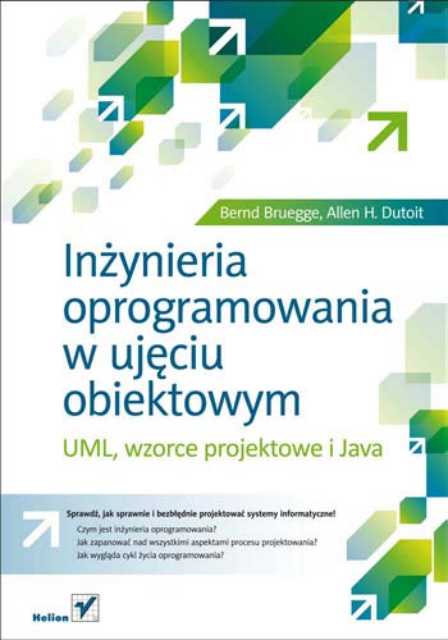




Bernd Bruegge, Allen H. Dutoit

Inżynieria oprogramowania w ujęciu obiektowym

UML, wzorce projektowe i Java



Sprawdź, jak sprawnie i bezbłędnie projektować systemy informatyczne!

Czym jest inżynieria oprogramowania?

Jak zapanować nad wszystkimi aspektami procesu projektowania?

Jak wygląda cykl życia oprogramowania?

Helion



» Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

Inżynieria oprogramowania w ujęciu obiektowym. UML, wzorce projektowe i Java

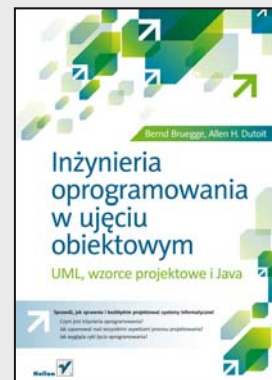
Autorzy: Bernd Bruegge, Allen H. Dutoit

Tłumaczenie: Andrzej Grażyński

ISBN: 978-83-246-2872-8

Tytuł oryginału: [Object-Oriented Software Engineering Using UML, Patterns, and Java \(3rd Edition\)](#)

Format: 172×245, stron: 872



Sprawdź, jak sprawnie i bezbłędnie projektować systemy informatyczne!

- Czym jest inżynieria oprogramowania?
- Jak zapanować nad wszystkimi aspektami procesu projektowania?
- Jak wygląda cykl życia oprogramowania?

Projektowanie systemów informatycznych to zadanie bardzo skomplikowane. Ogromna liczba zależności, zasad i wyjątków od nich sprawia, że nie jest możliwe podejście do tego zadania ot tak, z marszu. Zbieranie i analiza wymagań, przygotowanie diagramów klas, aktywności, stanów czy interakcji to tylko część etapów, z którymi musi poradzić sobie projektant. Jeżeli nałożyć na to wszystko wzorce projektowe, stajemy przed prawie nierozwiązywalnym zadaniem.

Dzięki tej książce dowiesz się, jak sprostać temu karkołomnemu zadaniu! Poznasz język UML, który wprowadził porządek w tym skomplikowanym procesie, oraz podstawowe koncepcje inżynierii oprogramowania. Nauczysz się zarządzać procesem tworzenia oprogramowania, zbierać oraz analizować wymagania, identyfikować podsystemy, specyfikować interfejsy oraz testować. Znajdziesz tu wszystko na temat zarządzania zmianami. W trakcie lektury sprawdzisz, jak wygląda cykl życia oprogramowania oraz jak zarządzać konfiguracją. Dodatkowo poznasz metodologię działań, które doprowadzą Cię do wyznaczonego celu. Książka ta stanowi obowiązkową pozycję dla każdego projektanta oraz analityka, także programiści również znajdą tu wiele cennych wskazówek!

- Niepowodzenia w inżynierii oprogramowania
- Podstawowe koncepcje inżynierii oprogramowania
- Modelowanie przy użyciu języka UML
- Organizacja projektu
- Narzędzie do komunikacji grupowej
- Proces zbierania wymagań
- Identyfikacja aktorów, scenariuszy oraz przypadków użycia
- Określanie obiektów modelu analitycznego
- Analiza wymagań
- Dekompozycja systemu na podsystemy
- Identyfikacja celów projektowych
- Projektowanie obiektów
- Wzorce projektowe
- Specyfikowanie interfejsów
- Odzworowywanie modelu na kod
- Testowanie
- Zarządzanie zmianami i konfiguracją
- Cykl życia oprogramowania
- Metodologie

Dobry projekt systemu to podstawa sukcesu!

Spis treści

Przedmowa	17
Wstęp	19
Podziękowania	31

CZĘŚĆ I	Zaczynamy	33
----------------	------------------	-----------

Rozdział 1.	Wprowadzenie do inżynierii oprogramowania	35
1.1.	Wprowadzenie: niepowodzenia w inżynierii oprogramowania	36
1.2.	Czym jest inżynieria oprogramowania?	38
1.2.1.	Modelowanie	38
1.2.2.	Rozwiązywanie problemów	40
1.2.3.	Pozyskiwanie wiedzy	41
1.2.4.	Racjonalizacja	42
1.3.	Podstawowe koncepcje inżynierii oprogramowania	43
1.3.1.	Uczestnicy i role	44
1.3.2.	Systemy i modele	46
1.3.3.	Produkty	46
1.3.4.	Aktywności, zadania i zasoby	47
1.3.5.	Wymagania funkcyjne i pozafunkcyjne	48
1.3.6.	Notacje, metody i metodologie	48
1.4.	Aktywności inżynierii oprogramowania	49
1.4.1.	Zbieranie wymagań	50
1.4.2.	Analiza	51
1.4.3.	Projekt systemu	51
1.4.4.	Projektowanie obiektów	53
1.4.5.	Implementowanie	53
1.4.6.	Testowanie	54
1.5.	Zarządzanie tworzeniem oprogramowania	54
1.5.1.	Komunikacja	55
1.5.2.	Zarządzanie racjonalizacją	55
1.5.3.	Zarządzanie konfiguracją oprogramowania	56
1.5.4.	Zarządzanie projektem	56
1.5.5.	Cykl życiowy oprogramowania	56
1.5.6.	Podsumowanie	57

1.6.	Analiza przypadku — system ARENA	57
1.7.	Literatura uzupełniająca	58
1.8.	Ćwiczenia	59
Rozdział 2.	Modelowanie w języku UML	63
2.1.	Wprowadzenie	64
2.2.	Ogólnie o UML	65
2.2.1.	Diagramy przypadków użycia	65
2.2.2.	Diagramy klas	65
2.2.3.	Diagramy interakcji	67
2.2.4.	Diagram stanów	67
2.2.5.	Diagramy aktywności	68
2.3.	Podstawowe koncepcje modelowania	69
2.3.1.	Systemy, modele i widoki	69
2.3.2.	Typy danych, abstrakcyjne typy danych i instancje	72
2.3.3.	Klasy, klasy abstrakcyjne i obiekty	73
2.3.4.	Klasy zdarzeniowe, zdarzenia i komunikaty	75
2.3.5.	Modelowanie zorientowane obiektowo	76
2.3.6.	Falsyfikacja i prototypowanie	77
2.4.	UML — głębszy wgląd	78
2.4.1.	Diagramy przypadków użycia	79
2.4.2.	Diagramy klas	86
2.4.3.	Diagramy interakcji	95
2.4.4.	Diagramy stanów	98
2.4.5.	Diagramy aktywności	101
2.4.6.	Organizacja diagramów	104
2.4.7.	Rozszerzenia diagramów	106
2.5.	Literatura uzupełniająca	107
2.6.	Ćwiczenia	108
Rozdział 3.	Organizacja projektu i komunikacja	113
3.1.	Wstęp — katastrofa Ariane	114
3.2.	O projekcie ogólnie	115
3.3.	Koncepcje organizacyjne projektu	119
3.3.1.	Organizacja projektów	119
3.3.2.	Role w realizacji projektu	122
3.3.3.	Zadania i produkty	124
3.3.4.	Harmonogramy	126
3.4.	Koncepcje komunikacyjne projektu	128
3.4.1.	Komunikacja planowa	128
3.4.2.	Komunikacja pozaplanowa	135
3.4.3.	Mechanizmy komunikacyjne	138
3.5.	Aktywności organizacyjne	146
3.5.1.	Dołączanie do zespołu	146
3.5.2.	Dołączanie do infrastruktury komunikacyjnej	146

