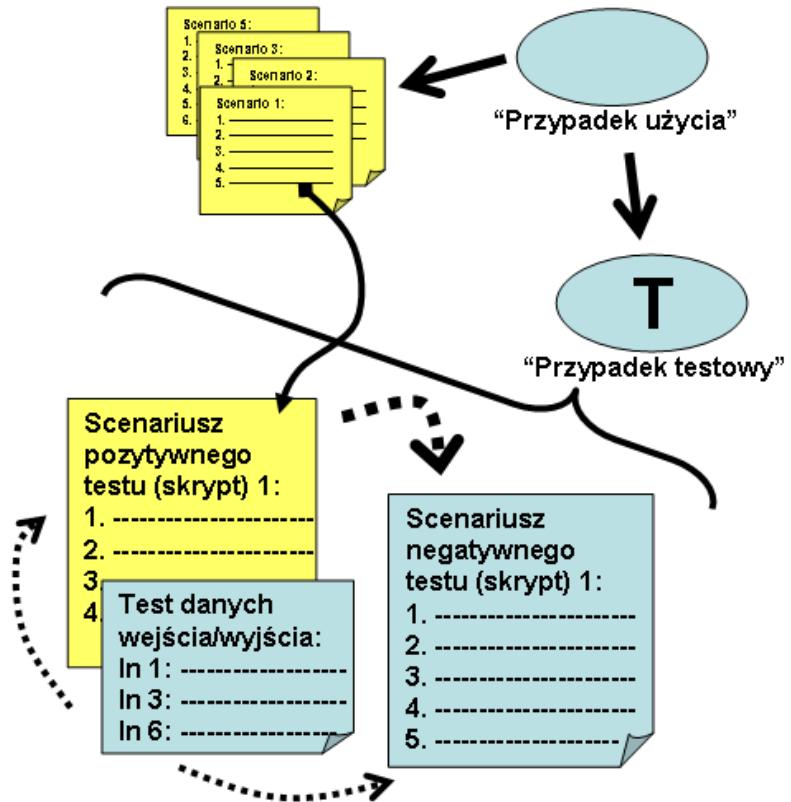
11.4. Budowa scenariuszy testowych na podstawie przypadków użycia systemu

Szczegółowa specyfikacja wymagań, zawierająca przypadki użycia opisane scenariuszami, pozwala na łatwe utworzenie wartościowych testów akceptacyjnych. Bazują one na odpowiadających przypadkom użycia przypadkach testowych (ang. *test case*), na które składają się sekwencje zdarzeń systemowych oraz przypisane do nich dane testowe.

Jak wspomniano w rozdziale 7, przypadki użycia najczęściej opisuje się przez tzw. scenariusz podstawowy (przedstawiający interakcję aktor – system zakończoną sukcesem, czyli realizacją celu przypadku użycia) oraz scenariusze alternatywne (prezentujące nie zawsze osiągające cel wersje scenariusza podstawowego). Wykorzystanie tych scenariuszy w testach akceptacyjnych polega na przekształceniu ich na scenariusze testowe. Odbywa się to, co ilustruje Rys. 11.3, poprzez przełożenie kolejnych zdań przypadków użycia na kroki wykonywane w ramach przypadków testowych. Kroki scenariuszy opisujące zachowania aktorów odpowiadają danym wprowadzanym w odpowiednich momentach przeprowadzania testów. Wynikami procesu testowania są reakcje systemu zachodzące pod wpływem takich sygnałów. „Pozytywne” zakończone scenariusze testu sprawdzają pożąданie działania systemu, a „negatywne” pozwalają określić reakcje na niewłaściwe dane wejściowe.

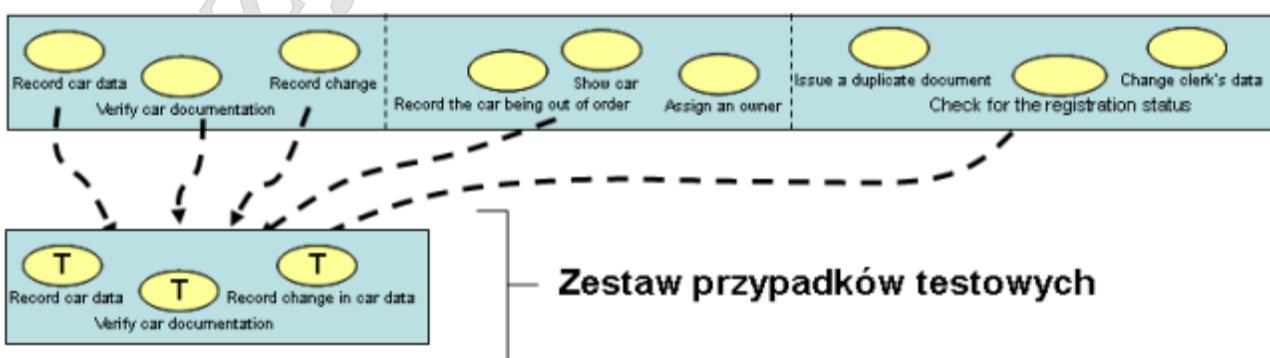
Zorganizowanie testów akceptacyjnych w scenariusze składające się ze ścisłe określonych kroków-zdarzeń wraz z przypisanymi im danymi pozwala na tworzenie automatycznych skryptów testowych. Przygotowanie tego typu testów automatycznych, mimo wymaganego pewnego nakła-

du pracy, pozwala na dokładne i wielokrotne przetestowanie systemu. Dużą zaletą posiadania zbioru automatycznych testów funkcjonalnych jest możliwość badania systemu po wprowadzeniu zmian (patrz: sekcja 10.4), w celu ustalenia czy dokonane modyfikacje nie wprowadziły błędów do zweryfikowanych już kiedyś części systemu (są to tzw. testy regresyjne). Automatyzacja procesu testowania jest wspierana przez szereg narzędzi opisanych w sekcji 12.6.



Rys. 11.3: Budowa scenariuszy testowych na podstawie przypadków użycia systemu

Każdy przypadek testowy jest wykonywany w izolacji. Inne elementy systemu lub systemy zewnętrzne są reprezentowane jako obiekty-atrapy, metody-zaślepki itp. Testy jednostkowe mogą być grupowane w kolekcje testów (ang. *test suite*). Kolekcje te mogą być kombinacjami testów w zależności od wybranego kryterium (np. testujące przypadki dotyczące wybranego aktora albo zagadnienia w obrębie dziedziny systemu). Istotnym zagadnieniem jest odpowiednie tworzenie zestawów przypadków testowych w oparciu o wymagania przypisane kolejnym iteracjom. Zależnie od przyjętej strategii testowania takie zestawy obejmują przypadki testowe weryfikujące funkcjonalność „aktualnej” iteracji oraz zbiór przypadków testowych poprzednich iteracji. Zbiór dla poprzednich iteracji może obejmować wszystkie już weryfikowane przypadki użycia lub ich podzbiór określany w zależności od wpływu wymagań aktualnie implementowanych na wymagania poprzednich iteracji.



Rys. 11.4: Tworzenie zestawów przypadków testowych

Wzorzec wykonania przypadków testowych obejmuje następujące czynności:

- czynności wykonywane przed procedurą testującą (przygotowanie środowiska testowego, np. wypełnienie bazy danych danymi testowymi)
- procedura testująca (odtworzenie kroków scenariusza): niezgodność rezultatów testów i wzorcowego ich przebiegu może być badana w każdym kroku scenariuszy testowych
- porządkowanie (przywrócenie środowiska do stanu pierwotnego)

Zautomatyzowane przypadki testowe pozwalają także na badanie pewnych cech pozafunkcjonalnych systemu.

11.5. Testowanie cech pozafunkcjonalnych systemu

Weryfikacja cech pozafunkcjonalnych systemu jest złożonym problemem. Wymagania opisujące charakterystykę pozafunkcjonalną systemu (patrz: sekcja 7.2.2) nie mają tak precyzyjnej formy jak określające dokładnie, krok po kroku działanie aplikacji scenariusze przypadków użycia. Dodatkowo jakość systemu postrzegana z perspektywy cech niefunkcjonalnych ma wiele aspektów, z których każdy podlega innym miarom i metodom testowania.

Dwa podstawowe ujęcia cech pozafunkcjonalnych to ich analiza z punktu widzenia wykonywania programu oraz z punktu widzenia ewolucji systemu. Przy pierwszym podejściu wymienia się najczęściej efektywność (ang. *efficiency*), użyteczność (ang. *usability*), bezpieczeństwo (ang. *security*) oraz przenośność (ang. *portability*). Druga perspektywa identyfikuje parametry związane z utrzymaniem systemu, takie jak koszt zmian, skalowalność (ang. *scalability*), testowalność (ang. *testability*), łatwość analizy (ang. *analyzability*), stabilność (ang. *stability*) czy dojrzałość (ang. *maturity*)⁵.

Zasadniczo łatwiejszymi do pomiarów testowych są cechy odnoszące się do dynamiki systemu. Efektywność, będącą miarą związku między zasobami (mocą obliczeniową) a wynikami szybkości działania osiąganymi przez program łatwo wyrazić liczbowo mierząc czas wykonania pewnych operacji, czas odpowiedzi przy założeniu pewnych sygnałów itd. Proces takiej weryfikacji można łatwo zautomatyzować np. „nagrywając” ścieżki działania użytkowników i odtwarzając je następnie w odpowiednio zwielokrotnionej liczbie, dokonując pomiarów i zwiększając obciążenie aż do osiągnięcia granic wydajności (jest to tzw. testowanie obciążeniowe). Testowanie tego typu pozwala ocenić nie tylko czasy przetwarzania i odpowiedzi przy pewnych obciążeniach, ale także sposób degradacji systemu w skrajnych warunkach.

Użyteczność systemu to cecha opisująca wysiłek jaki użytkownicy systemu wkładają w korzystanie z jego funkcjonalności. Użyteczność podlega najczęściej weryfikacji poprzez ocenę użytkowników systemu oraz ekspertów ergonomii systemów informatycznych dotyczącą kryteriów takich jak (klasyfikacja za [Patt02]):

- zgodność ze standarami i regułami komunikacji człowiek-komputer: spełnienie założeń interfejsów (nie tylko użytkownika!) dla danej platformy, systemu, środowiska itp.
- intuicyjność: odpowiednia ilość informacji prezentowanych jednocześnie użytkownikowi, właściwe rozplanowanie konkretnych „ekranów”
- spójność: jednolity wygląd i zachowanie różnych części interfejsu, przemyślane przepływy między ekranami aplikacji, spójna (także z np. systemem operacyjnym pod jakiego kontrolą pracuje aplikacja) terminologia, skróty klawiszowe, układ menu itd.
- elastyczność: możliwość dostosowania aplikacji do sposobu pracy różnych osób
- wygoda: łatwość dostępu do pewnych funkcji programu (powinna być tym wyższa, im częściej używana funkcja)

⁵ Prezentowane tutaj nazewnictwo i ujęcie cech jakościowych oprogramowania pochodzi z normy ISO 9126

- poprawność: zgodność oczekiwanej zachowania się programu pod wpływem inicjowania danych operacji w jego interfejsie

Część z ww. kryteriów można przedstawić liczbowo (np. przez ilość kliknięć myszą wymaganą by uruchomić daną funkcję programu), jednak miary użyteczności uchodzą za dość subiektywne.

Istotnym jest, by pamiętać, że użyteczność nie opisuje (choć jest tak najczęściej) jedynie graficznych lub tekstowych interfejsów użytkownika, ale także interfejsy, np. pozwalające na wymianę danych z systemami zewnętrznymi.

Testowanie bezpieczeństwa systemów często dotyczy testów całych środowisk uruchomieniowych, ponieważ zagadnienia związane z dostępnością do aplikacji i ochroną danych ściśle powiązane są ze sprzętem i aplikacjami z jakich korzystają systemy. Testy polegają na weryfikacji czy operacje dotyczące danych spełniają określone standardy i założenia, a także na badaniu ścieżek przepływu komunikacji aktorzy-system (pod kątem np. przekroczenia uprawnień). Przetwarzanie danych przez system należy ocenić ze względu na zachowanie ich integralności, poufność przesyłu i wymiany, poziomy dostępu, czy sposób dziennikowania⁶ zmian. Częstą techniką jest przekazywanie części budżetu przeznaczonego na testowanie specjalistycznej firmie, która w danych czasie, przy danych zasobach próbuje „złamać” system.

Rozmiar przedsięwzięcia testowania konfiguracji jest często ogromny (wystarczy sobie wyobrazić jaką liczbę drukarek i ich konfiguracji musi obsługiwać zwykły edytor tekstu). Dlatego przenośność systemu, a także jego zgodność z różnymi konfiguracjami i kompatybilność z założonymi środowiskami, w jakich ma pracować, bada się także sprawdzając zgodność zewnętrznych interfejsów systemu ze standardami. Duże możliwości daje też wirtualizacja systemów operacyjnych i sprzętu – korzystając ze środowisk-emulatorów można zbadać działanie systemu na różnych platformach.

Testowanie cech programów związanych z ich utrzymaniem łączy się ściśle z zarządzaniem zmianami w obrębie danego projektu (patrz: sekcja 10.4). Estymację wyceny przyszłych poprawek, czy usprawnień można często oprzeć jedynie na analizie kosztów przeszłych zmian w systemie. Także od liczby przeszłych wydań i czasu trwania projektu można uzależnić miarę stabilności czy dojrzałości aplikacji. Ścisłe mierzalna jest testowalność oprogramowania – badanie jej polega ona na ocenie liczby punktów dostępu do systemu, tzn. czy indywidualne komponenty można zasilać danymi testowymi, czy też komunikacja z aplikacją odbywa się jedynie poprzez interfejs użytkownika.

11.6. Organizacja procesu testowania systemu

Organizacja procesu testowania systemu częściowo zależy od przyjętego w danym projekcie modelu twórczego oprogramowania (patrz: sekcja 2.3) i metodyki (patrz: rozdział 3). Występująca w każdym popularnym modelu faza testowania, mająca miejsce najczęściej po fazie implementacji, powinna być utożsamiona z opisanymi powyżej testami akceptacyjnymi. Testowanie aplikacji na pozostałych poziomach (patrz sekcja 11.1) powinno odbywać się już podczas fazy implementacji w dwóch niezależnych procesach: wstępującym i zstępującym.

Podejście wstępujące wymaga, aby osoby zaangażowane w prace nad kodem na bieżąco badały powstający (pisany przez siebie) kod. Drobne fragmenty aplikacji (procedury i funkcje oraz klasy) testowane są przez programistów metodami „szklanej skrzynki” w izolacji od reszty systemu. Każdy z koderów i każdy z zespołów programistów odpowiedzialny jest za bezawaryjność „swoich” elementów, których współdziałanie z innymi weryfikowane jest podczas równolegle przebiegającego procesu testowania zstępującego.

⁶ Dziennikowanie (ang. *logging*) to sposób monitorowania operacji na danych i metadanych, polegający na zapisie parametrów dostępu do danych. Parametry takie obejmują najczęściej osobę lub proces posiadający kontakt z danymi, znacznik czasowy wykonania operacji, czy jej charakter – zapis, odczyt, uaktualnienie. W literaturze polskojęzycznej dziennikowanie bywa mylone z kronikowaniem (ang. *journaling*) – zagadnieniem związanym z opóźnionym zapisem operacji na danych (zmiany, zanim zostaną utrwalone w pamięci stałej, są zapisywane w *kronice*).

Testowanie wg. podejścia zstępującego wymaga stworzenia w obrębie projektu odpowiedniej infrastruktury technicznej pozwalającej na ciągłą integrację. Infrastruktura taka powinna obejmować narzędzie automatycznej integracji, centralne repozytorium kodu oraz automatyczne skrypty testów. Zależnie od przyjętego planu integracji (integracja godzinna, dzienna, tygodniowa) osoby kierujące projektem, korzystając z techniki *build and smoke* mogą ocenić, które elementy powstającego systemu nie funkcjonują jeszcze prawidłowo, co daje dobry przegląd ogólnego postępu prac implementacyjnych.