3.5.3.	Udział w zebraniach zespołu	147
3.5.4.	Organizacja przeglądów	149
3.6.	Literatura uzupełniająca	151
3.7.	Ćwiczenia	152

CZĘŚĆ II	Zmagania ze złożonością	155
-----------------	--------------------------------	------------

Rozdział 4.	Zbieranie wymagań	157
4.1.	Wstęp: przykłady problemów z użytecznością	158
4.2.	O zbieraniu wymagań ogólnie	159
4.3.	Koncepcje zbierania wymagań	161
4.3.1.	Wymagania funkcyjne	161
4.3.2.	Wymagania pozafunkcyjne	162
4.3.3.	Kompletność, spójność, jednoznaczność i poprawność	164
4.3.4.	Realizm, weryfikowalność i identyfikowalność	165
4.3.5.	Inżynieria pierwotna, inżynieria wtórna i inżynieria interfejsu	165
4.4.	Aktywności związane ze zbieraniem wymagań	166
4.4.1.	Identyfikacja aktorów	167
4.4.2.	Identyfikacja scenariuszy	169
4.4.3.	Identyfikacja przypadków użycia	171
4.4.4.	Doskonalenie przypadków użycia	173
4.4.5.	Identyfikacja relacji między aktorami a przypadkami użycia	176
4.4.6.	Początkowa identyfikacja obiektów modelu analitycznego	179
4.4.7.	Identyfikacja wymagań pozafunkcyjnych	182
4.5.	Zarządzanie zbieraniem wymagań	183
4.5.1.	Negocjowanie specyfikacji z klientem: metoda Joint Application Design	185
4.5.2.	Zarządzanie identyfikowalnością	187
4.5.3.	Dokumentowanie zbierania wymagań	188
4.6.	Analiza przypadku — system ARENA	190
4.6.1.	Wstępna deklaracja problemu	190
4.6.2.	Identyfikacja aktorów i scenariuszy	192
4.6.3.	Identyfikacja przypadków użycia	195
4.6.4.	Doskonalenie przypadków użycia i identyfikacja relacji	198
4.6.5.	Identyfikacja wymagań pozafunkcyjnych	204
4.6.6.	Wnioski	204
4.7.	Literatura uzupełniająca	205
4.8.	Ćwiczenia	207

Rozdział 5.	Analiza wymagań	211
5.1.	Wstęp: złudzenie optyczne	212
5.2.	O analizie wymagań ogólnie	212
5.3.	Koncepcje analizy wymagań	214
5.3.1.	Analityczny model obiektowy i modele dynamiczne	214
5.3.2.	Obiekty encji, obiekty brzegowe i obiekty sterujące	215
5.3.3.	Generalizacja i specjalizacja	216
5.4.	Aktywności analizy wymagań:	
	od przypadków użycia do obiektów	217
5.4.1.	Identyfikacja obiektów encji	218
5.4.2.	Identyfikacja obiektów brzegowych	220
5.4.3.	Identyfikacja obiektów sterujących	222
5.4.4.	Odwzorowywanie przypadków użycia w obiekty za pomocą diagramów sekwencji	224
5.4.5.	Modelowanie interakcji między obiektami za pomocą kart CRC	228
5.4.6.	Identyfikacja skojarzeń	228
5.4.7.	Identyfikacja agregacji	231
5.4.8.	Identyfikacja atrybutów	232
5.4.9.	Modelowanie zachowania poszczególnych obiektów uzależnionego od ich stanu	233
5.4.10.	Modelowanie relacji dziedziczenia między obiektami	234
5.4.11.	Przeglądy modelu analitycznego	235
5.4.12.	Podsumowanie analizy	236
5.5.	Zarządzanie analizą wymagań	237
5.5.1.	Dokumentowanie analizy wymagań	238
5.5.2.	Przydzielanie odpowiedzialności	239
5.5.3.	Komunikacja w związku z analizą wymagań	240
5.5.4.	Iteracje modelu analitycznego	241
5.5.5.	Uzgodnienie modelu analitycznego z klientem	243
5.6.	Analiza przypadku — system ARENA	245
5.6.1.	Identyfikacja obiektów encji	245
5.6.2.	Identyfikacja obiektów brzegowych	250
5.6.3.	Identyfikacja obiektów sterujących	251
5.6.4.	Modelowanie interakcji między obiektami	252
5.6.5.	Weryfikacja i konsolidacja modelu analitycznego	254
5.6.6.	Wnioski	256
5.7.	Literatura uzupełniająca	258
5.8.	Ćwiczenia	258
Rozdział 6.	Projektowanie systemu — dekompozycja na podsystemy	263
6.1.	Wstęp: projekt mieszkania	264
6.2.	O projektowaniu systemu ogólnie	266
6.3.	Koncepcje projektowania systemu	267

6.3.1.	Podsystemy i klasy	268
6.3.2.	Usługi i interfejsy podsystemów	270
6.3.3.	Sprzężenie i spoistość	271
6.3.4.	Warstwy i partycje	275
6.3.5.	Style architektoniczne	279
6.4.	Aktywności projektowania systemu: od obiektów do podsystemów	288
6.4.1.	Punkt wyjścia: model analityczny systemu planowania podróży	288
6.4.2.	Identyfikowanie celów projektowych	290
6.4.3.	Identyfikowanie podsystemów	294
6.5.	Literatura uzupełniająca	296
6.6.	Ćwiczenia	297

Rozdział 7. Projekt systemu: realizacja celów projektowych **301**

7.1.	Wstęp: przykład redundancji	302
7.2.	O aktywnościach projektowania systemu ogólnie	303
7.3.	Koncepcje: diagramy wdrażania UML	304
7.4.	Aktywności realizacji celów projektowych	306
7.4.1.	Odwzorowywanie podsystemów w procesory i komponenty	306
7.4.2.	Identyfikowanie trwałych danych i ich przechowywanie	309
7.4.3.	Definiowanie założeń kontroli dostępu	312
7.4.4.	Projektowanie globalnego przepływu sterowania	319
7.4.5.	Identyfikowanie usług	321
7.4.6.	Identyfikowanie warunków granicznych	323
7.4.7.	Weryfikowanie projektu systemu	326
7.5.	Zarządzanie projektowaniem systemu	328
7.5.1.	Dokumentowanie projektu systemu	328
7.5.2.	Przydzielanie odpowiedzialności	330
7.5.3.	Komunikacja w projektowaniu systemu	331
7.5.4.	Iteracje projektowania systemu	333
7.6.	Analiza przypadku — system ARENA	334
7.6.1.	Identyfikowanie celów projektowych	335
7.6.2.	Identyfikowanie podsystemów	336
7.6.3.	Odwzorowanie podsystemów w procesory i komponenty	337
7.6.4.	Identyfikowanie i przechowywanie trwałych danych	339
7.6.5.	Definiowanie założeń kontroli dostępu	340
7.6.6.	Projektowanie globalnego przepływu sterowania	341
7.6.7.	Identyfikowanie usług	343
7.6.8.	Identyfikowanie warunków granicznych	345
7.6.9.	Wnioski	347
7.7.	Literatura uzupełniająca	348
7.8.	Ćwiczenia	348

Rozdział 8. Projektowanie obiektów:

wielokrotne wykorzystywanie rozwiązań wzorcowych	353
8.1. Wstęp: wypadki produkcyjne	354
8.2. O projektowaniu obiektów ogólnie	355
8.3. Koncepcja wielokrotnego wykorzystywania	
— dziedziczenie, delegowanie i wzorce projektowe	359
8.3.1. Obiekty aplikacyjne i obiekty realizacyjne	359
8.3.2. Dziedziczenie implementacyjne	
i dziedziczenie specyfikacyjne	360
8.3.3. Delegowanie	363
8.3.4. Zasada zastępowania Liskov	364
8.3.5. Delegowanie i dziedziczenie	
we wzorcach projektowych	364
8.4. Wybór wzorców projektowych i gotowych komponentów	367
8.4.1. Hermetyzacja przechowywania danych	
za pomocą wzorca projektowego Most	368
8.4.2. Hermetyzacja niekompatybilnych komponentów	
za pomocą wzorca projektowego Adapter	371
8.4.3. Hermetyzacja kontekstu za pomocą	
wzorca projektowego Strategia	373
8.4.4. Hermetyzacja platformy za pomocą	
wzorca projektowego Fabryka abstrakcyjna	376
8.4.5. Hermetyzacja przepływu sterowania za pomocą	
wzorca projektowego Polecenie	377
8.4.6. Hermetyzacja hierarchii za pomocą	
wzorca projektowego Kompozyt	378
8.4.7. Heurystyki wyboru wzorców projektowych	379
8.4.8. Identyfikowanie i przystosowywanie	
frameworków aplikacyjnych	381
8.5. Zarządzanie wykorzystywaniem gotowych rozwiązań	386
8.5.1. Dokumentowanie wykorzystywania	
gotowych rozwiązań	388
8.5.2. Przydzielanie odpowiedzialności	389
8.6. Analiza przypadku — system ARENA	390
8.6.1. Zastosowanie wzorca projektowego	
Fabryka abstrakcyjna	390
8.6.2. Zastosowanie wzorca projektowego Polecenie	392
8.6.3. Zastosowanie wzorca projektowego Obserwator	393
8.6.4. Wnioski	393
8.7. Literatura uzupełniająca	394
8.8. Ćwiczenia	395

Rozdział 9.	Projektowanie obiektów: specyfikowanie interfejsów	399
9.1.	Wstęp: kolej miejska i tramwaje	400
9.2.	O specyfikowaniu interfejsów ogólnie	401
9.3.	Koncepcje specyfikowania interfejsów	403
9.3.1.	Implementator, ekstender i użytkownik klasy	403
9.3.2.	Typy, sygnatury i widzialność	403
9.3.3.	Kontrakty: niezmienniki, warunki wstępne i warunki końcowe	406
9.3.4.	Język OCL (Object Constraint Language)	407
9.3.5.	Kolekcje OCL: zbiory, wielozbiory i ciągi	411
9.3.6.	Kwantyfikatory OCL: <code>forall()</code> i <code>exists()</code>	415
9.4.	Aktywności specyfikowania interfejsów	416
9.4.1.	Identyfikowanie brakujących atrybutów i operacji	417
9.4.2.	Specyfikowanie typów, sygnatur i widzialności	418
9.4.3.	Specyfikowanie warunków wstępnych i warunków końcowych	419
9.4.4.	Specyfikowanie niezmienników	421
9.4.5.	Dziedziczenie kontraktów	424
9.5.	Zarządzanie projektowaniem obiektów	425
9.5.1.	Dokumentowanie projektowania obiektów	425
9.5.2.	Przydzielanie odpowiedzialności	431
9.5.3.	Wykorzystywanie kontraktów w analizie wymagań	432
9.6.	Analiza przypadku — system ARENA	433
9.6.1.	Identyfikowanie brakujących operacji w klasach <code>TournamentStyle</code> i <code>Round</code>	434
9.6.2.	Specyfikowanie kontraktów dla klas <code>TournamentStyle</code> i <code>Round</code>	435
9.6.3.	Specyfikowanie kontraktów dla klas <code>KnockOutStyle</code> i <code>KnockOutRound</code>	438
9.6.4.	Wnioski	439
9.7.	Literatura uzupełniająca	440
9.8.	Ćwiczenia	440
Rozdział 10.	Odwzorowywanie modelu na kod	445
10.1.	Wstęp: Władca Pierścieni	446
10.2.	O odwzorowywaniu ogólnie	447
10.3.	Koncepcje odwzorowywania	448
10.3.1.	Transformowanie modelu	449
10.3.2.	Refaktoryzacja	450
10.3.3.	Inżynieria postępująca	452
10.3.4.	Inżynieria odwracająca	452
10.3.5.	Zasady transformacji	453
10.4.	Aktywności odwzorowywania	454
10.4.1.	Optymalizowanie modelu obiektowego	455
10.4.2.	Odwzorowywanie skojarzeń na kolekcje	458

10.4.3.	Odwzorowywanie kontraktów w wyjątki	465
10.4.4.	Odwzorowywanie modelu obiektowego w schematy bazy danych	469
10.5.	Zarządzanie transformacjami	475
10.5.1.	Dokumentowanie transformacji	475
10.5.2.	Przydzielanie odpowiedzialności	477
10.6.	Analiza przypadku — system ARENA	478
10.6.1.	Statystyki systemu ARENA	478
10.6.2.	Odwzorowywanie skojarzeń na kolekcje	480
10.6.3.	Odwzorowywanie kontraktów w wyjątki	482
10.6.4.	Odwzorowywanie modelu obiektowego w schemat bazy danych	484
10.6.5.	Wnioski	485
10.7.	Literatura uzupełniająca	485
10.8.	Ćwiczenia	486
Rozdział 11.	Testowanie	491
11.1.	Wstęp: testowanie wahadłowców	492
11.2.	O testowaniu ogólnie	494
11.3.	Koncepcje związane z testowaniem	498
11.3.1.	Usterki, błędne stany i awarie	500
11.3.2.	Przypadki testowe	503
11.3.3.	Namiaszki testowe i sterowniki testowe	505
11.3.4.	Poprawki	505
11.4.	Aktywności związane z testowaniem	506
11.4.1.	Inspekcja komponentu	507
11.4.2.	Testowanie użyteczności	508
11.4.3.	Testowanie jednostkowe	510
11.4.4.	Testowanie integracyjne	519
11.4.5.	Testowanie systemu	526
11.5.	Zarządzanie testowaniem	531
11.5.1.	Planowanie testów	532
11.5.2.	Dokumentowanie testowania	532
11.5.3.	Przydzielanie odpowiedzialności	536
11.5.4.	Testowanie regresyjne	537
11.5.5.	Automatyzacja testowania	538
11.5.6.	Testowanie bazujące na modelach	539
11.6.	Literatura uzupełniająca	541
11.7.	Ćwiczenia	543
CZĘŚĆ III	Zarządzanie zmianami	547
Rozdział 12.	Zarządzanie racjonalizacją	549
12.1.	Wstęp: przycinanie wędzonej szynki	550
12.2.	O racjonalizacji ogólnie	551