Niezależnie od wybranego rodzaju procesu testowania wymogiem jest, aby automatyczne testy jednostkowe powstawały najpóźniej jednocześnie z kodem. Warunek ten ma przesłanki techniczne: pozwala na wcześnie wykrycie usterek, ponadto testy niejako wyznaczają kierunek w jakim rozwijać ma się kod i „ustalają” jego prawdziwą funkcjonalność. Powstanie dobrych testów zanim zacznie się implementacja jest także motywacją dla zespołu programistów, którzy mają wymierne, aktualne wskaźniki swojej pracy („patrzcie, ile już działa”) oraz znają ich kierunek („do zaimplementowania zostało jeszcze to i to”). Podejście ortodoksyjne, zakładające postawianie kodu jedynie po stworzeniu testów lub nawet jedynie po zakończonym błędem uruchomieniu testów nazywa się „procesem sterowanym testami” (ang. *Test-Driven Development*, TDD).

Osoby (role) uczestniczące w procesie testowania są określane przez metodykę całego procesu twórczego. Charakterystyczne dla procedur weryfikacji systemu role to:

- projektant testów – osoba odpowiedzialna za tworzenie planów testów i specyfikacji testów w oparciu o wymagania systemu i określone dokumenty techniczne
- koordynator testów – osoba nadzorująca sposób przeprowadzania testów i ich wyniki; rolą ta często łączona jest z rolą projektanta testów i rolami związanymi z zarządzaniem projektem na poziomie technicznym
- tester – wykonuje testy i dokumentuje rezultaty

Organizacja procesu testowania jest zagadnieniem przenikającym proces tworzenia oprogramowania na wielu poziomach i wymaga współdziałania wielu osób o różnych kompetencjach. Odpowiednie zaplanowanie testów jest warunkiem zapewnienia wysokiej jakości wytwarzanego oprogramowania.

12. Narzędzia 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 itp. Aby praca informatyków była nie tylko wydajna i zakończona sukcesami, ale w ogóle możliwa, niezbędne jest powstanie odpowiedniego środowiska, w którym te tworzone artefakty są właściwie katalogowane. Takie środowisko pracy powinno pozwalać nie tylko na łatwą wymianę informacji między uczestnikami projektu, ale również automatyzować możliwie wiele pracochłonnych czynności na wszystkich etapach procesu twórczego.

W celu konstrukcji środowisk pracy informatyków powstają narzędzia typu CASE (ang. *Computer-Aided Software Engineering* – Inżynieria oprogramowania wspomagana komputerowo) wspierające praktyczne zastosowanie inżynierii oprogramowania. 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. Kolejne sekcje omawiają szczegółowo dziedziny, w których proces twórczy może być usprawniony przez zastosowanie narzędzi.

12.1. Omówienie narzędzi wspomagających modelowanie obiektowe

Narzędzia wspierające modelowanie obiektowe należą do najważniejszych jakie zespół projektowy posiada w swoim „warsztacie” inżynierii oprogramowania. Znaczenie modelowania jest o tyle istotne, że ma ono miejsce w czasie wszystkich właściwie faz rozwoju produktów. Modelowanie wspierane narzędziami może być odpowiednio automatyzowane. Pozwalające na łatwą wymianę informacji, dodatkowo umożliwia przetwarzanie modeli przez maszyny, w celu ich walidacji, analizy czy przekształceń.

Podstawową funkcjonalnością typowego narzędzia służącego do modelowania obiektowego jest wsparcie dla rysowania diagramów w wybranej notacji. Wsparcie to polega na udostępnieniu użytkownikowi możliwości łatwego umieszczania elementów na diagramach (poprzez np. tzw. przybornik), edycji diagramów oraz przeglądania modeli (Rys. 12.1 – po lewej stronie widoczny przybornik, w części środkowej edytowany diagram, po prawej podgląd struktury modelu). Wsparcie narzędzi dla notacji polega na łatwym i przejrzystym tworzeniu diagramów różnych typów. Dobre narzędzia „pilnują” poprawności notacji nie pozwalając użytkownikowi na umieszczenie na diagramie danego typu niewłaściwych elementów czy nieprawidłowe zapisanie konstrukcji języka.

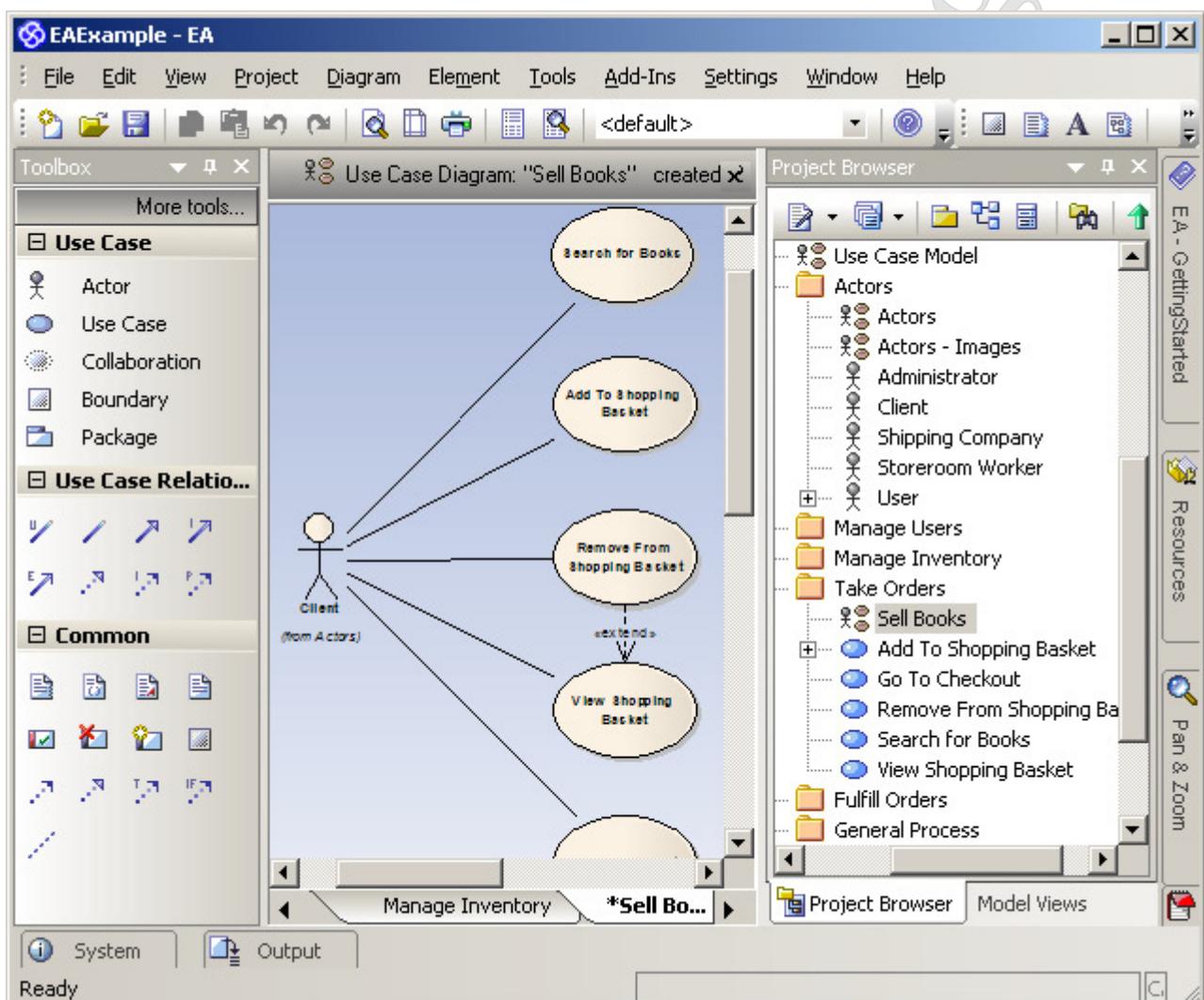
Istotną funkcją narzędzia modelowania jest także umożliwienie zarządzania modelami: odpowiedniego porządkowania ich struktury (np. w formie pakietów), a także wygodnego przechodzenia między poziomami abstrakcji i śledzenia zależności między fragmentami systemów opisanych z różnym stopniem szczegółowości. 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.

Zapis wiedzy związanej z modelowaniem systemów może odbywać się (a nawet często odbywa się!) w dość prymitywny sposób – do modelowania wystarczy przecież parę kolorowych pisaków i papier. Jednak wyspecjalizowane narzędzia modelowania nie dość, że upraszczają ten proces od strony czysto technicznej (np. łatwiej jest nanosić poprawki na diagramach), to także udostępniają szereg innych przydatnych funkcjonalności.

Jedną z nich jest inżynieria kodu (opisana w sekcji 10.6). Korzystając z funkcji z nią związanych można generować szkielet kodu odzwierciedlający wiedzę zawartą w modelu (tzw. „inżynieria w przód”), tworzyć modele w celu zrozumienia istniejącego kodu („inżynieria odwrotna”), a także wspierać proces jednoczesnego pisania kodu i aktualniania modelu („inżynieria w obie strony”).

Narzędzia najczęściej pozwalają na szeroką parametryzację tych procesów – nie tylko przez określenie „docelowego” języka programowania, ale sposobu interpretacji konstrukcji języków modelowania i opisujących kod.

Ważnym zagadnieniem związanym z tworzeniem modeli jest udostępnianie efektów pracy pozostały osobom uczestniczącym w projekcie w dogodnej dla nich formie. Większość narzędzi pozwala na generację dokumentacji modeli. Jest to, podobnie jak inżynieria kodu, proces parametryzowany: dostępna jest mnogość formatów docelowych dokumentów (statyczny lub dynamiczny HTML, pliki RTF, pliki graficzne itd.). Możliwa jest także konfiguracja zakresu generowanej treści, często istnieje też wsparcie dla tworzenia szablonów dokumentów. Inny sposób wymiany modeli to poddanie ich kontroli wersji (której podstawowe mechanizmy opisano w sekcji 10.3). Umożliwia to pracę grupową nad modelami. Czasami zachodzi także potrzeba wymiany modeli między zespołami korzystającymi z różnych zestawów narzędzi modelowania – służy temu ustandaryzowany⁷ format XMI (ang. *XML Metadata Interchange*).



Rys. 12.1: Tworzenie modelu w narzędziu CASE (tu: model UML tworzony w Sparx Systems Enterprise Architect).

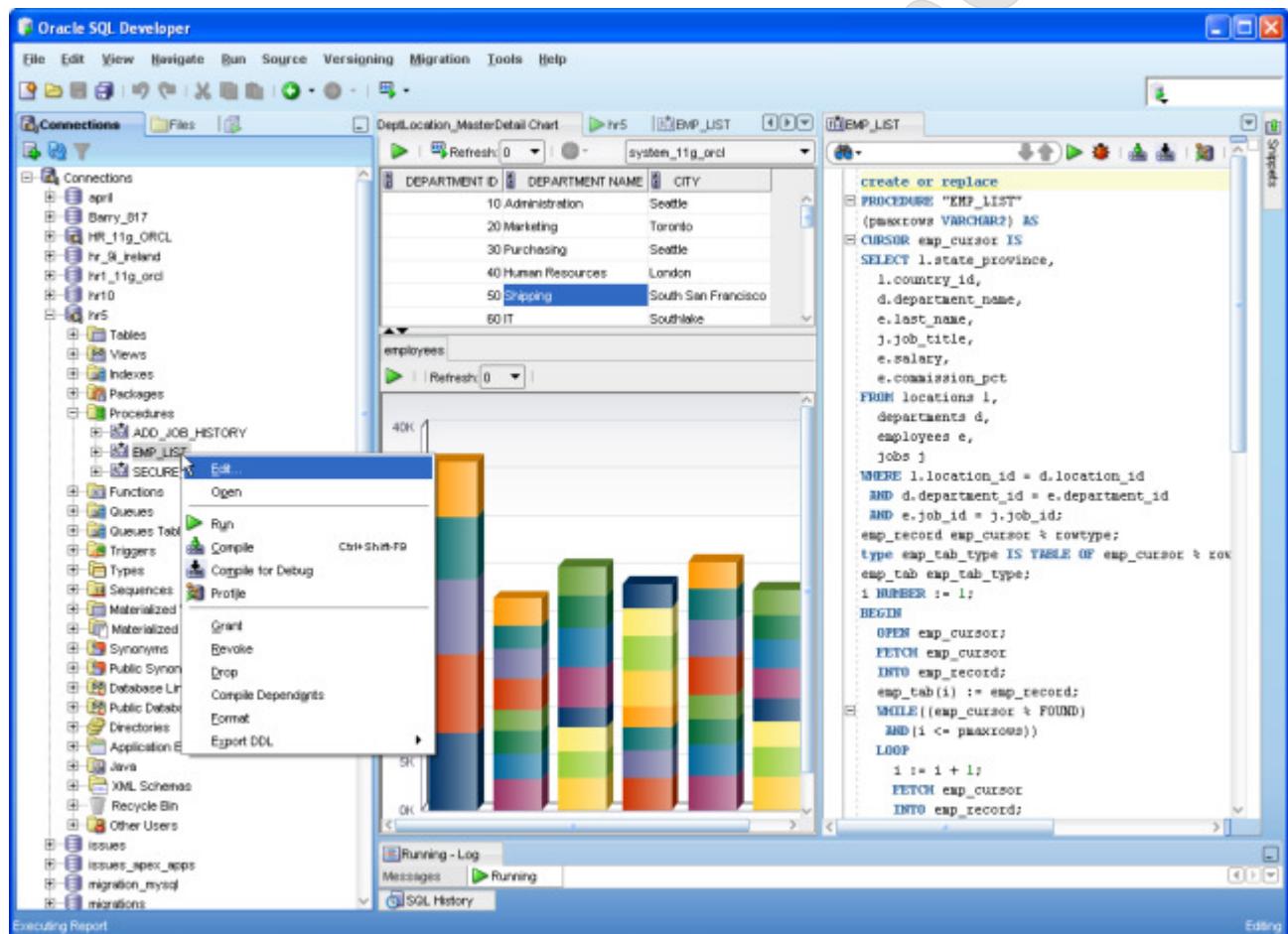
Zaawansowane narzędzia modelowania pozwalają także przypisać elementom zapisanym w wybranym języku modelowania (np. UML) pewne atrybuty związane z zarządzaniem projektami, takie jak ryzyko związane z danym elementem, osobę za niego odpowiedzialną, fazę prac, wersję

⁷ Niestety współpraca narzędzi CASE poprzez wymianę plików XMI w praktyce okazuje się dużym problemem: wytwórcy oprogramowania odmiennie interpretują i korzystają ze standardu, także mnogość wersji języka UML i różne sposoby jego reprezentacji wewnętrznej w narzędziach powodują częsty brak kompatybilności wymienianych plików.

itp. Atrybuty te pozwalają one na łatwą analizę pewnych zagadnień związanych z zarządzaniem projektami, śledzenie zmian, kalkulację ryzyka, czy obliczanie miar w projekcie.

12.2. Narzędzia do zarządzania bazami danych

Szczególnym rodzajem narzędzi modelowania są aplikacje wspierające modelowanie danych i zarządzanie bazami danych. O ile samo modelowanie danych odbywać się może w sposób podobny do modelowania obiektowego (nawet przy wykorzystaniu tej samej notacji), to aplikacje CASE związane z bazami danych ukierunkowane są często na fizyczne oddziaływanie na schematy bazodanowe. Polegać ono może na wsparciu procesu utrzymania baz danych: narzędzia pozwalają na wygodne tworzenie kopii zapasowych, importowanie lub eksportowanie danych, kreację raportów dotyczących danych i ich schematów, opcje wydajności i „strojenia” (ang. *tunning*) bazy. Niektóre posiadają także funkcje IDE, np. uruchamianie poleceń administracyjnych albo związanych z danymi czy debugowanie (z ang. *debugging*) przetwarzanych przez silnik bazy skryptów. Często dostępne w tego typu narzędziach są opcje analizy danych pozwalające na „wizualne” tworzenie zapytań.



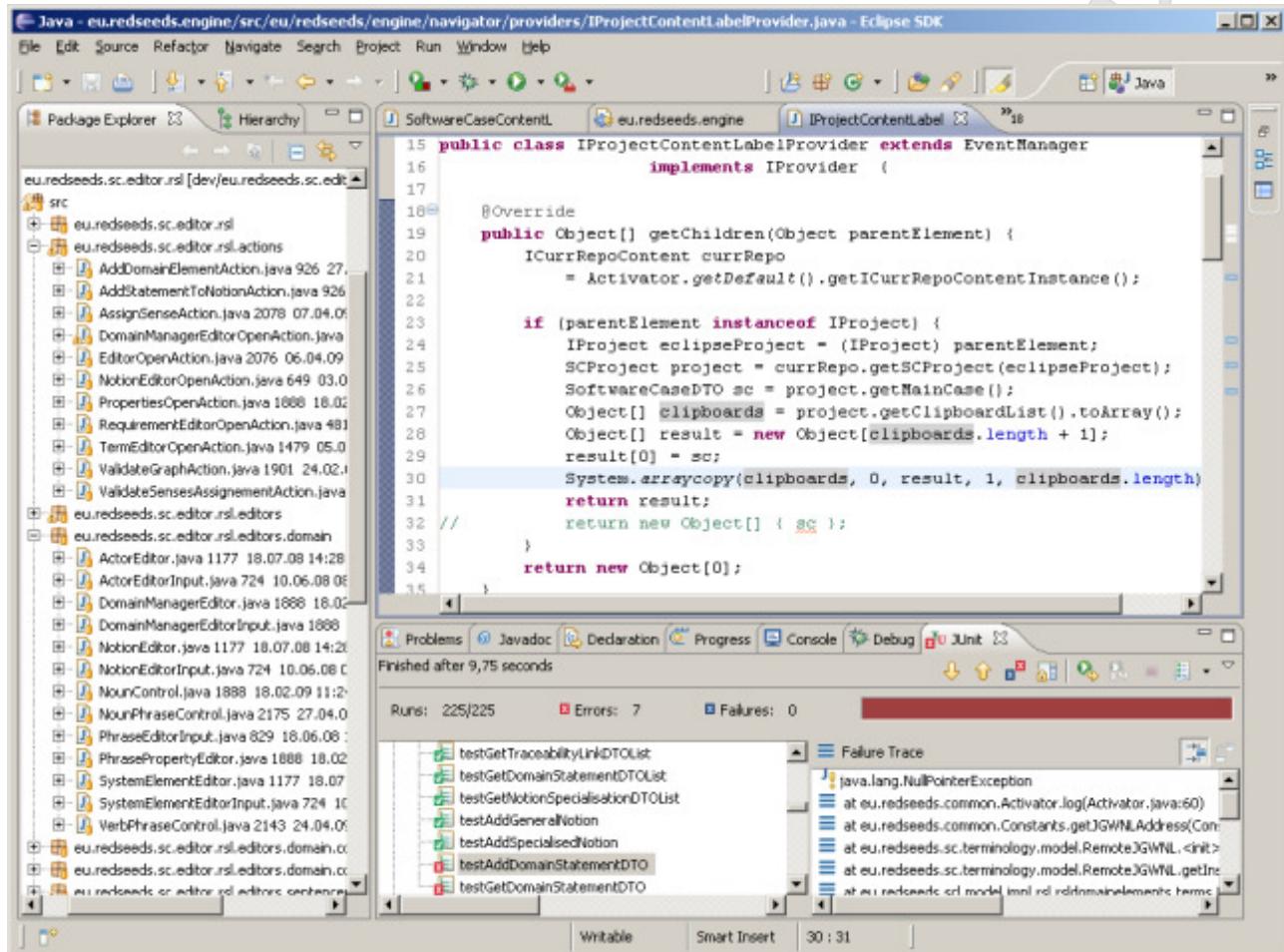
Rys. 12.2: Widoki w obrębie programu Oracle® SQL Developer. Obraz pochodzi z materiałów producenta.

Rys. 12.2 przedstawia widok narzędzia wspierającego współpracę z bazami danych. Umożliwia ono przeglądanie struktury instancji baz danych (lewy panel), specyfikacji konkretnej tabeli (środek-góra), raportu (środek-dół) oraz pisanie kodu (prawy panel).

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 (np. edycji typu *master – detail* czy słownikowanych list wartości).

12.3. Narzędzia wspierające programowanie

Narzędzia wspierające tworzenie kodu to bardzo rozwinięta klasa narzędzi CASE. Podstawowym zadaniem takich narzędzi jest zwiększenie produktywności programistów przez funkcje ułatwiające tworzenie i konserwowanie kodu. Te z nich, które integrują wiele funkcjonalności, poza typowym edytorem tekstu-kodu, nazywa się zintegrowanymi środowiskami programistycznymi (IDE).



Rys. 12.3: Widok IDE Eclipse

W ciągu ostatnich lat same edytory kodu zostały znaczco rozwinięte: do typowego dla nich „kolorowania” kodu, ułatwień formatowania, czy autouzupełniania (ang. *autocompletion*) i innych funkcji „edytorskich”, dodano możliwości pozwalające 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 z pośród 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 (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.

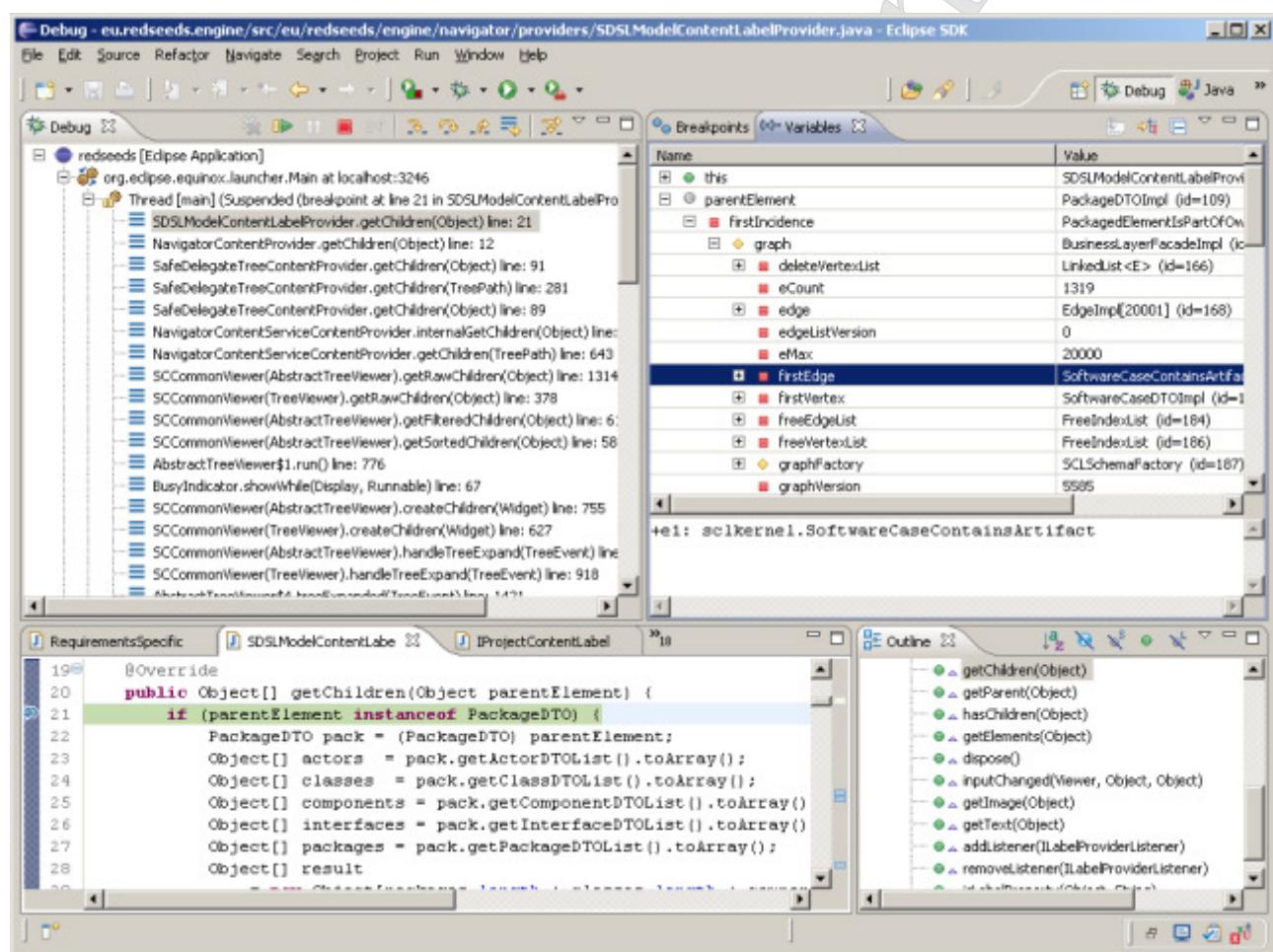
Prawdziwa siła wsparcia, jakie IDE udzielają programiście leży jednak w tym, w jaki sposób łączą się one z narzędziami zewnętrznymi. Scisła integracja z kompilatorami pozwala na wskazywanie (także tylko potencjalnych) błędów w kodzie, a często nawet automatyczne ich poprawianie. Popularne narzędzia modelowania 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. Środowiska programistyczne ułatwiają także tworzenie środowisk testowych przez automatyczną generację szkieletu skryptów weryfikujących aplikację oraz możliwość analizy i kontroli uruchomienia już zaimplementowanych testów.

Rys. 12.3 Rys. 12.4 ilustruje przykładowe środowisko zintegrowane (tu: Eclipse): na lewym panelu widoczna przeglądarka hierarchii kodu, ma górnym środkowym panelu edytor kodu, na dolnym środkowym panelu efekt działania wtyczki testów jednostkowych, widoczne także zakładki dla wielu innych narzędzi

Ważnym narzędziem w obrębie IDE jest tzw. debugger („odpluskwiacz”) pozwalający na analizę programów podczas ich wykonywania. Uruchomiony w ramach debugera proces może być zatrzymywany w czasie wykonywania wybranych przez programistę instrukcji czy nawet wykonywany krokowo. Debugger pozwala na podgląd stanu pamięci przypisanej do debugowanego programu. Sam proces „odpluskwiacza” pozwala na rozwiązywanie problemów, których przezwyciężenie tradycyjnymi metodami (np. poprzez analizę kodu) jest bardzo pracochłonne.

Na Rys. 12.4 przedstawiono debugger uruchomiony w środowisku Eclipse – widoczny jest podgląd kodu z zaznaczoną aktualnie przetwarzaną linią (lewo-dół), uruchomione procesy i związane z nimi instancje klas (lewo-góra) oraz podgląd stanu zmiennych programu (prawo-góra)

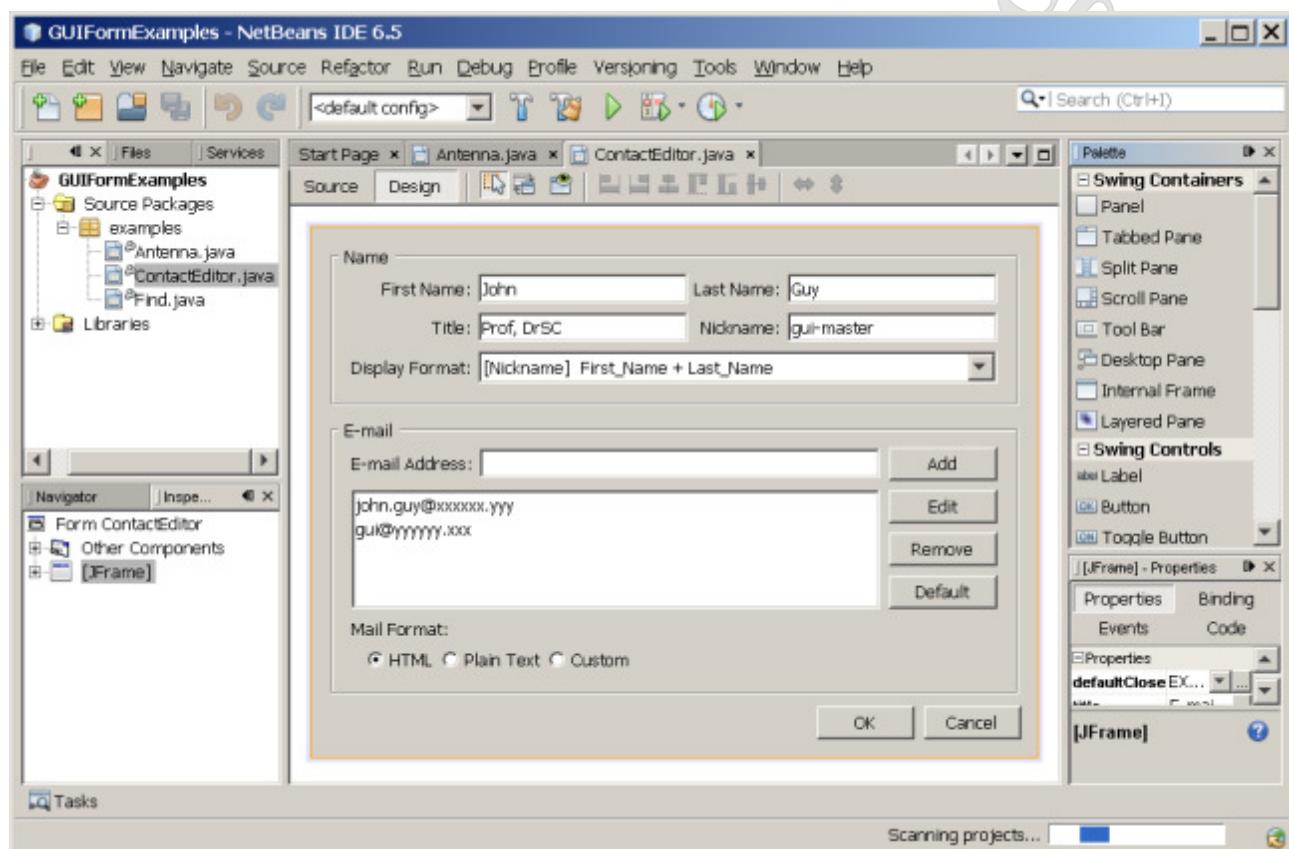


Rys. 12.4: Debugger uruchomiony w środowisku Eclipse.

Nowoczesne narzędzia klasy IDE posiadają często wsparcie tworzenia graficznego interfejsu użytkownika (GUI – ang. *Graphical User Interface*). 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 obsłużenie zdarzeń – akcji użytkownika. Edytory GUI dla technologii internetowych pozwalają także na wygodne projektowanie stron WWW połączonych z daną logiką realizowaną przez pisany kod wykonywalny.

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 reinikcji 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 szybkie 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. Rys. 12.5 przedstawia tworzenie prototypu interfejsu użytkownika w środowisku NetBeans - widoczny przybornik z elementami UI i właściwości wybranych elementów na prawym panelu oraz podgląd tworzonego formularza na środkowym.



Rys. 12.5: Tworzenie prototypu interfejsu użytkownika w środowisku NetBeans

Mnogość narzędzi z jakimi współpracują IDE powoduje, że środowiska często mają budowę modułarną, opartą na „wtyczkach” (ang. *plug-ins*). Taka konstrukcja aplikacji CASE pozwala ich użytkownikom na wybór zbioru technologii, jakie chcą wykorzystać w procesie twórczym. Jest to ważne z uwagi na dynamikę rozwoju rynku narzędzi informatycznych – każdego dnia pojawią się nowe „instrumenty” inżynierii oprogramowania, a dla tych o już ugruntowanej pozycji na rynku wydawane są nowe wersje.