12.3.	Koncepcje racjonalizacji	554
12.3.1.	CTC — system centralnego sterowania ruchem	555
12.3.2.	Definiowanie problemów: zagadnienia	556
12.3.3.	Eksploracja przestrzeni rozwiązań: propozycje	557
12.3.4.	Wartościowanie elementów przestrzeni rozwiązań: kryteria i argumenty	559
12.3.5.	Kolapsacja przestrzeni rozwiązań: rozstrzygnięcie	560
12.3.6.	Implementowanie rozstrzygnięć: elementy działania	561
12.3.7.	Przykłady modeli zagadnień i ich realizacji	562
12.4.	Aktywności racjonalizacji — od zagadnień do decyzji	567
12.4.1.	Projekt systemu CTC	568
12.4.2.	Kolekcjonowanie racjonalizacji w ramach zebrań	569
12.4.3.	Asynchroniczne kolekcjonowanie racjonalizacji	577
12.4.4.	Racjonalizacja dyskutowanych zmian	579
12.4.5.	Rekonstruowanie racjonalizacji	582
12.5.	Kierownicze aspekty zarządzania racjonalizacją	585
12.5.1.	Dokumentowanie racjonalizacji	585
12.5.2.	Przypisywanie odpowiedzialności	587
12.5.3.	Heurystyki komunikowania racjonalizacji	588
12.5.4.	Modelowanie i negocjowanie zagadnień	589
12.5.5.	Strategie rozwiązywania konfliktów	590
12.6.	Literatura uzupełniająca	592
12.7.	Ćwiczenia	593

Rozdział 13. Zarządzanie konfiguracją 597

13.1.	Wstęp: samoloty	598
13.2.	O zarządzaniu konfiguracją ogólnie	600
13.3.	Koncepcje zarządzania konfiguracją	602
13.3.1.	Elementy konfiguracji i agregaty CM	603
13.3.2.	Wersje i konfiguracje	604
13.3.3.	Żądania zmian	604
13.3.4.	Promocje i emisje	605
13.3.5.	Repozytoria i przestrzenie robocze	606
13.3.6.	Schematy identyfikowania wersji	606
13.3.7.	Zmiany i zbiory zmian	608
13.3.8.	Narzędzia wspomagające zarządzanie konfiguracją	610
13.4.	Aktywności tworzące zarządzanie konfiguracją	611
13.4.1.	Identyfikowanie elementów konfiguracji i agregatów CM	613
13.4.2.	Zarządzanie promocjami	615
13.4.3.	Zarządzanie emisjami	616
13.4.4.	Zarządzanie gałęziami	619
13.4.5.	Zarządzanie wariantami	623
13.4.6.	Zarządzanie propozycjami zmian i ich implementowaniem	626

13.5.	Kierownicze aspekty zarządzania konfiguracją	627
13.5.1.	Dokumentowanie zarządzania konfiguracją	627
13.5.2.	Przypisywanie odpowiedzialności	628
13.5.3.	Planowanie aktywności w ramach zarządzania konfiguracją	629
13.5.4.	Integracja ciągła jako optymalizacja zarządzania promocjami i ich testowaniem	630
13.6.	Literatura uzupełniająca	632
13.7.	Ćwiczenia	633

Rozdział 14. Zarządzanie projektem 637

14.1.	Wstęp: uruchomienie misji STS-51L	638
14.2.	O zarządzaniu projektem ogólnie	639
14.3.	Koncepcje zarządzania projektem	646
14.3.1.	Zadania i aktywności	646
14.3.2.	Produkty, pakiety pracy i role	646
14.3.3.	Struktura podziału pracy	648
14.3.4.	Model zadań	649
14.3.5.	Macierz kwalifikacji	650
14.3.6.	Plan zarządzania projektem	651
14.4.	Aktywności klasycznego zarządzania projektem	653
14.4.1.	Planowanie projektu	654
14.4.2.	Organizowanie projektu	659
14.4.3.	Kontrolowanie projektu	665
14.4.4.	Kończenie projektu	671
14.5.	Aktywności „zwinnej” realizacji projektu	673
14.5.1.	Planowanie projektu: wykazy zaległości produktu i przebiegu	674
14.5.2.	Organizowanie projektu	675
14.5.3.	Kontrolowanie projektu: dni robocze i wykresy wygaszania	675
14.5.4.	Kończenie projektu: przeglądy przebiegów	677
14.6.	Literatura uzupełniająca	677
14.7.	Ćwiczenia	679

Rozdział 15. Cykl życiowy oprogramowania 683

15.1.	Wstęp: nawigacja polinezyjska	684
15.2.	IEEE 1074: standard cykli życiowych	688
15.2.1.	Procesy i aktywności	688
15.2.2.	Modelowanie cyklu życiowego	690
15.2.3.	Zarządzanie projektem	690
15.2.4.	Prerealizacja	691
15.2.5.	Realizacja — tworzenie systemu	692
15.2.6.	Postrealizacja	693
15.2.7.	Procesy integralne (międzyrealizacyjne)	694

15.3.	Charakteryzowanie dojrzałości modeli cyklu życiowego	695
15.4.	Modele cyklu życiowego	698
15.4.1.	Sekwencyjne modele ukierunkowane na aktywności	699
15.4.2.	Iteracyjne modele ukierunkowane na aktywności	701
15.4.3.	Modele ukierunkowane na encje	706
15.5.	Literatura uzupełniająca	709
15.6.	Ćwiczenia	710
Rozdział 16.	Wszystko razem, czyli metodologie	713
16.1.	Wstęp: pierwsze zdobycie K2	714
16.2.	Środowisko projektu	717
16.3.	Zagadnienia metodologiczne	719
16.3.1.	Ile planowania?	719
16.3.2.	Ile powtarzalności?	720
16.3.3.	Ile modelowania?	721
16.3.4.	Ile procesów cyklu życiowego?	723
16.3.5.	Ile kontroli i monitorowania?	723
16.3.6.	Kiedy przeddefiniować cele projektu?	724
16.4.	Spektrum metodologii	724
16.4.1.	Metodologia Royce'a	725
16.4.2.	Programowanie ekstremalne (XP)	731
16.4.3.	Metodologie rugby	737
16.5.	Analizy przypadku	742
16.5.1.	Projekt XP: ATTRACT	743
16.5.2.	Lokalny klient: FRIEND	746
16.5.3.	Rozproszony projekt: JAMES	754
16.5.4.	Podsumowanie analiz przypadku	761
16.6.	Literatura uzupełniająca	766
16.7.	Ćwiczenia	766
Dodatki		769
Dodatek A	Wzorce projektowe	771
A.1.	Fabryka abstrakcyjna (<i>Abstract Factory</i>) — hermetyzacja platformy	772
A.2.	Adapter (<i>Adapter</i>) — otoczka dla starszego kodu	773
A.3.	Most (<i>Bridge</i>) — podmiana implementacji	774
A.4.	Polecenie (<i>Command</i>) — hermetyzacja przepływu sterowania	775
A.5.	Kompozyt (<i>Composite</i>) — rekurencyjna reprezentacja hierarchii	776
A.6.	Fasada (<i>Facade</i>) — hermetyzacja podsystemów	777
A.7.	Obserwator (<i>Observer</i>) — oddzielenie encji od widoków	778
A.8.	Proxy (<i>Proxy</i>) — hermetyzacja kosztownych obiektów	779
A.9.	Strategia (<i>Strategy</i>) — hermetyzacja algorytmów	780
A.10.	Heurystyki pomocne w wyborze wzorców projektowych	781

Dodatek B	Objaśnienia haseł	783
	B.1. Terminologia	783
	B.2. Słownik terminów angielskich	817
Dodatek C	Bibliografia	831
	Skorowidz	847



Zarządzanie racjonalizacją

Można opisywać motocykl w kategoriach „co?” i „jak”, czyli w kategoriach komponentów, z których się składa, oraz zasad, według których komponenty te funkcjonują; oglądając ilustrację motocykla, można się jednak tylko domyślać tych wszystkich „gdzie?” i „dlaczego?” decydujących o takim, a nie innym kształcie poszczególnych części i ich wzajemnym rozmieszczeniu.