12.4. Narzędzia ciągłej integracji

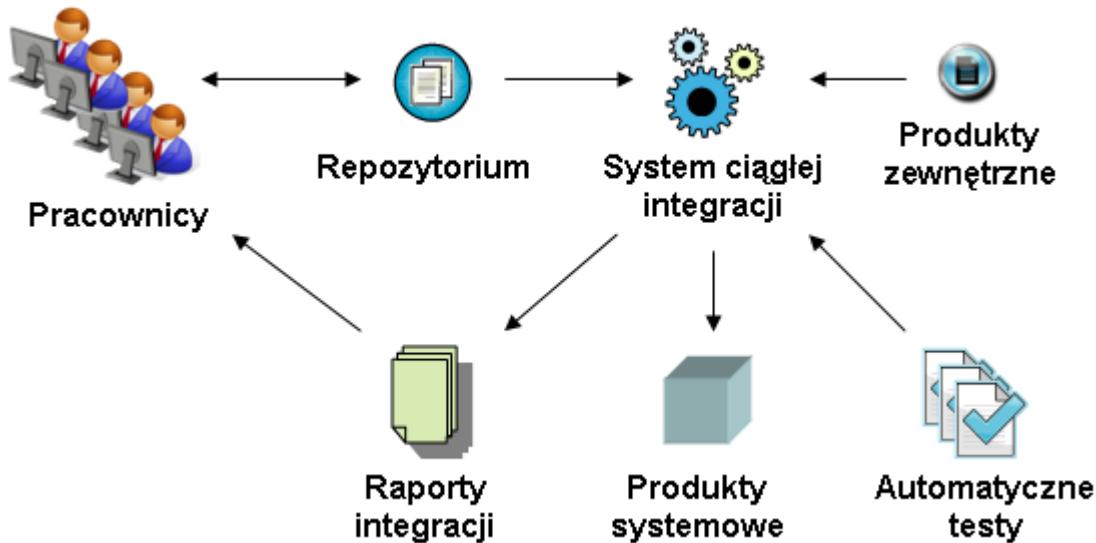
Główym celem istnienia narzędzi ciągłej integracji jest usprawnienie procesu kombinacji zmieniających się elementów systemu. W rozproszonym systemie pracy grupowej lokalna integracja prowadzona przez każdego uczestniczącego w procesie często okazuje się praco- i czasochłonna: każdy pracujący z kodem musi zaktualizować swoją roboczą kopię systemu, skompilować ją, uruchomić, przetestować, wprowadzić własne zmiany, ponownie skompilować, przetestować i dopiero zapisać zmiany w repozytorium centralnym. Przy odpowiednio dużych projektach próby takiej integracji okazują się częstokroć daremne: ilość zmian zachodzących w projekcie okazuje

się być tak duża, a projekt tak skomplikowany, że długi czas integracji powoduje ustawiczną nieaktualność kopii roboczych i wyników pracy z nimi. Integracja lokalna może być nie odzwierciedlać efektów integracji centralnej z uwagi na specyfikę systemu (np. współpracującego ze zdalnymi aplikacjami będącymi poza kontrolą jednostek zaangażowanych w projekt).

Centrum systemu ciągłej integracji stanowi odpowiednie oprogramowanie, które potrafi stworzyć działającą wersję rozwijanego produktu wykorzystując popularne narzędzia i zewnętrzne artefakty programistyczne. Wśród najpopularniejszych systemów ciągłej integracji można wymienić Apache Continuum, Bamboo, CruiseControl, Microsoft Team Foundation Server oraz Mozilla Tinderbox.

System ciągłej integracji opiera się na centralnym repozytorium artefaktów systemu oraz repozytorium produktów systemowych. Często oba repozytoria działają w obrębie jednej instancji bazy danych czy systemu kontroli wersji. Dodatkowymi elementami systemu są zestawy automatycznych testów, system umożliwiający raportowanie wyników integracji, narzędzia pozwalające na opisanie sposobu integracji (może to być np. make, Apache Ant, Apache Maven albo komercyjne Microsoft Build czy IBM Rational Build Forge) oraz produkty zewnętrzne (np. biblioteki wykorzystywane w projekcie, a produkowanych przez firmy trzecie).

Schemat działania takiego systemu wygląda jak przedstawiono na Rys. 12.6. Pracownicy w toku rozwoju projektu umieszczają w repozytorium wyniki swojej pracy. Narzędzie ciągłej integracji wg. określonego harmonogramu pobiera odpowiednie zbiory artefaktów z repozytorium kodu i na podstawie dokładnie opisanych zasad tworzy z nich, w oparciu o produkty zewnętrzne, produkty systemowe. Są one poddawane automatycznym testom w celu określenie ich cech. Pod koniec procesu tworzone są raporty integracji, obejmujące takie zagadnienia jak: błędy i ostrzeżenia komplikacji, komunikaty związane z produktami zewnętrznymi, rezultaty uruchomienia testów automatycznych na skonsolidowanych wersjach produktów systemowych, wyniki działania rozmaitych analizatorów produktów i artefaktów z repozytorium (np. ocenяacych kod pod kątem jego zgodności z zasadami programowania ustalonego w obrębie organizacji), a także automatycznie generowaną dokumentację.



Rys. 12.6: Schemat działania systemu ciągłej integracji

Wdrożenie narzędzi ciągłej integracji jest dość skomplikowane i ma znaczenie szczególnie przy dużych, złożonych przedsięwzięciach informatycznych, gdzie koszt organizacji systemu równoważony jest korzyściami z jego stosowania.

12.5. Zastosowanie narzędzi do zarządzania konfiguracją i zmianami

Podstawy zagadnień zarządzania zmianami i konfiguracją opisano w sekcjach 10.4 i 10.5. Z uwagi na istotność tych zagadnień wsparcie narzędziowe dotyczące rozwoju i utrzymania systemu jest nieodzowne.

W sekcji 10.3 omówiono już podstawowe narzędzie pozwalające na panowanie nad zmianami w projekcie – system kontroli wersji. Wersjonowane repozytorium jest jednak tylko bazą, na której swoje funkcjonowanie opiera wiele innych systemów wspierających zarządzanie zmianami – może to być oprogramowanie służące do ciągłej integracji (patrz: sekcja 12.4), ale też systemy dokumentacji zmian. Służą one do formalizacji dialogu między osobami tworzącymi produkt a testerami i użytkownikami systemu. Komunikacja ta dotyczy problemów związanych z funkcjonowaniem aplikacji, ale zawierać może też propozycje udoskonaleń produktu.

Oprogramowanie umożliwiające śledzenie błędów lub innych zagadnień związanych z systemem (ang. *bug tracker*, *issue tracker*) pozwala na dokładne opisanie przykładów niepożdanego działania systemu. Użytkownicy systemu, poprzez odpowiedni interfejs (najczęściej sieciowy), wypełniają formularze opisujące zagadnienia. Opis taki, poza otwartym tekstem dokumentującym zagadnienie, zawiera szereg atrybutów usprawniających zarządzanie ogólnem 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 „w którym komponencie jest najwięcej poważnych usterek”, „który programista ma przypisanych najwięcej zadań”, „które wydanie wymagało najmniej poprawek”, „które komponenty wymagały zmiany po rozwiązaniu zagadnienia Z” itp. Większość popularnych systemów śledzenia zagadnień pozwala na generację raportów opartych o informacje zawarte w opisach.

The screenshot shows the Bugzilla web interface for reporting a new bug. At the top, there's a navigation bar with links for Home, New, Browse, Search, Log out, Search, Reports, My Requests, Preferences, and Help. Below the navigation is a message encouraging users to read writing guidelines and search for existing bugs. The main form has fields for Product (set to WorldControl), Reporter (a dropdown menu), Component (a dropdown menu showing EconomicControl, PoliticalBackStabbing, WeatherControl), Version (set to 1.0), Severity (set to normal), Hardware (set to PC), OS (set to Windows XP), and a large Description text area. A summary field is also present. At the bottom, there's an Attachment section with an 'Add an attachment' button.

Rys. 12.7: Widok formularza zgłaszania usterki w systemie śledzenia błędów BugZilla

Odpowiednio opisane zagadnienia rozsyłane są do osób odpowiedzialnych za dane moduły albo wydania w celu poddania procedurze obsługi zmiany. Warto podkreślić, że narzędzia śledzenia zagadnień pozwalają na organizację przepływu informacji, dokumentów i pracy (ang. *workflow*) w obrębie przedsiębiorstwa. Pozwalają one na wygodną i jednocześnie formalną komunikację międ-

dy przedstawicielami producenta oprogramowania a użytkownikami aplikacji. Wymiana informacji jest dokumentowana, a jej przebieg można ograniczyć do schematu przepływu pracy charakterystycznego dla danego przedsięwzięcia.

Zarządzanie konfiguracją jest także wspierane przez narzędzia CASE powalające opisywać wydania, konfiguracje sprzętowo-programowe środowisk uruchomieniowych i aplikować w tych środowiskach (często niejednorodnych, rozproszonych i wielomaszynowych) produkty systemowe. Operacje ułatwiające pracę z takimi środowiskami to możliwość instalacji, konfiguracji i aktualizacji oprogramowania przeprowadzana bez bezpośredniego dozoru administratorów systemów i wdrożeniowców.

12.6. Narzędzia wspierające proces testowania

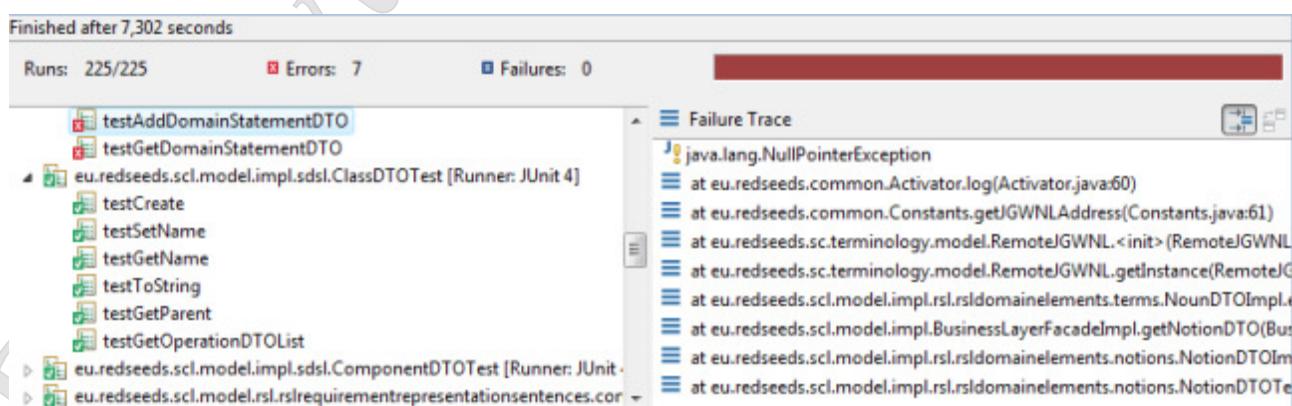
Istnieje szereg narzędzi pozwalających na usprawnienie procesu testowania. Umożliwiają one automatyzację powtarzalnych i kosztownych operacji badania rozmaitych aspektów poprawności programów.

Jedną z podstawowych funkcjonalności narzędzi wspierających testowanie jest „nagrywanie” interakcji użytkowników z badanym systemem. Proces rejestracji dialogu użytkownik-aplikacja polega na zapisie decyzji użytkownika (naciśniętych przez niego przycisków, odwiedzonych opcji menu, wprowadzonych danych itd.). Taki zapis (akcje użytkownika oraz ich parametry) może być później modyfikowany.

„Nagrania” działań użytkowników mogą posłużyć testowaniu funkcjonalności systemu, jeśli realizują scenariusze przypadków testowych (patrz: sekcja 11.4). Tego typu testy znacznie obniżają koszty weryfikacji systemu po dokonaniu w nim zmian: każde kolejne uruchomienie opiera się na już raz wykonanej (i zarejestrowanej) pracy testera – są to tzw. testy regresywne. 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” akcji użytkownika mogą towarzyszyć różne zestawy danych, co pozwala na dokładne przetestowanie systemu.

Popularne narzędzia pozwalają także na „odtwarzanie” wielu instancji tych samych „nagrań” jednocześnie, co pozwala na przetestowanie odporności systemu na określone obciążenia (patrz: sekcja 11.5). Zarówno zestawy danych, jak i parametry akcji użytkowników mogą podczas testów obciążeniowych 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.

„Nagrania” ścieżek aktywności użytkowników aplikacji mogą także służyć do weryfikacji poprawności interfejsu użytkownika (w szczególności typu graficznego). Tego typu testowanie pozwala nie tylko na zbadanie reakcji interfejsu użytkownika czy poprawności jego struktury (jak np. zachowane są pozycje elementów na ekranie), ale także przejść między stanami aplikacji (np. kolejność pokazywanych ekranów).



Rys. 12.8: Narzędzie testów jednostkowych JUnit zintegrowane ze środowiskiem Eclipse

O ile powyżej opisane rodzaje testów dotyczą głównie metod „czarnej skrzynki”, to istnieje także wsparcie narzędziowe dla testów „przezroczystej skrzynki”. Różnego rodzaju analizatory kodu pozwalają na weryfikację zgodności efektów pracy programistów z przyjętymi w ramach danej organizacji praktykami tworzenia kodu (patrz sekcja 10.2). Badanie kodu pozwala 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.

Wsparcie dla testowania oferują także narzędzia 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 (dane testowe są określone dla każdego uruchomienia oddzielnie, inne części systemu reprezentowane są zaślepkiem). Uruchomienie testów jest kontrolowane – kroki przypadków testowych realizowane są w określonej kolejności, a wyniki działania 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. Jak wspomniano w sekcji 12.3 narzędzia wspierające testy jednostkowe łatwo integrują się z popularnymi środowiskami programistycznymi – prosty przykład działania narzędzia w obrębie IDE zawiera Rys. 12.8.

12.7. Podsumowanie

Z bogatego zbioru narzędzi wspierających rozwój oprogramowania każda organizacja powinna wybrać te odpowiadające podejmowanym przez nią przedsięwzięciom. Ważne kryteria selekcji narzędzi CASE to m.in. wielkość zespołów pracujących nad produktami, stopień komplikacji rozwiązywanych systemów, charakterystyka docelowych platform uruchomieniowych, ograniczenia środowiskowe projektu, plany dotyczące utrzymania powstałych aplikacji, a także metodykę, 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, a także procentować w przyszłości przez wyższą jakość produktów powstających przy mniejszym nakładzie pracy.

13. Inżynieria procesu wytwórczego oprogramowania

Celem każdego projektu konstrukcji oprogramowania jest wykonanie systemu oprogramowania spełniającego wymagania sponsora projektu w określonych ramach czasowych i budżetowych. Działania podejmowane w celu wykonania systemu oprogramowania polegające na projektowaniu, konstrukcji, modyfikacji, utrzymaniu efektywnych kosztowo rozwiązań dla praktycznych problemów, z wykorzystaniem wiedzy naukowej oraz technicznej składają się na inżynierię oprogramowania.

Mimo iż inżynieria oprogramowania jest dziedziną stosunkowo młodą, to tak jak każda inna dziedzina inżynierijna, wytworzyła już pewien zbiór mechanizmów, narzędzi, norm i standardów, których celem jest wspomaganie i kontrola procesów wytwórczych oprogramowania.

Poniższy rozdział nie opisuje wszystkich zagadnień związanych z inżynierskim podejściem do procesu wytwórczego. Jego zadaniem jest wskazanie i krótkie omówienie kilku podstawowych elementów inżynierijnych – od zarządzania konstrukcją oprogramowania, przez organizację pracy zespołów wytwórczych i mierzenie oprogramowania, do ponownego jego użycia w innych przedsięwzięciach. To krótkie podsumowanie ma celu wprowadzenie czytelnika w rozległy obszar zagadnień związanych z inżynierią procesu wytwórczego.

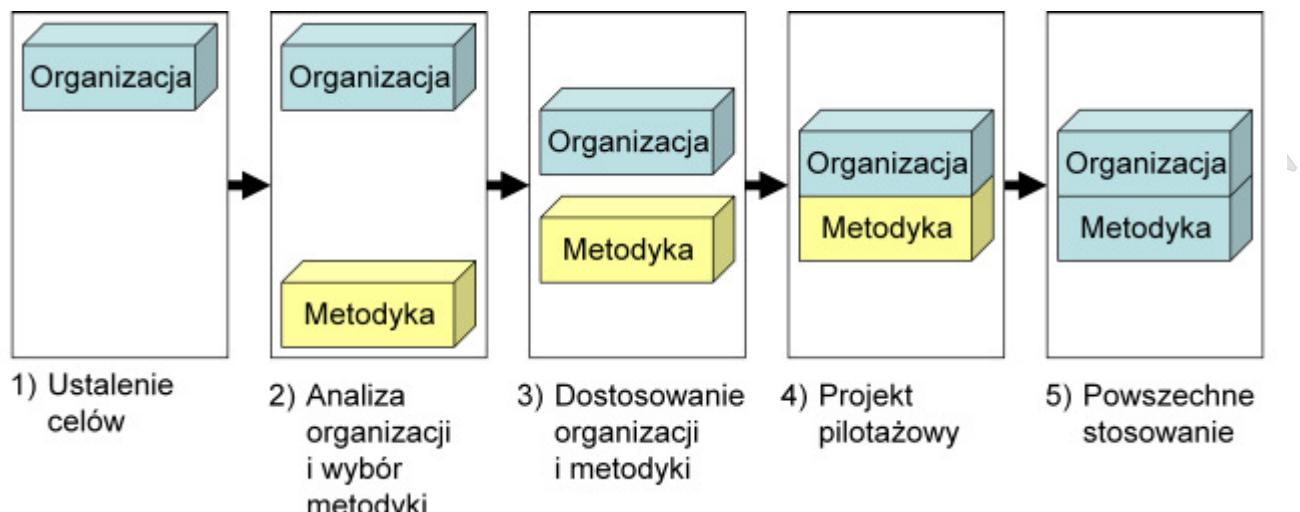
13.1. Inżynierskie aspekty zarządzania konstrukcją oprogramowania

Wszystkie działania techniczne i organizacyjne, które podlegają planowaniu i mają prowadzić do wytworzenia systemu oprogramowania, nazywane są projektem informatycznym. W celu osiągnięcia sukcesu wszystkie działania powinny być skoordynowane, tak aby ich wzajemne relacje zapewniały sprawny przepływ informacji, a postawione problemy mogły być efektywnie rozwiązywane. Koordynację wszystkich działań dotyczących wytworzeniem oprogramowania realizuje się poprzez zarządzanie projektem informatycznym. Jest ono niezbędne, ponieważ inżynieria oprogramowania, tak jak każda inna dziedzina inżynierii, podlega zarządzaniu. Dzięki temu możliwe jest zapewnienie, że podjęte przedsięwzięcie zmieści się w narzuconych ograniczeniach i doprowadzi do dostarczenia oczekiwanej oprogramowania.

Zarządzający konstrukcją oprogramowania zwykle wspomagają się metodą wytwarzania oprogramowania i na jej podstawie wykonują czynności planowania i tworzenia harmonogramu przedsięwzięcia, szacując jego koszty, organizując pracę zespołów wytwórczych, śledzą postępy prac, wspierając ponowne wykorzystanie oprogramowania, jako sposób tańszego wytworzenia oprogramowania. Kierują się przy tym narzędziami, standardami i normami, które mają pomóc, aby projekt informatyczny przebiegał zgodnie z harmonogramem i w ramach wyznaczonego budżetu.

13.2. Wdrażanie metodyk wytwarzania oprogramowania do praktyki organizacji wytwarzających oprogramowanie

W celu efektownego wykorzystania metodyki w procesie wytwarzania oprogramowania, należy wykonać pewne czynności związane z wdrożeniem wybranej metodyki w organizacji odpowiedzialnej za wykonanie systemu oprogramowania, jak również w projekcie, którego celem jest wytworzenie takiego systemu. Rys. 13.1 przedstawia uproszczony proces wdrażania metodyki w organizacji, której zadaniem jest wytwarzanie oprogramowania.



Rys. 13.1. Uproszczony proces wdrażania metodyki

Pierwszym krokiem tego procesu jest ustalenie celów (1), jakie mają być osiągnięte dzięki pracy wspomaganej metodą. Można wyróżnić wiele typów motywacji do wdrożenia nowej metodyki w organizacji. Główne z nich to [Szej02]:

- wzrost efektywności działań twórczych,
- podniesienie poziomu satysfakcji klientów poprzez wzrost szybkości rozwiązywania problemów oraz zapewnienie powtarzalności rozwiązań,
- zmniejszenie pracochłonności rozwiązywania problemów,
- organizacja wiedzy pracowników i łatwiejsze jej wykorzystanie.

Decyzja o wdrożeniu wybranej metodyki wytwarzania oprogramowania powinna być poprzedzona analizą stanu i możliwości danej organizacji (2). Każda firma zajmująca się wytwarzaniem oprogramowania posiada własną specyfikę. Różne są zasoby ludzkie, kompetencje, organizacja i styl pracy, infrastruktura. Czasem wybrani, najbardziej doświadczeni programiści, którzy posiadają predyspozycje analityczne, zajmują się formułowaniem wymagań. W innej firmie analitycy, którzy mają doświadczenie programistyczne lub kierownicze, uczestniczą w implementacji lub kierują procesem testowania. Zdarza się, że w procesie wytwarzania oprogramowania biorą udział zespoły rozproszone nawet po całym świecie, gdzie ze względu różne strefy czasowe, czas pracy pokrywa się tylko w małym stopniu, lub nie pokrywa się wcale. Często analitycy, którzy znajdują się blisko klienta, współpracują z architektami i programistami, znajdującymi się w zupełnie innym miejscu. Przez to spotkania są ograniczane, bądź nawet rezygnuje się z nich całkowicie na rzecz telekonferencji lub wymiany dokumentów. Duże znaczenie ma także styl pracy. Duże firmy, korporacje, gdzie obarczony procedurami i biurokracją długi proces decyzyjny może wydłużać realizację zadań i ograniczać pomysły i cechy indywidualne członków zespołu, zasadniczo różnią się od firm mniejszych, gdzie jest łatwy dostęp do wszystkich członków zespołu i kadry zarządzającej i gdzie promuje się innowacyjność. Nie bez znaczenia pozostaje także infrastruktura działania danej firmy. Możliwe jest, że duże koszty, w rozwijającej się firmie, ograniczają całosciowe wdrożenie skomplikowanych narzędzi do kontroli i wspomagania wytwarzania oprogramowania lub ogranicza się „obsługę” metodyki do narzędzi, które już funkcjonują lub które są zalecane przez kadrę zarządzającą.

Na podstawie wyników tej analizy opracowywany jest plan wdrożenia metodyki, który obejmuje kolejne kroki wprowadzania nowego standardu w organizacji. Plan taki uwzględnia ustalenie zakresu wdrożenia, oszacowania dotyczące wymaganego budżetu, czasu i szkoleń, harmonogram. Należy się tu kierować zasadą „minimum kosztów, maksimum zysków”. Czasem zmiana formy lub nawet rezygnacja z pewnych elementów metodyki przyniesie większe zyski niż zmiana nawyków pracy lub obarczenie członków zespołu dodatkowymi obowiązkami (np. wypełnianie formularzy zgodnie z zaleceniami metodyki). Nie mniej jednak czasem konieczne jest wprowadzenie nowych standardów pracy.

Zgodnie z planem następuje dostosowanie organizacji i samej metodyki do wykorzystania pełnych możliwości zarówno organizacji jak i metodyki (3). Zmiany te powinny jednak być wprowadzane łagodnie, tak aby pokazać, że nowe podejście jest potrzebne, i że ma wpływ na końcowy sukces procesu wytwarzania oprogramowania.

Kolejnym krokiem wdrożenia metodyki jest przeprowadzenie projektu pilotażowego (4). Powinien być to projekt rzeczywisty (ale o odpowiednio małym ryzyku niepowodzenia), gdyż tylko w takiej sytuacji będą zauważalne zalety, jak również wady przebiegu procesu zgodnego z metodyką. Uwagi przekazane przez członków zespołu realizującego projekt pilotażowy są podstawą do dalszego wzajemnego dostosowania metodyki i organizacji.

W trakcie pracy nad kolejnymi projektami (5) może okazać się, że założenia wynikające z analizy poprzedzającej wdrożenie metodyki, nie są trafne. Sygnały odbierane od kierowników projektów o właściwym przebiegu procesu budowy oprogramowania, powinny być impulsem do ponownej analizy i wzajemnym dostrojeniu metodyki i organizacji do siebie.

Podejmując się realizacji kolejnego projektu zgodnie z metodyką już wdrożoną w organizacji, należy również wdrożyć metodykę w projekcie. Sprowadza się to do pokierowania współpracy wykonawcy i klienta drogą obsługiwana przez metodykę (notacja, techniki, proces).

Właściwie żadna metodyka nie nadaje się do wykorzystania w kształcie, w jakim przedstawia ją definicja. Dlatego konieczny jest proces wdrożenia, który dostosuje metodykę do potrzeb organizacji. Niektóre metodyki oferują narzędzia do przystosowywania ich do konkretnych potrzeb. Przykładem może być narzędzie Rational Metod Composer, które umożliwia „dostrojenie” metodyki RUP do indywidualnych potrzeb organizacji i projektu.

13.3. Organizacja zespołów wytwarzających oprogramowanie

Nawet najlepsze narzędzia, metodyki, notacje, techniki nie są w stanie zapewnić powodzenia procesowitworzenia oprogramowania, jeżeli nie zostaną one wykorzystane przez sprawny zespół realizacyjny. To właśnie ludzie, a nie komputery, oprogramowanie, procedury, budynki, są tworzą rzeczywisty potencjał firm zajmujących się wytwarzaniem oprogramowania. Dlatego tak ważne jest utrzymywanie kompetentnej i zmotywowanej kadry informatyków. Opracowany przez Instytut Inżynierii Oprogramowania (*Software Engineering Institute*) standardowy model dojrzałości zarządzania pracownikami (*people capability maturity model – P-CMM*) wskazuje firmom wytwarzaniu oprogramowania najważniejsze elementy zarządzania zasobami ludzkimi. Obejmuje on rekrutację i selekcję, ocenę wyników pracy, motywację, szkolenia, wynagradzanie, rozwój umiejętności i kariery, organizację pracy i kulturę organizacyjną (szczegółowe informacje o P-CMM można znaleźć na stronie www.sei.cmu.edu/cmm-p). Wysoki poziom zarządzania pracownikami, znacznie zwiększa efektywność pracy i pozwala skutecznie stosować metody inżynierii oprogramowania.

Analiza i tworzenie zasad zarządzania zespołami ludzi jest zajęciem interdyscyplinarnym. Obejmuje zagadnienia związane z psychologią, socjologią. W tej sekcji zostaną omówione tylko poszczególne zagadnienia dotyczące organizacji zespołów wytwarzających oprogramowania. Najpierw jednak należy przedstawić wszystkich uczestników, którzy mają wpływ i biorą udział w procesie wytwarzania oprogramowania.



Rys. 13.2.: Uczestnicy procesu wytwarzania oprogramowania

Na Rys. 13.2 wszyscy uczestnicy procesy wytwarzania oprogramowania zostali podzieleni na pięć grup [Pres04]. Są to:

- Dyrektorzy i prezesi, którzy określają politykę firmy i w ten sposób wpływają na realizowane przedsięwzięcia.
- Kierownicy, którzy planują, inspirują, organizują i kontrolują działania twórców oprogramowania.
- Twórcy oprogramowania, czyli analitycy, architekci, projektanci, programiści, testerzy, administratorzy i inni informatycy, którzy mają techniczną wiedzę i umiejętności niezbędne do stworzenia produktu.
- Klienci, którzy określają wymagania co do produktu, i inni uczestnicy, którym zależy na sukcesie przedsięwzięcia.
- Użytkownicy, którzy stosują gotowy produkt na co dzień.

Łatwo sobie wyobrazić, że współpraca uczestników w ramach jednego wielkiego zespołu, byłaby chaotyczna, a zarządzanie takim zespołem byłoby niemożliwe. Dlatego należy stworzyć mniejsze wyspecjalizowane zespoły, aby jak najlepiej wykorzystać ich umiejętności i możliwości poprzez zorganizowanie i skoordynowanie pracy.

Zespoły powinny posiadać liderów, którzy będą reprezentować zespół, a zarazem będą kierować jego pracę. Liderzy powinni posiadać predyspozycje do współpracy z ludźmi. Należy wystrzegać się przed przypadkowym mianowaniem na kierowników pracowników nieprzygotowanych. Aby praca kierownika przynosiła spodziewane efekty, jego działania powinny skupiać się trzech głównych aspektach:

- Motywacja – Członkowie zespołu powinni być zachęcani do pracy. Nie należy stosować bezpośredniego nacisku, ale rozbudzić w pracowniku potrzebę i chęć pracy na pełnych obrotach.
- Organizacja – Praca zespołu powinna przebiegać sprawnie. Należy zapewnić sprawny przepływ informacji i koordynację wszystkich prac. W miarę potrzeb należy zmodyfikować

istniejące lub stworzyć lub nowe procesy. Ma to na celu sprawne przejście od początkowego problemu poprzez pomysł na realizację do gotowego produktu.

- Innowacja – Efektywność pracy wzrasta, jeżeli pracownicy czują się rzeczywistymi twórcami. Jeżeli ich działanie jest ścisłe ograniczone wymaganiami przedsięwzięcia, należy sprawić, aby pracownicy mieli wrażenie, że robią coś nowego.

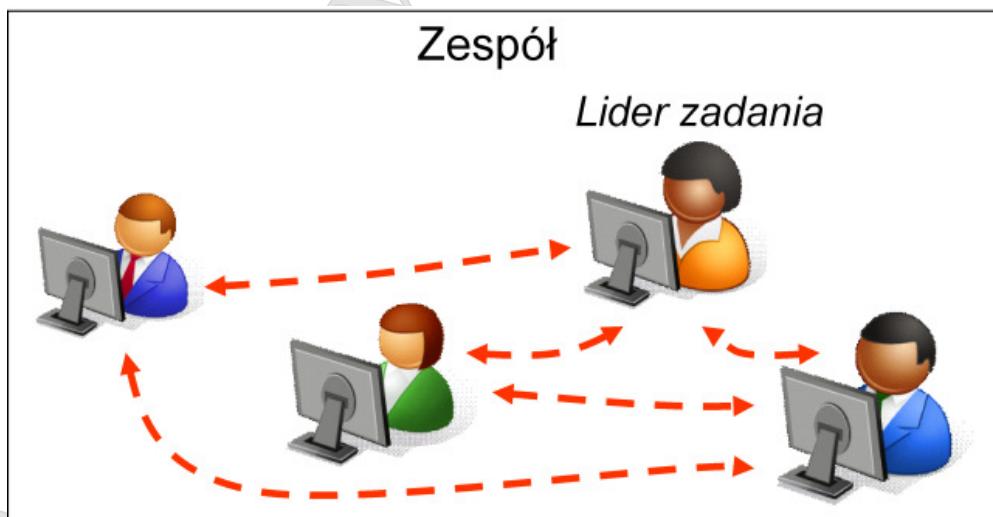
Dobrym sposobem na zarządzanie zespołami jest ukierunkowywanie go rozwiązywanie konkretnych problemów. Oznacza to, że liderzy powinni dokładnie zrozumieć problem i przedstawić go swojemu zespołowi i zachęcić do swobodnej wymiany pomysłów. W tym samym czasie, liderzy powinni pilnować, aby cały czas były podejmowane działania na rzecz zachowania jakości powstającego produktu.

Aby kierownicy zespołów (nie tylko wytwarzających oprogramowanie) dobrze pełnili swoją rolę powinni posiadać zestaw odpowiednich cech. Bez wątpienia są to zdolności analityczne, wnioskowania. Dzięki temu kierownicy rozpoznają najważniejsze problemy techniczne i organizacyjne. Na tej podstawie mogą ukierunkować pracę zespołu do ich rozwiązania. Osobowość przywódcy pozwoli godnie reprezentować cały zespół poprzez przejęcie za niego odpowiedzialności i w krytycznych momentach przejąć nad nim kontrolę. Poprzez dążenie do sukcesu, kierownik daje przykład członkom zespołu, że należy wykazywać się kreatywnością. Zdolność do wywierania wpływu na ludzi pozwala zbudować zespół, a odporność na stres kierować nim w każdej sytuacji.

Zespoły powinny posiadać odpowiednią strukturę, aby liderzy mogli efektywnie organizować ich pracę i nimi kierować. Zadaniem lidera nie jest zmiana kultury organizacyjnej firmy, ale organizację i kierowanie pracą ludzi będącymi członkami jego zespołu. Istnieje bardzo wiele sposobów organizacji pracy w firmach zajmujących się wytwarzaniem oprogramowania – można powiedzieć, że w każdej z nich mamy do czynienia z innym sposobem, ponieważ każda firma posiada własną specyfikę, zwyczaje, zasoby. Praktyka pokazuje jednak, że warianty oparte o formalny podział pracowników na zespoły daje najlepsze wyniki.

Tak samo jak występuje wiele sposobów organizacji pracy, tak samo istnieje wiele schematów definiujących strukturę zespołu. Jako przykłady takich schematów, spośród wielu zdefiniowanych struktur zespołu, poniżej zostaną przedstawione trzy podstawowe schematy struktury zespołów.