— Robert Pirsig *Zen i sztuka oporządzania motocykla*

W kontekście realizowania projektów pojęcie „racjonalizacja”¹ (*rationale*) oznacza uzasadnianie podejmowanych decyzji. Dotychczas w treści tej książki opisywaliśmy modele reprezentujące różne oblicza systemu; model racjonalizacji reprezentuje natomiast powody, dla których funkcjonalność systemu i jej implementacja przyjmują taki, a nie inny kształt. Racjonalizacja ma kluczowe znaczenie w co najmniej dwóch aspektach realizacji projektu: wspomaganie podejmowania decyzji oraz kolekcjonowaniu wiedzy. Na tak pojmowaną racjonalizację składają się:

- problemy wymagające rozwiązywania,
- możliwe ewentualności rozwiązań tych problemów,
- decyzje podejmowane w związku z rozwiązywaniem problemów,
- kryteria uwzględniane przy podejmowaniu decyzji,
- dyskusje programistów prowadzone w związku z wypracowywaniem decyzji.

Racjonalizacja przyczynia się do jakości podejmowanych decyzji, bo ujawniania podstawowe elementy decyzyjne — kryteria, priorytety i argumenty. Jeśli chodzi o kolekcjonowanie wiedzy, racjonalizacja ma znaczenie krytyczne w sytuacji, gdy pojawia się konieczność zmian: przy wprowadzaniu do systemu nowej funkcjonalności programiści mają możliwość prześledzenia podjętych dotychczas decyzji w kontekście ich uzasadnienia, sensowności, adekwatności i tym podobnych; gdy do zespołu dołączają nowi programiści, mogą łatwo zapoznać się ze wspomnianymi decyzjami, studiując dokumenty racjonalizacji systemu.

Niestety, racjonalizacja należy do najbardziej złożonych kategorii informacji zarówno w kontekście jej tworzenia, jak i utrzymywania w aktualnym stanie. Zarządzanie racjonalizacją jest bowiem inwestycją, która przynosi zysk dopiero w dłuższej perspektywie.

¹ Ponieważ większość badań związanych z racjonalizacją koncentrowała się początkowo na etapie projektowania systemu, do samego rzeczownika „racjonalizacja” przyłgnał na dobre przymiotnik „projektowa” (*design rationale*). Jak jednak pokazujemy w tym rozdziale, racjonalizacja jest nierozłącznie związana ze wszystkimi etapami projektu, dlatego będziemy o niej pisać bez przymiotników wiążących ją z konkretnymi etapami czy aktywnościami.

W tym rozdziale opiszemy modelowanie zagadnień jako reprezentację modelowania racjonalizacji, omówimy także aktywności związane z tworzeniem, pielęgnowaniem i wykorzystywaniem modeli racjonalizacji. Rozdział zakończymy opisem zagadnień menedżerskich związanych z racjonalizacją, takich jak wspomaganie podejmowania decyzji i negocjowanie.

12.1. Wstęp: przycinanie wędzonej szynki

Modele systemu są abstrakcjami wykonywanych przez niego funkcji. Modele związane z analizą wymagań — model przypadków użycia, model klas i model sekwencji (o których pisaliśmy w rozdziałach 4. „Zbieranie wymagań” i 5. „Analiza wymagań”) — reprezentują zachowanie systemu z perspektywy jego użytkownika. Model projektu systemu (patrz rozdział 6. „Projektowanie systemu — dekompozycja na podsystemy”) stanowi reprezentację systemu w kontekście podziału na podsystemy, celów projektowych, węzłów sprzętowych, przechowywania danych, kontroli dostępu i tak dalej. Model racjonalizacji systemu reprezentuje natomiast całokształt decyzji, przesłanek, powodów i tym podobnych, decydujących o tym, *dlaczego* system ten zbudowany jest i działa właśnie tak, a nie inaczej.

Rozważmy najpierw, dlaczego w ogóle powinniśmy się zastanawiać nad wspomnianym „dlaczego?” — na początek mała dygresja².

Mary zapytała kiedyś swego męża Johna, dlaczego przycina szynkę na obu końcach przed włożeniem jej do wędzarni. John odparł, że tak robiła jego mama, prawdopodobnie zgodnie z jakimś przepisem, on sam nigdy się nad tym nie zastanawiał.

Zaciekawiona Mary zapytała więc o to swą teściową, Ann. Ta odrzekła, że jej mama, Zoe, wymyśliła taki właśnie sposób na poprawienie smaku wędzonej szynki

Indagowana na tę okoliczność Zoe wyraziła głębokie zdziwienie: nigdy nie przycinała szynki przed wędzeniem, a już sugestia, jakoby takie przycinanie miało poprawiać smak szynki, jest kompletnym absurdem dla kogoś, kto ma choć blade pojęcie na temat kulinariów. Po chwili Zoe przypomniała sobie jednak, że gdy Ann była małą dziewczynką, zdarzyło się wielokrotnie, iż standardowych rozmiarów szynka nie mieściła się w zbyt ciasnym piecu i faktycznie trzeba ją było skracać o kilka centymetrów z obu końców. Wkrótce jednak kupiono nową, większą wędzarnię i technologia skracania szynki poszła w zapomnienie.

Programiści i kucharze uwielbiają rozpowszechniać rozmaite pomysły, sposoby, techniki i tym podobne, którym jednak nie towarzyszy wystarczająca racjonalizacja, która mogłaby zwiększyć ich przydatność w konkretnym zastosowaniu. Wszyscy znamy tak zwany problem roku 2000: w latach 60. i 70. ubiegłego wieku wysokie koszty pamięci i magazynowania danych skłaniały do poszukiwań różnych optymalizacji rozmiaru tych danych i jedną z takich technik było zapisywanie tylko dwóch końcowych cyfr roku (na przykład „78”), dwie pierwsze („19”) były bowiem niezmiennie. Co do tworzonego i używanego wówczas oprogramowania przyjmowano (jako pewnik) założenie, że w roku 2000 pozostanie już po nim co najwyżej wspomnienie. Czas mijał nieubłaganie, pamięci RAM i dyski drastycznie staniały, a programiści w dalszym ciągu nie przejmowali się problemem zapisywania cyfr reprezentujących stulecie. Ludzkość wkroczyła dumnie w XXI wiek i zaczęły się problemy z operacjami arytmety-

² Popularna historyjka niewiadomego autorstwa.

tycznymi wykonywanymi na dwucyfrowych zapisach lat: piszący te słowa tłumacz, urodzony w roku 1958, latem 2001 roku ukończył (jak to wyliczył jeden z programów) $01-58 = -57$ lat, a jego dziadek, świętujący w 2004 roku 105. urodziny, zaproszony został do lokalnego centrum kultury na (darmową) imprezę dla ... pięciolatek. I nie można przy tym zrzucić całej winy na pokolenie programistów lat 90.: zdając sobie sprawę ze zbliżającego się *fin de siècle*, skrupowani byli jednocześnie względami kompatybilności wstecz z eksploatowanym oprogramowaniem — w pewnym sensie kompatybilności z krótkowzrocznością swych starszych kolegów. W rezultacie, mimo iż mamy już za sobą pierwszą dekadę nowego stulecia, wiele używanych dziś programów dotkniętych jest „syndromem Y2K”, jak zwykło się nazywać (z angielska) opisany problem.

Spadek cen pamięci czy kupno większej wędzarni to przykład *zmian*, które właściwie rozumieć i wykorzystywać (lepiej sobie z nimi radzić) pozwala właśnie modelowanie racjonalizacji. Kolekcjonowanie uzasadnień podejmowanych decyzji umożliwia modelowanie zależności między tymi decyzjami i początkowo przyjmowanymi założeniami. Gdy zmieniają się założenia, należy ponownie rozważyć zasadność decyzji podjętych na ich podstawie. W tym rozdziale opiszemy techniki kolekcjonowania, pielęgnowania i wykorzystywania modeli racjonalizacji, w szczególności:

- ogólny pogląd na modelowanie racjonalizacji (patrz sekcja 12.2),
- koncepcje związane z modelami racjonalizacji (patrz sekcja 12.3),
- aktywności związane z tworzeniem i wykorzystywaniem modeli racjonalizacji (patrz sekcja 12.4),
- zadania menedżera projektu związane z utrzymywaniem modeli racjonalizacji (patrz sekcja 12.5).

Zacznijmy jednak od koncepcji samego modelu racjonalizacji.

12.2. O racjonalizacji ogólnie

Racjonalizacja decyzji to zespół motywacji kryjących się za jej podjęciem. Na racjonalizację tę składają się:

- **Zagadnienie** (*issue*). Każda decyzja związana jest z rozwiązywaniem konkretnego problemu. Klarowny opis tego problemu jest kluczowym elementem racjonalizacji; w związku z przedstawionymi przykładami pytania te brzmią „Jak należy wędzić szynkę?” oraz „W jakiej postaci prezentować rok kalendarzowy?”.
- **Ewentualności** (*alternatives*). Ewentualnościami nazywamy możliwe sposoby rozwiązania danego problemu, w tym również sposoby, które nie zostały zastosowane ze względu na niespełnienie jednego lub więcej określonych kryteriów. Przykładowo duży piec wędzarniczy może być zbyt drogi, a przechowywanie liczb oznaczających lata w postaci dwubajtowych słów zamiast bajtów może spowolnić szybkość obliczeń i zwiększyć ilość wymaganej pamięci.

- **Kryteria** (*criteria*). Określają one cechy, którymi charakteryzować się powinno rozwiązanie problemu — i tak na przykład przepis na uwędownienie szynki powinien być wykonalny przy użyciu podstawowego wyposażenia domowej kuchni, a programiści decydujący się pół wieku temu na dwucyfrowy zapis roku dążyli do oszczędności pamięci masowych. Na etapie analizy wymagań wspomniane kryteria mają postać wymagań pozafunkcyjnych i ograniczeń dotyczących między innymi łatwości użytkowania systemu czy oczekiwanej liczby błędów popełnianych codziennie przez użytkownika wprowadzającego dane wejściowe. Na etapie projektowania systemu kryteria te wyrażają cele projektowe w postaci niezawodności, czasu reakcji i tym podobnych. Z perspektywy menedżera projektu kryteria te wyrażają jego specyficzne cele oraz kompromisy, które musi rozstrzygać — na przykład kompromis między jakością systemu a terminem jego dostarczenia.
- **Argumentacja** (*arguments*). Zarówno sztuka kulinarna, jak i inżynieria oprogramowania nie są dyscyplinami algorytmicznymi, bowiem kucharze i programiści napotykają rozmaite problemy i poszukują różnych sposobów ich rozwiązywania, uwzględniając zalety i wady danego sposobu w porównaniu z innymi. Argumentowanie dotyczy wszelkich aspektów procesu decyzyjnego — kryteriów, uzasadnień, rozważanych ewentualności oraz rozstrzyganych kompromisów.
- **Decyzje** (*decisions*). Decyzja to rozwiązanie problemu, reprezentujące konkretną ewentualność wybraną zgodnie z określonymi kryteriami ewaluacji. Kilkucentymetrowe skracanie wędzonej szynki czy ignorowanie cyfr stulecia w zapisie lat to przykłady konkretnych decyzji. Konkretny kształt modelu analitycznego czy modelu projektu systemu jest niczym innym, jak skumulowanym efektem ciągu podjętych decyzji. Być może wiele z tych decyzji podjętych zostało bez starannego przeanalizowania rozwiązywanego problemu lub należytej eksploracji możliwych ewentualności.

Przez cały czas realizowania projektu, na wszystkich jego etapach, zmuszeni jesteśmy do podejmowania licznych decyzji, wspomaganych modelami racjonalizacji, i tak:

- W czasie *zbierania wymagań* i ich *analizy* decydujemy o kształcie funkcjonalnym systemu, przy wydatnej współpracy z klientem. Podejmowane decyzje motywowane są przez użytkownika lub przez potrzeby organizacyjne. Uzasadnienie tych decyzji staje się użyteczne przy tworzeniu przypadków testowych na potrzeby testów integracyjnych i testów akceptacyjnych.
- W czasie *projektowania systemu* formułujemy cele projektowe i determinujemy kształt dekompozycji systemu na podsystemy. Decyzje związane z celami projektowymi wynikają najczęściej z wymagań pozafunkcyjnych, tak więc kolekcjonowanie racjonalizacji tych decyzji umożliwia śledzenie zależności między dwiema encjami — wymaganiami pozafunkcyjnymi i celami projektowymi: gdy zmienią się wymagania, łatwiejsze będzie odpowiednie przeformułowanie celów projektowych.
- *Zarządzanie projektem* wiąże się z przyjmowaniem wielu różnych założeń związanych z relatywnym ryzykiem projektowym — i tak na przykład komponenty utworzone stosunkowo niedawno wymagają (statystycznie) więcej fady ze strony programistów

niż komponenty w miarę dojrzale. Kolekcjonowanie racjonalizacji ryzykownych decyzji i opracowywanie „planów odwrotu” umożliwi złagodzenie ewentualnych problemów, które wynikać mogą z tego ryzyka.

- W czasie *testowania i integrowania* podsystemów wykrywamy niezgodności między ich interfejsami. Na podstawie racjonalizacji kryjącej się za konkretnym kształtem tych interfejsów możemy łatwo zidentyfikować założenie, którego niespełnienie okazało się przyczyną usterki i usunąć tę usterkę przy minimalnym wpływie na resztę systemu.

Zarządzanie racjonalizacją jest inwestycją, wymagającą określonych zasobów dedykowanych zarządzaniu zmianami: kolekcjonując pewne informacje *teraz*, ułatwiamy sobie *późniejsze* weryfikowanie podejmowanych obecnie decyzji, w związku z wprowadzanymi zmianami w systemie. Wielkość wspomnianych zasobów zależna jest od konkretnego typu systemu.

Gdy na przykład tworzymy skomplikowany system dla konkretnego klienta, system ten będzie z pewnością doznawał wielu zmian w dłuższej perspektywie; klient, który jest świadom tego faktu, może wówczas nawet zażądać kolekcjonowania racjonalizacji także na swój użytek. Jeśli natomiast tworzymy koncepcyjny prototyp nowego produktu, prototyp ten prawdopodobnie okaże się nieprzydatny, gdy rozpoczną się prace nad rzeczywistym produktem. Kolekcjonowanie racjonalizacji na etapie tworzenia wspomnianego prototypu może opóźnić jego demonstrację, co stwarzać może ryzyko „skasowania” całego projektu; zarządzanie racjonalizacją jest w tym wypadku inwestycją nieopłacalną, bo w najlepszym razie dającą korzyści niewspółmierne do ponoszonego ryzyka.