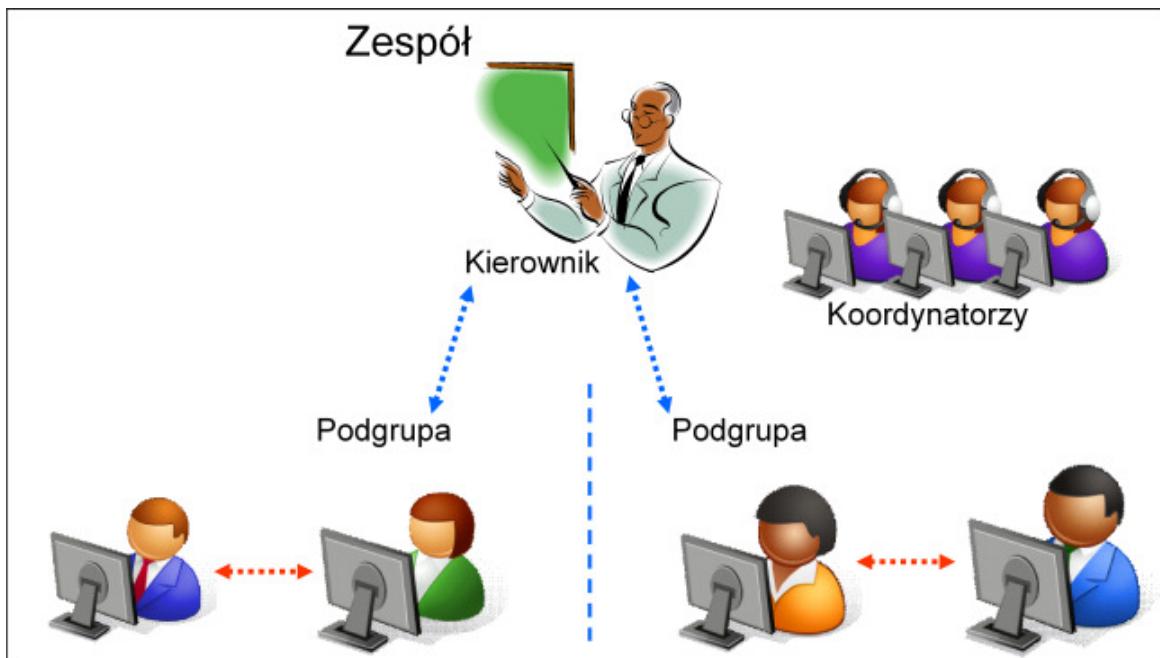
Demokratyczny, rozproszony. Schemat przedstawiony na Rys. 13.3. W zespole tego typu brak jest stałego lidera. Różne osoby, w zależności od bieżącej potrzeby, koordynują pracę nad poszczególnymi zadaniami. Członkowie zespołu komunikują się ze sobą bezpośrednio (komunikacja pozioma), a wszystkie decyzje podejmowane są wspólnie drogą głosowania.



Rys. 13.3. Demokratyczny, rozproszony schemat struktury zespołu

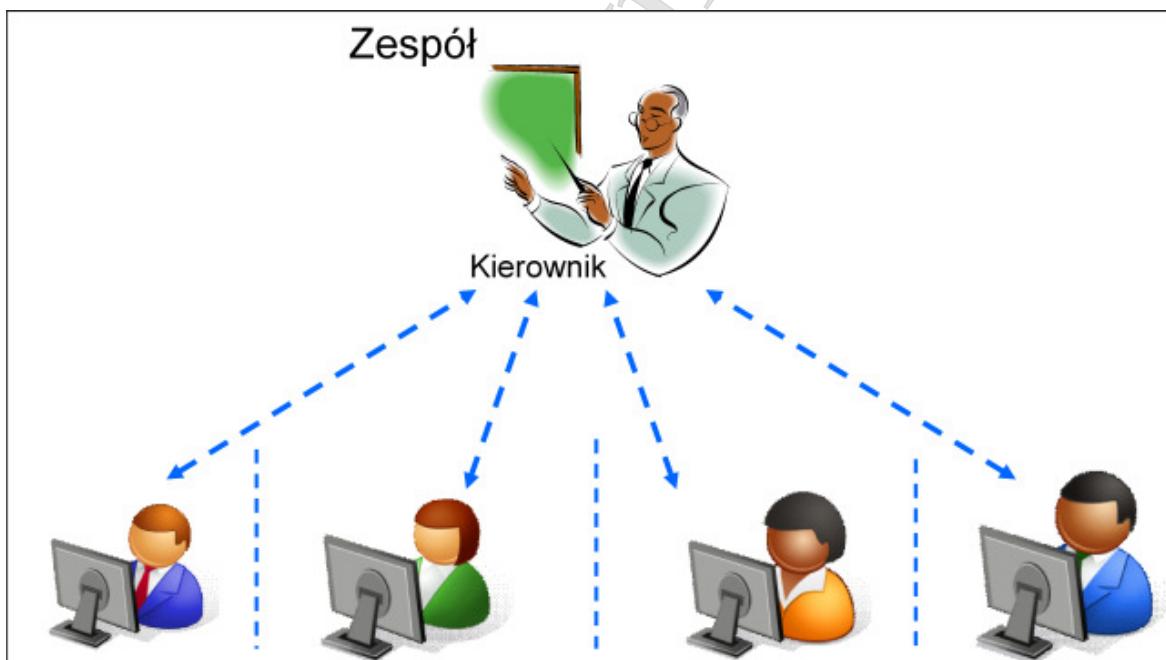
Kontrolowany, rozproszony. Schemat przedstawiony na Rys. 13.4. Występuje ustalony kierownik oraz kilku koordynatorów, którzy wspomagają go w różnych obszarach. Cały zespół pracuje nad rozwiązywaniem każdego problemu, ale może on zostać podzielony przez kierownika w celu

realizacji różnych zadań. Członkowie podgrup komunikują się bezpośrednio (komunikacja pozioma), podgrupy między sobą komunikują się za pośrednictwem kierownika (komunikacja pionowa).



Rys. 13.4. Kontrolowany, rozproszony schemat struktury zespołu

Kontrolowany, scentralizowany. Schemat przedstawiony na Rys. 13.5. Za koordynację pracy całego zespołu odpowiedzialny jest kierownik, a cała komunikacja odbywa się za pośrednictwem kierownika (komunikacja pionowa).



Rys. 13.5. Kontrolowany, scentralizowany schemat struktury zespołu

Dzięki koordynowaniu prac przez kierownika, zespoły kontrolowane mogą szybko i skutecznie rozwiązywać proste problemy. Zespoły rozproszone lepiej radzą sobie z zdaniami trudniejszymi, dostarczając wiele dobrych rozwiązań. Do zadań najtrudniejszych najlepiej nadają się demokratyczne zespoły rozproszone. Duże przedsięwzięcia lepiej jest prowadzić, używając struktur kontrolowanych. Ponieważ wydajność zespołu jest odwrotnie proporcjonalna do intensywności koniecznej komunikacji między jego członkami, to podział zespołu na podgrupy zmniejszy potrzebę komunikacji. Praca w zespole zarządzanym demokratycznie przynosi pracownikom dużo satysfakcji, co znacznie podnosi poziom efektywności pracy. Taka organizacji pracy ma więc zalety w wypadku

długo istniejących zespołów. Taka struktura zespołu sprawdza się również tam, gdzie istnieje konieczność prowadzenia intensywnej komunikacji, czyli w przypadku problemów, których nie da się poddać modularyzacji. Jeżeli problem można podzielić, a członkowie zespołu mogą pracować niezależnie, to dobrym rozwiązańiem jest struktura kontrolowana. Zwykle mniej błędów wytwarzają zespoły kontrolowane, a zespoły rozproszone potrzebują więcej czasu na ukończenie prac niż zespoły scentralizowane.

Podobnych porównań można by wymieniać jeszcze wiele. Każdy z trzech schematów posiada wady i zalety. Przy wyborze należy kierować się kilkoma czynnikami, takimi jak trudność problemów do rozwiązania, przewidywany czas istnienia zespołu, modularyzacja rozwiązania problemu, intensywność, z jaką pracownicy będą musieli się komunikować między sobą. Wybór odpowiedniej struktury zespołu ma na celu wykorzystanie twórczego zapału pracowników do intensywnego działania zgraneego zespołu. Wydajność zespołu osiąga się przez wzajemne zaufanie, kompetencje odpowiednie do potrzeb przedsięwzięcia oraz przez zgranie członków zespołu. W takim zespole znacznie wzrasta prawdopodobieństwo osiągnięcia sukcesu, a pracownicy nie potrzebują zbytnej motywacji i zarządzania.

Rzeczywistość nie zawsze jednak sprzyja budowaniu zgranych i wydajnych zespołów. Niektóre z nich można nazwać „toksycznymi”. Powstaniu takich zespołów sprzyjają czynniki zawarte w Tabl. 13-1. Obok w prawej kolumnie są przedstawione sposoby radzenia z wybranymi problemami.

Tabl. 13-1. Problemy zespołów wytwarzających oprogramowanie i ich rozwiązania

Problem	Rozwiązanie
Atmosfera chaosu w pracy powoduje, że pracownicy tracą dużo energii na działania nie związane z pracą i gubią z pola widzenia cel swego działania.	Kierownik zapewnia dostęp członków zespołu do wszystkich informacji potrzebne im do pracy. Unikanie niepotrzebnych modyfikacji ogólnych celów i zadań stojących przed zespołem. Szybka reakcja na przychodzące zmiany.
Frustracja spowodowana kłopotami osobistymi lub zawodowymi, co prowadzi do konfliktów między pracownikami	Jak najczęstsze udzielenia zespołowi prawa do podejmowania decyzji - im większa kontrola pracowników nad procesem twórczym, tym mniejsza frustracja.
Wadliwe procedury postępowania i niedoskonały lub źle wybrany proces twórczy, będący przeszkodą w osiąganiu sukcesu	Wybranie modelu procesu twórczego spełniającego wymagania danego przedsięwzięcia. Samodzielna modyfikacja i ustalanie odpowiednich procedur.
Niejasny podział kompetencji, co skutkuje brakiem poczucia obowiązku i zrzucaniem odpowiedzialności na innych.	Wczesne określenie odpowiedzialności za wykonanie poszczególnych zadań i za ewentualne błędy oraz opracowanie planu działania na wypadek niepowodzenia któregoś ze współpracowników.
Świadomość ciągłych porażek i niepowodzeń, prowadząca do braku pewności siebie i do obniżenia morale zespołu	Wypracowanie metody wspólnego analizowania i pokonywania problemów. Podejmowanie zespołowych działań naprawcze. Unikanie wzajemnych oskarżeń i braku zaufania.

Bardzo ważne dla stworzenia zgranej i sprawnego zespołu są także indywidualne cechy jego członków. Każdy człowiek inaczej rozwiązuje stawiane przed nim problemy. Inaczej się zachowuje w sytuacjach stresowych. Każdy człowiek ma inny system organizacji pracy. Poznanie, zrozumienie i odpowiednie wykorzystanie osobowości wszystkich członków zespołu daje możliwość stworzenia dobrej atmosfery pracy i uzyskanie maksymalnej wydajności pracy całego zespołu.

Sposób wytwarzanie systemów oprogramowania posiada wiele cech odróżniających go od metod wytwarzania innych produktów. Są to, między innymi, różne skale wytwarzanych systemów, stale zmieniające się wymagania na system i środowisko, w którym system działa, konieczność współpracy

działania z wieloma innymi systemami. Wszystkie działania mające na celu dostarczenie sprawnego produktu na czas należy odpowiednio skoordynować. Sposób koordynacji wyznaczony jest przez metodykę według której przebiega proces wytwórczy (rozdział 3). W celu koordynacji wykorzystuje się mechanizmy komunikacji między pracownikami i zespołami. Podobnie jak w przypadku mnogości struktur organizacyjnych, tak samo istnieje wiele sposobów komunikacji zaproponowanych w różnych źródłach. Jednym z nich jest podział metod koordynacji, który wyróżnia 5 kategorii:

- Metody formalne nie wymagające kontaktu osobistego - wszelka dokumentacja i kod źródłowy, listy, okólniki, harmonogramy, narzędzia wspomagające zarządzania, formalne żądania zmian w produkcie, raporty o błędach, repozytoria.
- Metody formalne związane z kontaktem osobistym. Służą głównie zapewnieniu jakości obejmując ocenę stanu prac oraz przeglądy projektów i kodu źródłowego.
- Metody nieformalne związane z kontaktem osobistym - wszelkie spotkania zwoływane w celu wymiany poglądów, rozwiązania jakiegoś problemu lub przekazania pracownikom wiadomości.
- Komunikacja elektroniczna - poczta elektroniczna, listy dyskusyjne, fora internetowe, telekonferencje.
- Sieć komunikacji międzyludzkiej - nieformalne dyskusje między członkami zespołu, rozmowy z osobami nie biorącymi bezpośredniego udziału w przedsięwzięciu, ale dysponującymi potrzebną wiedzą i doświadczeniem.

Z analiz przeprowadzonych przez autorów powyższej klasyfikacji wynika, iż za najbardziej wartościowy sposób komunikacji to nieformalna komunikacja bezpośrednią, a czynności związane z kontrolą jakości stosowane we wcześniejszych etapach prac są bardziej wartościowe niż przeglądy kodu źródłowego na etapach końcowych.

13.4. Szacowanie złożoności oprogramowania

Każda dziedzina inżynierii wykorzystuje mechanizmy, które służą do dokonania obiektywnej oceny wytwarzanych produktów i procesów, które prowadzą do ich powstania. Wymiarowaniem nazywa się zbiór czynności, których wykonanie pozwala opisać wielkości przedmiotu wymiarowania poprzez przypisanie jego własnościom odpowiednich jednostek numerycznych lub symbolicznych.

Przed przystąpieniem do wdrożenia komercyjnego projektu informatycznego, niezbędna jest wyznaczenie jego miar, poprzez analizę podstawowych parametrów:

- złożoność (rozmiar) tworzonego oprogramowania,
- koszty wytworzenia oprogramowania,
- wielkość potrzebnego nakładu pracy,
- czas potrzebny do wytworzenia oprogramowania.

Ustalenie planu, harmonogramu i budżetu przedsięwzięcia wytwarzania oprogramowania oparte jest na powyższych miarach. Wstępne oszacowanie kosztu na samym początku procesu wytwarzania oprogramowania powinno być regularnie aktualizowane zgodnie z postępem prac wytwórczych. Pomaga to w planowaniu procesu i umożliwia efektywne korzystanie z zasobów.

Przedstawione wyżej trzy ostatnie parametry wymiarowania oprogramowania (koszty, ilość pracy, czas) są pochodną pierwszego parametru (złożoności). Do określania miar złożoności oprogramowania, oprócz precyzyjnych algorytmów, wykorzystuje się mniej rygorystyczne metody, które są uzyskiwane poprzez verbalizację lub racjonalizację intuicji analityka, popartej jedynie jego doświadczeniem lub porównaniem ze znanyimi przypadkami, analogicznymi do wymiarowanego.

Szacowanie (estymacja) oprogramowania uważane jest za zadanie bardzo trudne. Zwykle opiera się tylko na przewidywaniu rozmiarów tworzonego systemu. Niestety we wcześniejszej fazie życia oprogramowania brakuje dokładnych, jednoznacznych danych, na których oprzeć można by szacowanie wielkości, a zatem i kosztów oraz nakładu pracy. Przewidywanie złożoności oprogramo-

wania jest niezbędnym elementem procesu decyzyjnego. Jest ono jednak wartościowe jedynie wówczas, gdy jego wyniki przystają do rzeczywistości.

13.4.1. Przegląd metod szacowania oprogramowania

Istnieje wiele metod szacowania miar oprogramowania. Wielkość i złożoność systemu oprogramowania może zostać określona poprzez analizę jego wewnętrznych atrybutów, takich jak wielkość fizyczna, funkcjonalność i złożoność, a następnie zmierzona metodami statystycznymi. Przy szacowaniu oprogramowania można brać pod uwagę liczbę linii kodu jego źródeł, rozmiar specyfikacji wymagań, ograniczenia techniczne. Poniżej zostanie przedstawiony przegląd metod szacowania, a w następnej sekcji znajdzie się dokładny opis jednej z nich.