Kolekcjonowanie racjonalizacji może mieć różną intensywność, odzwierciedlaną w postaci czterech następujących poziomów:

- *brak wyraźnego kolekcjonowania* — informacja związana z racjonalizacją ma jedynie formę ulotną, bo istnieje tylko w umysłach programistów, a w najlepszym razie — w listach, faksach i innej postaci komunikatach wymienianych między nimi. Całość jawnej dokumentacji systemu skupia się w jego modelach.
- *rekonstruowanie racjonalizacji* — ulotnej (w sensie powyżej opisanym) informacji związanej z racjonalizacją nadaje się fizyczną postać w ramach aktywności dokumentacyjnych projektu. Kryteria projektowe i motywacje kryjące się za najważniejszymi decyzjami architektonicznymi integrowane są z odpowiednimi modelami systemu. Odrzucone ewentualności oraz argumenty „za” i „przeciw” nie zostają jawnie udokumentowane.
- *bieżące kolekcjonowanie racjonalizacji*, w związku z każdą podejmowaną decyzją. Informacja związana z racjonalizacją reprezentowana jest w formie odrębnego modelu, zawierającego odniesienia do innych modeli — i tak na przykład modele zagadnień reprezentują racjonalizację w formie grafu, którego każdy węzeł prezentuje zagadnienie, ewentualność lub kryteria ewaluacji. Racjonalizacja modelu analitycznego może być wówczas reprezentowana przez powiązania poszczególnych zagadnień odpowiednimi przypadkami użycia.
- *zintegrowanie racjonalizacji z modelami systemu* sprawia, że staje się ona centralnym modelem systemu. Pojawiające się sukcesywnie nowe elementy racjonalizacji zapisywane są w „żywej”, przeszukiwalnej, informacyjnej bazie danych. Wszelkie

zmiany w systemie rozpoczynają swój żywot od zarejestrowania ich w tej bazie, w formie dyskusji towarzyszącej poszczególnym decyzjom. Wspomniana baza stanowi zatem swoiste podsumowanie decyzji, których konsekwencje rozproszone są po modelach systemu.

W dwóch pierwszych przypadkach miejscem przechowywania informacji związanej z racjonalizacją są umysły programistów; dwa następne przypadki reprezentują fizyczne umocowanie wspomnianej informacji, która tym samym staje się niezależna od umysłu ludzkiego, zawodnego i ograniczonego pod względem ilościowym. Wybór złotego środka między dwiema skrajnościami — brakiem fizycznego reprezentowania racjonalizacji a jej ścisłą integracją z modelami systemu — uzależniony jest od możliwości zainwestowania zasobów w owo reprezentowanie, we wczesnych etapach realizacji projektu. W tym rozdziale skupimy się wyłącznie na dwóch ostatnich z czterech wymienionych poziomów.

Oprócz korzyści długofalowych, jakie daje kolekcjonowanie racjonalizacji, przynosić ono może także korzyści doraźne. Po pierwsze, zmusza do podejmowania decyzji przemyślnych, niedyktowanych emocjami — a przynajmniej pozwala odróżnić decyzje starannie przemyślane od podejmowanych w obliczu presji czy innych „pilnych” okoliczności. Po drugie, pozwala programistom lepiej rozumieć decyzje podejmowane przez kolegów.

Modele racjonalizacji, w porównaniu z innymi modelami systemu, zawierają wyraźnie więcej informacji, zmieniającej się w wyraźnie szybszym tempie. Złożoność tej informacji stwarza potrzebę opracowania technik zarządzania nią w sposób sformalizowany. Za chwilę opiszemy reprezentowanie racjonalizacji w formie modeli zagadnień.

12.3. Koncepcje racjonalizacji

W tej sekcji opiszemy modelowanie zagadnień, opierające się na założeniu, że aktywności programistów są działaniami dialektycznymi, zmierzającymi do rozwiązywania problemów drogą poszukiwania i ewaluowania różnych ewentualności. Modelując argumenty przemawiające „za” i „przeciw” wspomnianym ewentualnościom, tworzymy reprezentację racjonalizacji obejmującą:

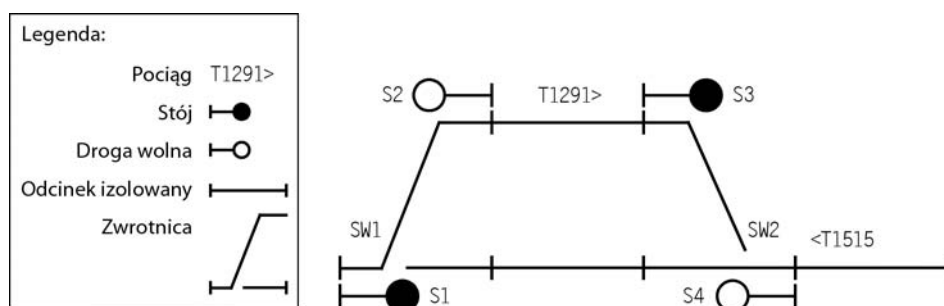
- zagadnienia reprezentujące problemy projektowe i pytania związane z projektem (patrz sekcja 12.3.2),
- ewentualności rozwiązań tych problemów (patrz sekcja 12.3.3),
- argumenty „za” poszczególnymi rozwiązaniami i „przeciw” nim (patrz sekcja 12.3.4),
- decyzje wyboru konkretnych rozwiązań (patrz sekcja 12.3.5),
- implementacje podjętych decyzji w postaci przydziału zadań (patrz sekcja 12.3.6).

W sekcji 12.3.7 dokonamy przeglądu wybranych systemów reprezentowania problemów — systemów o znaczeniu historycznym. Najpierw jednak przyjrzymy się szczegółowo scentralizowanemu systemowi kierowania ruchem kolei miejskiej, który to system stanowić będzie kanwę do budowania rozmaitych przykładów na potrzeby tego rozdziału.

12.3.1. CTC — system centralnego sterowania ruchem

Scentralizowany system sterowania ruchem (w dalszym ciągu określać go będziemy skrótem CTC, od *Centralized Traffic Control*) umożliwia zdalne monitorowanie ruchu pociągów i zdalne sterowanie tym ruchem. Każda trasa podzielona jest na tak zwane odcinki izolowane, z których każdy reprezentuje najmniejszy niepodzielny odcinek monitorowania. Zadaniem sygnalizatorów i innych urządzeń jest stworzenie gwarancji, że w danej chwili na każdym ze wspomnianych odcinków znajduje się co najwyżej jeden pociąg. Gdy pociąg wjeżdża na dany odcinek izolowany, specjalne czujniki rejestrują ten fakt i powodują wyświetlenie na monitorze dyspozytora (w odpowiedniej lokalizacji, odpowiadającej danemu odcinkowi) informacji zawierającej między innymi identyfikator pociągu. Do łączenia odcinków w kompletne trasy służą zwrotnice, zarządzane zdalnie przez dyspozytorów. Zbiór wszystkich odcinków znajdujących się pod kontrolą określonego dyspozytora nazywamy „sekcją dyspozycyjną”.

Na rysunku 12.1 przedstawiona jest uproszczona postać interfejsu użytkownika systemu CTC. Odcinki izolowane reprezentowane są przez linie, zwrotnice — przez zbieg trzech lub czterech linii. Sygnały reprezentowane są przez ikony odzwierciedlające dwa stany: „stój” i „droga wolna”. Zwrotnice, pociągi i sygnalizatory opatrzone są unikalnymi identyfikatorami, używanymi przez dyspozytora w celu wydawania poleceń. Na rysunku 12.1 widzimy zatem sygnalizatory S1, S2, S3 i S4, zwrotnice SW1 i SW2 i pociągi T1291 i T1515. Komputery zlokalizowane w pobliżu odcinków, zwane „stacjami przydrożnymi”, stanowią zabezpieczenie gwarantujące, że grupa sygnalizatorów i zwrotnic nigdy nie znajdzie się w stanie zagrażającym bezpieczeństwu — czyli na przykład sygnał „droga wolna” nie zostanie wyświetlony równocześnie na sygnalizatorach S1 i S2. Stacje przydrożne zostały tak zaprojektowane, że zapewniają bezpieczeństwo nawet w przypadku awarii całego systemu, bowiem system ten komunikuje się z sygnalizatorami i zwrotnicami wyłącznie za ich pośrednictwem. Sam system nie musi więc być „całkowicie bezpieczny” (*failsafe*), choć jego ewentualne awarie powinny mieć charakter sporadyczny.