Miary uwzględniające wielkość kodu źródłowego. Jedna z podstawowych metryk rozmiaru oprogramowania. Polega na zliczaniu linii kodu źródłowego, który jest potrzebny do realizacji zadanej problemu. Wykorzystuje się jednostki LOC (*Lines Of Code*) lub KLOC (*Kilos Of Lines Of Code*). Nie ma ustalonego standardu mierzenia linii kodu, dlatego wartość LOC może być liczona z pustymi liniami lub bez nich, z komentarzami lub bez nich. Wartość jest też zależna od użytego języka programowania.

Szacowanie bottom-up i top-down. Estymacja metodą *bottom-up* zaczyna analizę od szacowania poszczególnych elementów składowych i sumując je oraz dokonując operacji na ich zbiorach, dochodzi do całościowego obrazu projektu. Estymacja *top-down* postępuje w sposób odwrotny – zaczyna od spojrzenia na projekt „z lotu ptaka”, zaś jego poszczególne elementy są traktowane jako części składowe.

Opinia ekspercka. Metoda przewidywania rozmiaru i kosztów opierająca się na szacunkach osób mających doświadczenie z analogicznymi przedsięwzięciami do estymowanego (eksperci). Ponieważ nie wymaga ona żadnych danych ilościowych, często jest stosowana w przypadku nowatorskich przedsięwzięć.

Szacowanie przez analogię. Usystematyzowanie metody opinii eksperckiej, które polega na porównywaniu szacowanego przedsięwzięcia z innymi już zakończonymi. Porównywane są różnice i podobieństwa, i na tej podstawie formułowana jest ocena. Trafność wyników szacowania jest zależna od dostępności danych historycznych, dotyczących określonych przedsięwzięć.

Algorytmy kosztorysowe. Obliczają stosunki między wielkościami wejściowymi, takimi jak rozmiar produktu, z wielkościami wyjściowymi, czyli nakładami lub czasem wykonania i wdrożenia. Algorytmy kosztorysowe służą do bezpośredniej oceny nakładów, a ich reprezentacją są formuły matematyczne lub tabele i arkusze kalkulacyjne. Przykładem metody opartej o algorytmy kosztorysowe jest COCOMO (*COmputational COst MOdel*).

Metody oparte o punkt funkcyjny. Estymacja oprogramowania oparta o punkt widzenia użytkownika, pomijająca niemal całkowicie kwestie technologiczne. Estymacja odbywa się w oparciu o dane zawarte we wstępnej specyfikacji wymagań funkcjonalnych. Dzięki niezależności tej metody od technologii, szczególnie dobrze sprawdza się ona w porównywaniu rozwiązań opartych o różnorodne platformy i w ich szacowaniu we wczesnych fazach życia. Przykładami metod opartych o punkt funkcyjny są: FPA (*Function Point Analysis*), UCP (*Use Case Points*).

13.4.2. Metoda Punktów Przypadków Użycia (*Use Case Points*)

Metoda została opracowana jako rozwinięcie metody FPA. Służy ona wymiarowaniu i szacowaniu oprogramowania realizowanego zgodnie z paradygmatem obiektowym. Szacowanie tą metodą opiera się analizie funkcjonalności – od strony użytkownika, a nie od strony projektanta lub programisty. Elementem wejściowym dla tej metody jest model przypadków użycia. Na jego podstawie naliczane są punkty i uwzględniane są pewne wagи. Poniższy opis został sporządzony w oparciu o [Schn99].

Aby wyniki estymacji były trafne, należy zwrócić uwagę na trzy podstawowe cechy modelu przypadków użycia:

- starannie zamodelowani aktorzy,
- przemyślane powiązania między poszczególnymi przypadkami użycia,
- odpowiedni poziom szczegółowości modelu (scenariusze).

Krokiem pierwszym algorytmu szacowania złożoności oprogramowania metodą UCP jest zaklasyfikowanie zidentyfikowanych aktorów do jednej z grup i przypisanie mu odpowiadającego wspólnika wag. Oto trzy grupy aktorów:

- aktor prosty - reprezentuje systemy zewnętrzne, komunikujące się przez interfejs programistyyczny API, waga = 1;
- aktor średnio złożony - najczęściej system zewnętrzny dostępny przez sieć lub terminal znakowy, waga = 2;
- aktor złożony - zazwyczaj użytkownik końcowy, komunikujący się z systemem poprzez graficzny interfejs użytkownika, waga = 3;

Sumę nie skorygowanych wag aktorów (*Unadjusted Actor Weight* – UAW) oblicza się poprzez dodanie ilości wszystkich aktorów danego typu, pomnożonych przez wielkość odpowiedniego współczynnika wag.

Następnym krokiem jest zliczanie i klasyfikacja przypadków użycia. Klasyfikację tą można wykonać na dwa sposoby: według złożonej interakcji i liczby zaangażowanych klas.

Mierzenie przypadków użycia według złożoności interakcji, polega na zliczeniu ilości kroków scenariuszy przypadku użycia (z uwzględnieniem nie powtarzających się kroków w scenariuszach alternatywnych). Na tej podstawie przypadki użycia przypisuje się do jednej z grup, które określają współczynniki wag. Przypadki użycia, pod względem złożoności interakcji, klasyfikuje się według następujących reguł:

- prosty przypadek użycia – zawiera do 3 kroków scenariusza, waga = 5;
- średnio-złożony przypadek użycia – zawiera od 4 do 7 kroków scenariusza, waga = 10;
- złożony przypadek użycia – zawiera powyżej 8 kroków scenariusza, waga = 15;

Przypadki użycia, pod względem liczby zaangażowanych klas, grupuje się według następujących reguł:

- prosty przypadek użycia – używa do 4 klas, waga = 5;
- średnio-złożony przypadek użycia – używa od 5 do 10 klas, waga = 10;
- złożony przypadek użycia – używa powyżej 11 klas, waga = 15;

Sumę nieskorygowanych wag przypadków użycia (*Unadjusted Use Case Weight* – UUCW) oblicza się poprzez dodanie ilości wszystkich przypadków użycia, pomnożonych przez wielkość odpowiedniego współczynnika wag.

Po analizie aktorów i przypadków użycia należy obliczyć nieskorygowane punkty przypadków użycia (*Unadjusted Use Case Points* – UUCP), według wzoru:

$$UUCP = UAW + UUCW$$

Następnym krokiem algorytmu jest uwzględnienie czynników technicznych i środowiskowych. Procedura ta została zaadaptowana bezpośrednio z algorytmu FPA, gdzie każdy czynnik jest oceniany w skali od 0 do 5 (0 –brak wpływu, zaś 5 – bardzo duży wpływ) i mnożony przez odpowiednią wagę. Czynniki techniczne, które sąbrane pod uwagę przedstawione są w Tabl. 13-2.

Tabl. 13-2. Czynniki techniczne i ich wagi, uwzględnianie przy obliczaniu TF

Czynnik techniczny	Waga
System rozproszony	2
Wymogi dot. czasu reakcji lub przepustowości	1

Sprawność działania po stronie użytkownika końcowego	1
Złożone wewnętrzne przetwarzanie	1
Wymagana ponowna używalność kodu	1
Łatwość instalacji	0,5
Łatwość użytkowania	0,5
Przenośność	2
Łatwość zmiany	1
Współbieżność procesów	1
Specjalne mechanizmy ochrony dostępu	1
Konieczność bezpośredniego udostępniania systemom zewnętrznym	1
Konieczność specjalnego przeszkolenia użytkowników	1

Suma iloczynów ocen czynników technicznych pomnożonych przez ich wagi nazywana jest wspólnikiem technicznym (*Technical Factor – TF*). Współczynnik złożoności technicznej (*Technical Complexity Factor TCF*) obliczany jest wg następującego wzoru:

$$TCF = 0,6 + (0,01 * TF)$$

Podobnie jak czynniki techniczne traktowane są czynniki środowiskowe. W Tabl. 13-3 przedstawione są czynniki środowiskowe, które sąbrane pod uwagę ora przypisane do nich wagi.

Tabl. 13-3. Czynniki środowiskowe i ich wagi, uwzględnianie przy obliczaniu EF

Czynnik środowiskowy	Waga
Znajomość metodyki i procesu iteracyjno-przyrostowego	1,5
Doświadczenie zespołu w dziedzinie zastosowań tworzonego systemu	0,5
Doświadczenie w zakresie technik obiektowych	1
Kwalifikacje głównego analityka	0,5
Motywacja uczestników	1
Stabilność wymagań	2
Udział pracowników z zewnątrz	-1
Trudny język oprogramowania	-1

Suma iloczynów ocen czynników środowiskowych pomnożonych przez ich wagi nazywana jest współczynnikiem środowiskowym (*Environmental Factor – EF*). Współczynnik złożoności środowiskowej (*Environmental Complexity Factor ECF*) obliczany jest wg następującego wzoru:

$$ECF = 1,4 + (-0,03 * EF)$$

Ostatnim krokiem, który daje oszacowanie oprogramowania, jest obliczenie skorygowanych punktów przypadków użycia (*Adjusted Use Case Points – AUCP*). Do obliczenia tej wielkości wykorzystuje się następujący wzór:

$$UCP = UUCP * TCF * ECF$$

Aby podać pracochłonność przedsięwzięcia, należy przeliczyć punkty uzyskane metodą UCP na osobogodziny. W tym celu należy pomnożyć wynik UCP przez współczynnik produktywności (*Productivity Factor PF*), zdefiniowany jako liczba osobogodzin przypadająca na jeden punkt przypadków użycia. Z wartości podawanych w literaturze wynika, że współczynnik ten może wawać się od 15 do 38 roboczych godzin na jeden punkt przypadków użycia.

13.5. Ponowne wykorzystanie oprogramowania

Obok poprawy procesu wytwarzania oraz czynności związanych z używaniem narzędzi, które automatyzują ten proces, sposobem na zwiększenie produktywności wytwarzania oprogramowania jest wielokrotne używanie zasobów oprogramowania (Rys. 13.6). Istotą ponownego wykorzystania, w kontekście inżynierii oprogramowania, jest użycie jednego produktu (lub pewnych jego elementów) do wytworzenia innego produktu. Warto zauważyć, że zasadę ponownego użycia możemy zastosować nie tylko do namacalnych części systemu oprogramowania, ale także do pomysłów i doświadczeń jego twórców.

Właściwie każdy kto kiedykolwiek programował w języku wyższego poziomu (np. Pascal, C, Java) miał do czynienia z ponownym wykorzystaniem oprogramowania. Pisząc programy często wykorzystuje się biblioteki, które dostarczają pewnych funkcjonalności. Przykładem mogą być aplikacji z graficznym interfejsem graficznym, gdzie wykorzystywane są biblioteki zawierające metody manipulacji grafiką na ekranie komputera, programy obliczeniowe, gdzie wykorzystywane są biblioteki zawierające metody oferujące skomplikowane działania matematyczne, programy do przetwarzania danych, gdzie wykorzystuje się biblioteki zawierające odpowiednie struktury danych i metody analizy.



Rys. 13.6. Ponowne wykorzystywanie zasobów oprogramowania

Wykorzystując tego typu biblioteki, które pochodzą z autoryzowanych źródeł, programista ma pewność, że oferowana przez nie funkcjonalność spełnia wymagania (wejście – wyjście) oraz że wszystkie operacje wykonywane są poprawnie. W taki sposób można wykorzystać zasoby oprogramowania na każdym poziomie abstrakcji od wymagań, przez architekturę i projekt szczegółowy do wspomnianego wyżej kodu.

13.5.1. Modułowe podejście do budowy oprogramowania

Tworząc oprogramowanie zgodnie z koncepcją modułowej budowy oprogramowania, twórcy mogą zasoby oprogramowania, które na wzór bibliotek będą mogły być ponownie wykorzystane. Elementy, które będą podlegać ponownemu wykorzystaniu, dalej będziemy nazywać zasobami oprogramowania. Aby efektywnie wykorzystać zasoby oprogramowania, powinny one spełniać pewne wymagania. Oto główne z nich:

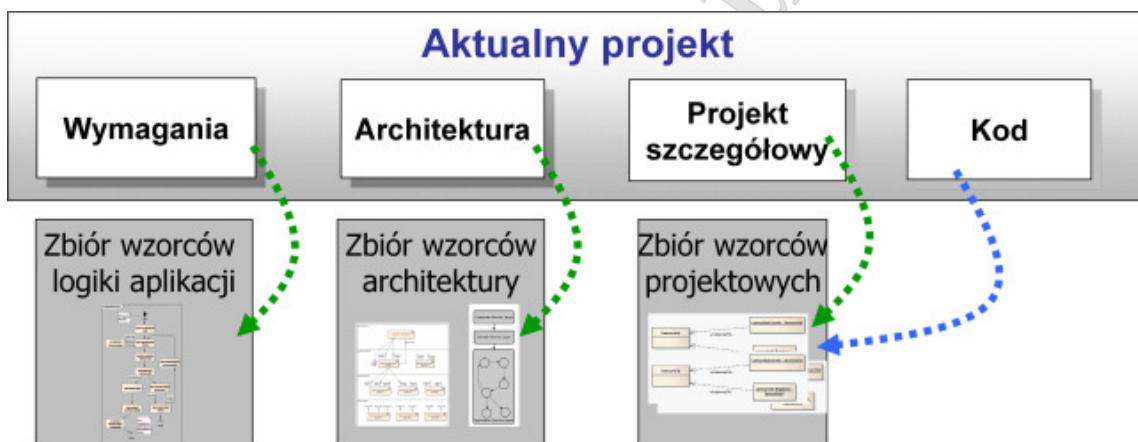
- generyczność,
- hermetyczność - maksymalnie wyizolowany z otoczenia,
- spójność i kompletność,
- niezawodność,
- odporność na błędy i wyjątki,
- dobra dokumentacja.

Cechy te można łatwo osiągnąć, poprzez jednoznaczny podział budowanego oprogramowania na moduły. Mogą to być np. komponenty, serwisy. Można wyróżnić trzy podstawowe typy modeli modułowych zasobów oprogramowania, które można ponownie wykorzystać [Szejko].

- Model czarnej skrzynki - najbardziej pożądany przy ponownym użyciu, najtrudniejszy w opracowaniu, używa się go najczęściej poprzez odsyłacze, lub kopowanie.
- Model szklanej skrzynki - budowa i cech zewnętrzne są widoczne, ale nie można ich zmienić. Poznanie właściwości pomaga w zrozumieniu funkcjonowania.
- Model białej skrzynki - użytkownik widzi strukturę i może ją dowolnie modyfikować, jest najłatwiejszy do wdrożenia. Użycie białej skrzynki polega na skopiowaniu i modyfikowaniu.

13.5.2. Wzorce oprogramowania

Omawiane do tej pory sposoby ponownego użycia oprogramowania sprowadzało się do wykorzystywania pewnych funkcji lub zespołu funkcji. Można jednak wykorzystywać zasoby innego typu – wzorce oprogramowania. Są one formą reprezentacji wiedzy i doświadczenia twórców oprogramowania w postaci opisu często powtarzającego się problemu występującego w tworzeniu oprogramowania oraz jego rozwiązania, które może zostać wielokrotnie wykorzystane [Szej02]. Wzorce oprogramowania w swojej definicji są zgodne z definicją dla wzorców konstruowania budynków podanej przez Alexandra. Spośród ogromnej liczby wzorców oprogramowania, poniżej zostaną przedstawione wybrane trzy najbardziej charakterystyczne ich grupy, które są przedstawione na Rys. 13.7.



Rys. 13.7. Wykorzystanie wzorców oprogramowania

Wzorce projektowe. Najczęściej tworzone są na potrzeby projektowania obiektowego. Służą one do przedstawienia uogólnionego rozwiązania problemów pewnej klasy, poprzez wskazanie zależności między elementami projektowanymi (w przypadku projektowania obiektowego będą to klasy, interfejsy, klasy abstrakcyjne), typów zależności, oraz rodzajów operacji wykonywanych na elementach projektowych. Wzorzec projektowy, tak jak inne wzorce, nigdy nie oferuje konkretnego rozwiązania. Wzorce projektowe odgrywają ważną rolę w fazie projektowania (dostarczając sprawdzonych rozwiązań projektowych) oraz w fazie implementacji (wyznaczając kierunek tworzenia najbardziej optymalnego kodu). Stosowanie wzorców projektowych pozwala standaryzować kod. Dzięki temu jest on zrozumiały, bardziej optymalny i mniej zawodny.

Na przestrzeni lat zostało odkrytych i zdefiniowanych bardzo wiele wzorców projektowych. Podstawowe informacje na temat pierwotnej ich klasyfikacji zaproponowana przez „Bandę Czworga” znajdują się w rozdziale 8.

Wzorce architektury. Dzięki nim można określić ogólną strukturę systemu oprogramowania, czyli elementy z jakich się składa, zakres funkcjonalności realizowany przez dany element oraz zasady komunikacji pomiędzy poszczególnymi elementami. Idea wzorców architektonicznych jest podobna do idei wzorców projektowych, z tą różnicą iż wzorce architektoniczne nie dotyczą elementów projektowych lecz całego systemu, bądź jego modułów.

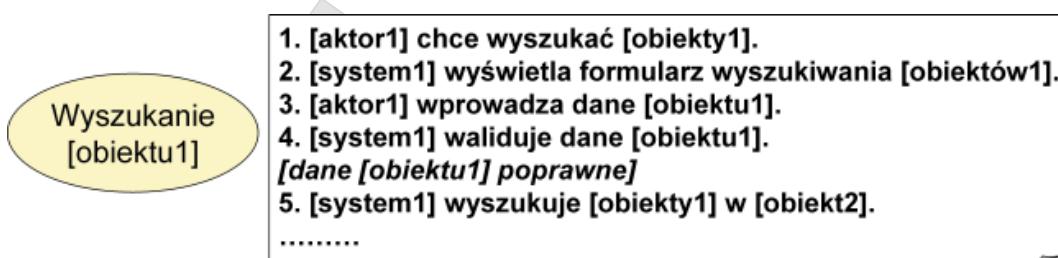
Popularne wzorce architektury to między innymi: architektura trójwarstwowa, architektura klient-serwer, model widok-kontroler (MVC), architektura zorientowana na usługi (SOA), Peer-to-peer (P2P). Omówienie przykładowych stylów architektonicznych znajduje się w rozdziale 8.

Wzorce logiki aplikacji. Logika aplikacji określa w jakie czynności należy wykonać, aby osiągnąć zamierzony cel. Rozważmy przykład wyszukania pojazdu w bazie pojazdów (Rys. 13.8). Na samym początku użytkownik powinien zainicjować akcję wyszukiwania. Następnie, po wyświetleniu formularza, powinien wprowadzić dane pojazdu, który chce znaleźć. Po potwierdzeniu poprawności danych pojazdu, system przystępuje do wyszukiwania pojazdu.



Rys. 13.8. Grupy podobnych przypadków użycia

Wszystkie czynności zamknięte są w scenariuszu przypadku użycia. Inne operacje związane z wyszukiwaniem (np. książek, klientów, produktów) będą charakteryzować się podobną (lub nawet taką samą) sekwencją kroków. Na tej podstawie można wyróżnić grupy charakterystycznych przypadków użycia, do których opisu wykorzystywane są scenariusze o takiej samej strukturze i podobnych czynnościach. Na tej podstawie definiujemy wzorce logiki aplikacji, które możemy także nazwać wzorcami scenariuszy przypadków użycia, gdzie operacje oddzielone są od dziedziny (Rys. 13.9).



Rys. 13.9. Wzorzec scenariusza przypadku użycia

Zastosowanie wzorców scenariuszy umożliwia tworzenie specyfikacji wymagań mniejszym nakładem pracy poprzez stosowanie sprawdzonych rozwiązań i ujednolicenie sposobu jej tworzenia.

13.5.3. Przechowywanie zasobów oprogramowania

Aby sprawnie móc wykorzystywać zasoby oprogramowania, potrzebne jest miejsce, z którego potrzebne zasoby będzie się pobierać. Do przechowywania, zarządzania, udostępniania zgromadzonych danych wykorzystuje się repozytoria zasobów oprogramowania. Główna zasada działanie repozytorium zasobów oprogramowania jest taka sama jak repozytoriów wykorzystywanych w przedsięwzięciach budowy oprogramowania (patrz rozdział 10, sekcja 3). Dodatkowo repozyto-

rium zasobów oprogramowania powinno posiadać kilka cech charakterystycznych, które będą pomocne w czynnościach związanych z ponownym wykorzystaniem:

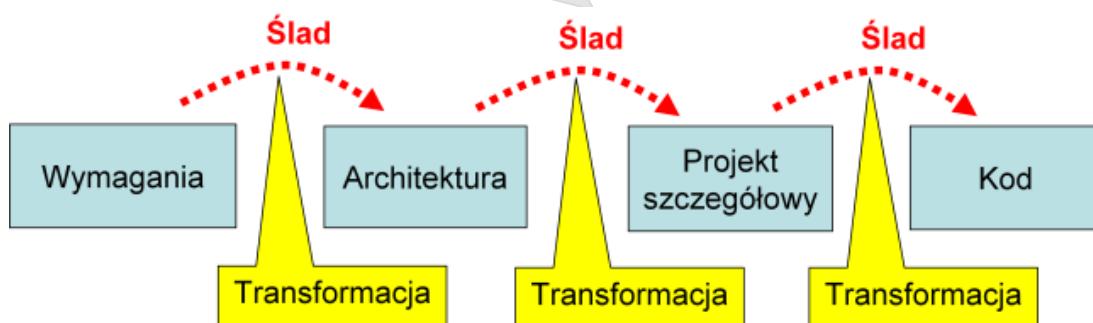
- przechowywania zasobów różnych typów: dokumenty, specyfikacje wymagań, modele architektoniczne i projektowe, kod, wzorce oprogramowania;
- możliwość definiowania struktury pozwalającej grupować zasoby;
- mechanizm przeglądania i filtrowania zbiorów zasobów;
- mechanizm wyszukiwania zasobów według zadanych kryteriów, takich jak np. nazwa, typ, obszar zastosowania, problem, dla którego szukane jest rozwiązanie.

Proces wypełniania repozytorium zasobami gotowymi do ponownego użycia rozpoczyna się od dodania tam pierwszych elementów (zbiory wzorców oprogramowania, otwarte projekty open source, zbiory modułów i bibliotek) i właściwie nigdy się nie kończy. Oferta zasobów wielokrotnego użycia cały czas jest wzbogacana zasobami pochodzący z prowadzonych lub zakończonych projektów.

13.5.4. Ponowne wykorzystanie oprogramowania sterowane wymaganiami

Przedstawione wyżej metody ponownego wykorzystania zasobów oprogramowania sprowadzały się do ponownego użycia pewnych konkretnych elementów, które po zaadaptowaniu ułatwiają rozwiązanie pewnego problemu. Sposobem na wykorzystanie oprogramowania, dzięki któremu pokryjemy całą ścieżkę budowy oprogramowania (od wymagań do kodu), jest wykorzystanie oprogramowania sterowane wymaganiami. Mechanizm ten, poprzez wskazanie odpowiadającego potrzebom twórców zbioru wymagań, pozwala na ponowne wykorzystanie pełnej ich realizacji.

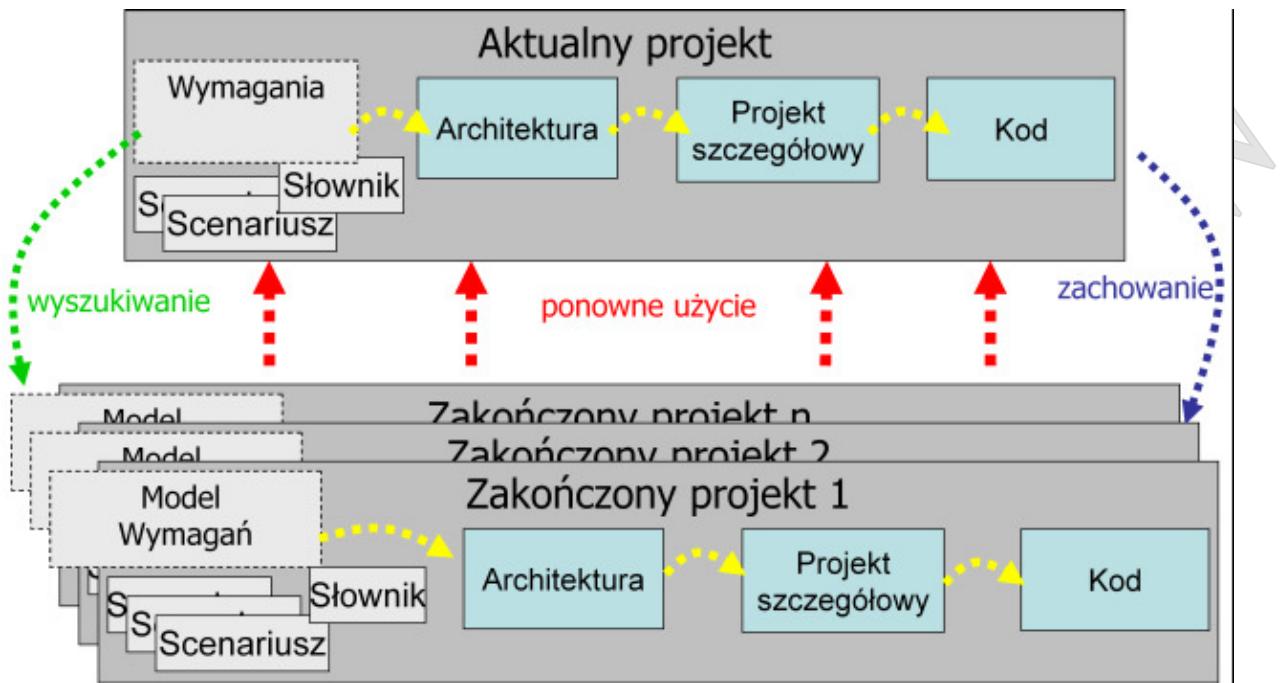
Aby takie rozwiązanie było możliwe, należy powiązać wymagania z odpowiednimi elementami architektury (np. komponenty i interfejsy), następnie z projektami szczegółowymi i na koniec z elementami kodu. Aby zautomatyzować proces wiązania elementów kolejnych produktów ze sobą, można użyć transformacji, które zgodnie z koncepcją MDA tworzą modele na niższych poziomach abstrakcji. Schemat automatycznego tworzenia śladów przedstawiony jest na Rys. 13.10.



Rys. 13.10. Automatyczne tworzenie śladów wspomagane koncepcją MDA

Wybrany zbiór wymagań pewnego systemu wraz z jego realizacją tworzą wycinek oprogramowania, który może być wykorzystany w aktualnym projekcie.

Repozytorium, z którego można pobrać wycinki oprogramowania, może być zwykłym repozytorium zawierającym zbiór zatwierdzonych do ponownego użycia kompletnych systemów oprogramowania. Schemat ponownego wykorzystania sterowanego wymaganiami przedstawiony jest na Rys. 13.11.



Rys. 13.11. Schemat ponownego wykorzystanie oprogramowania sterowanego wymaganiami

Znalezienie w repozytorium systemów odpowiadających kryteriom wyszukiwania w postaci fragmentu specyfikacji wymagań bieżącego systemu wymaga przejrzenia specyfikacji wymagań wszystkich znajdujących się tam systemów oprogramowania. Przy wyszukiwaniu brane są pod uwagę scenariusze przypadków użycia oraz słownik dziedziny. Pojęcia słownika są porównywane pod względem podobieństw leksykalnych i semantycznych. Przy porównywaniu scenariusze brane są pod uwagę dodatkowo podobieństwa strukturalne.

Po wykonaniu zapytania i wyborze odpowiedniego wycinka oprogramowania, jest on włączany do bieżącego systemu. Od wyboru twórców oprogramowania zależy czy wykorzystają cały wycinek, czy tylko pewne jego elementy. W miarę potrzeb może on być dostosowany do kształtu bieżącego systemu (np. modyfikacja fragmentów wymagań, bądź projektu szczegółowego).

Bieżący system oprogramowania, po zakończeniu prac nad jego rozwojem, można dołączyć do repozytorium zasobów oprogramowania. Wzbogaci on zbiór gotowych rozwiązań, które będzie można wykorzystać w kolejnych projektach.

Pomimo iż, przedstawiony sposób wykorzystanie oprogramowania jest rozwiązaniem nowatorskim, to zostało już zrealizowane w praktyce w ramach projektu ReDSeeDS (WWW.redseeds.eu). Wykazało ono:

- opracowania sposobu definiowania wymagań w sposób oddzielający opis dynamiki od statyki systemu,
- implementacji miar podobieństwa i mechanizmu znajdowania podobieństw,
- prezentacji wyników zapytań,
- realizacji mechanizmów włączania wycinków oprogramowania.

13.5.5. Proces wytwórczy wzbogacony o ponowne wykorzystanie oprogramowania

Decydując się na ponowne wykorzystanie oprogramowania w trakcie tworzenia nowego systemu oprogramowania, należy przede wszystkim odpowiednio skonfigurować mechanizmy, które pozwolą na efektywne wykorzystanie istniejących zasobów. Proces wytwórczy powinien zostać wzbogacony czynnościami umożliwiającymi obsługę repozytorium zasobów oprogramowania oraz mechanizmami wybierania zasobów i włączania ich do bieżącego systemu. Ponadto kadra zarządzająca powinna wpierać koncepcję ponownego wykorzystania oprogramowania i zachęcać do niej swoich pracowników. Przystosowanie procesu wytwórczego do koncepcji po-

nownego wykorzystania oprogramowania wymaga przezwyciężenia barier organizacyjnych i technicznych. W procesie twórczym powinny pojawić się nowe role, które będą odpowiedzialne za konstruowanie zasobów oprogramowania, zarządzanie repozytorium, specjalistę od adaptowania zasobów, kierownika procesu ponownego użycia oprogramowania.

Każda z przedstawionych w tym rozdziale technik ponownego wykorzystania oprogramowania ma pewne zalety i wady. Ich zrównoważenie zależy specyfiki działania danej organizacji i od typu projektów, które są przez tą organizację realizowane. Twórcy oprogramowania powinni wypracować własne techniki, które wzbogacą proces twórczy realizujący koncepcje ponownego użycia oprogramowania poprzez zmianę środowiska wytwarzania, zespołu, organizacji i celów biznesowych firmy. W zależności od potrzeb, ponownemu wykorzystaniu mogą ulegać dowolne produkty procesu twórczego (wzory dokumentacji, wyniki analizy dziedziny problemu, specyfikacje wymagań na system, architektury, wzorce oprogramowania, składniki oprogramowania, procedury, plany testów, formularze kontroli jakości, materiały szkoleniowe)

Takie podejście do wytwarzania oprogramowania, ukierunkowuje je na obniżenie kosztów wytwarzania, skrócenie czasu i podwyższenie jakości wytwarzanego oprogramowania. Należy jednak wykazać się cierpliwością, ponieważ dodatkowe koszty poniesione na wdrożenie idei ponownego użycia oprogramowania przyniosą korzyści dopiero po pewnym czasie.

13.6. Podsumowanie

Rozdział ten stawia inżynierię oprogramowania na równi z innymi dziedzinami inżynierijnymi, wskazując na główne wspólne elementy. Są to: zarządzanie przebiegiem projektu, wdrażanie metod wytwarzania, organizację i zarządzanie pracą zespołów twórczych, szacowanie złożoności procesów twórczych, mechanizmy ponownego wykorzystania.

Choć wymienione wyżej elementy występują we wszystkich przedsięwzięciach inżynierijnych, to objęte nimi czynności związane z wytwarzaniem oprogramowania, wykazują pewną specyfikę. Ze względu na nieuchwytność i złożoność produktu, jakim jest system oprogramowania, utrudnione jest ocenianie postępu prac, wydzielanie elementów, które mogą być wykorzystane ponownie w kolejnych przedsięwzięciach, wycenianie procesu wytwarzania. W odróżnieniu od innych inżynierijnych dziedzin, gdzie głównymi zasobami zwykle są ludzie, materiały i narzędzia, w inżynierii oprogramowania głównymi zasobami są ludzie, ich wiedza, doświadczenie oraz sposoby ich wykorzystania. Czyni to inżynierię oprogramowania dziedziną niezwykle dynamicznie się rozwijającą i zmienną.