Rysunek 12.1. Uproszczony przykład wyświetlanej sekcji dyspozycyjnej CTC

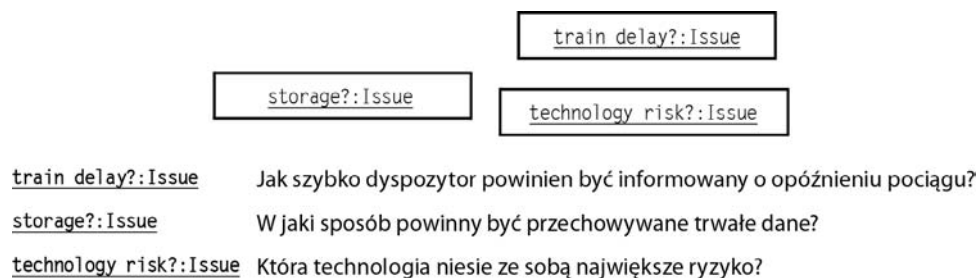
W latach 60. ubiegłego wieku pulpity sterownicze systemu CTC miały postać dedykowanych urządzeń wyposażonych w żarówki wyświetlające stan poszczególnych odcinków izolowanych oraz przyciski służące do przestawiania zwrotnic i ustawiania sygnalizacji. W latach 70. dedykowane pulpity zastąpione zostały monitorami kineskopowymi, umożliwiającymi wyświetlanie większej ilości szczegółów na mniejszej powierzchni. Ostatnio monitory

te ustąpiły miejsca stacjom roboczym umożliwiającym tworzenie bardziej wyrafinowanych interfejsów dla dyspozytorów i pozwalającym na rozproszenie przetwarzania między kilka komputerów. Mimo iż system nie jest krytyczny dla życia i zdrowia ludzkiego — nad którego bezpieczeństwem czuwają stacje przydrożne — jednak jego awaria spowodować może chaos komunikacyjny, przekładający się (między innymi) na straty ekonomiczne. W konsekwencji wszelkie transformacje technologiczne CTC — takie jak zastępowanie dedykowanych pulpitów monitorami czy wymiana tych monitorów (połączonych z komputerami *mainframe*) na stacje robocze — powinny odbywać się bardziej stopniowo niż w innych systemach. Tak oto CTC jawi się jako dziedzina, w której kolekcjonowanie racjonalizacji jest czynnikiem krytycznym — i dlatego właśnie wybraliśmy ją jako przykładową na potrzeby tego rozdziału.

12.3.2. Definiowanie problemów: zagadnienia

Zagadnienie to reprezentacja konkretnego problemu, którym może być wymaganie, projekt lub element zarządzania. *Jak szybko dyspozytor powinien być informowany o opóźnieniu pociągu? W jaki sposób powinny być przechowywane trwałe dane? Która technologia niesie ze sobą największe ryzyko?* Zagadnienia często reprezentują problemy, dla których nie istnieje jedyne poprawne rozwiązanie i które z tego powodu nie mogą być rozwiązywane algorytmicznie. Zagadnienia rozwiązywane są zwykle w drodze dyskusji i negocjacji.

W języku UML zagadnienia reprezentowane są przez instancje klasy Issue. W klasie tej definiowany jest atrybut *subject* reprezentujący streszczenie zagadnienia, atrybut *description* reprezentujący bardziej szczegółowy opis zagadnienia i jego związki z innymi materiałami oraz atrybut *status* informujący, czy zagadnienie zostało rozwiązane („zamknięte” — *closed*), czy nie („otwarte” — *open*). Zagadnienia zamknięte mogą być ponownie otwierane. Zgodnie z konwencją zagadnienia opatrywane są krótkimi nazwami, na przykład *train delay?*, umożliwiającymi ich jednoznaczną identyfikację. Na rysunku 12.2 przedstawiliśmy reprezentację trzech zagadnień sformułowanych w poprzednim akapicie.



Rysunek 12.2. Przykłady zagadnień

Zagadnienia pojawiające się w związku z realizacją projektu zwykle bywają powiązane, między innymi zagadnienia mogą być dekomponowane na prostsze podzagadnienia. Zagadnienie *Jaki powinien być czas reakcji w systemie sterowania ruchem?* automatycznie generuje zagadnienie *Jak szybko dyspozytor powinien być informowany o opóźnieniu pociągu?* Zresztą, samo tworzenie systemu CTC może być postrzegane jako pojedyncze złożone zagadnienie/*Jaki*

system kontroli ruchu powinniśmy zbudować?, podlegające dekomponowaniu na znaczną liczbę prostszych podzagadnień. Zagadnienia mogą być także generowane przez decyzje podejmowane w związku z innymi zagadnieniami. Przykładowo zagadnienie dotyczące cache'owania danych w węźle lokalnym generuje natychmiast zagadnienia związane z utrzymywaniem spójności między centralnym egzemplarzem danych a jego replikami w węzłach lokalnych. Takie wtórne zagadnienia nazywamy **zagadnieniami wynikowymi** (*consequent issues*).

Powróćmy do systemu CTC i wyobraźmy sobie, że rozważamy jego migrację z komputera *mainframe* na sieć stacji roboczych. W nowej wersji systemu każdy dyspozytor będzie korzystał z osobnej stacji roboczej komunikującej się z serwerem, który zapewnia komunikację ze stacjami przydrożnymi. W czasie dyskusji na ten temat wyniknęły dwa zagadnienia: *W jaki sposób dyspozytor powinien wydawać polecenia systemowi?* oraz *W jakiej postaci na ekranie stacji roboczej ma być wyświetlany stan odcinków izolowanych?*. Na rysunku 12.3 zagadnienia te przedstawione są na diagramie obiektów UML.



Rysunek 12.3. Zagadnienia związane z interfejsem CTC

Zagadnienie powinno koncentrować się wyłącznie na problemie, nie na możliwych ewentualnościach jego rozwiązania (ewentualności te są bowiem przedmiotem propozycji, o których pisać będziemy w następnej sekcji). Konwencją promującą tę regułę jest formułowanie zagadnień w formie pytań; aby dodatkowo wyeksponować tę regułę, dołączać będziemy znak zapytania na końcu nazwy zagadnienia.

12.3.3. Eksploracja przestrzeni rozwiązań: propozycje

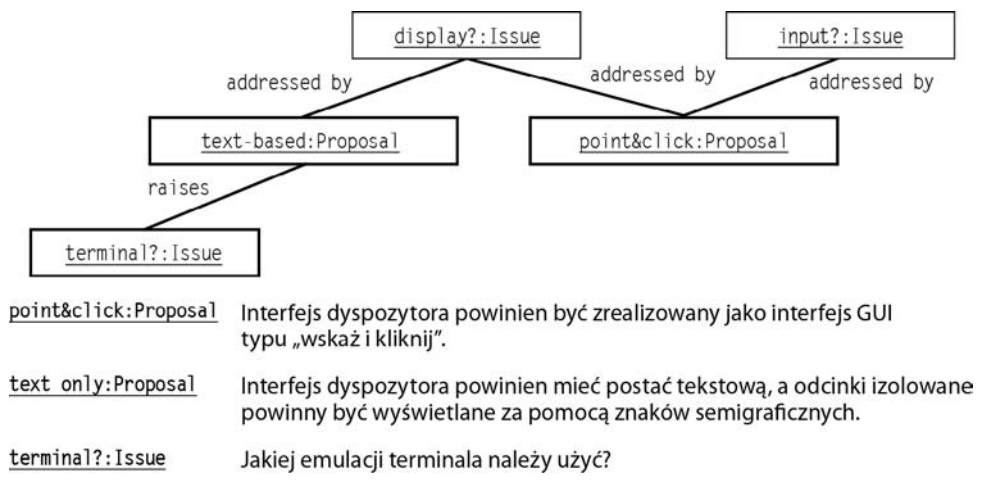
Propozycja jest jedną z odpowiedzi na zagadnienie. *Dyspozytor nie musi być informowany o opóźnieniach pociągów* to propozycja związana z zagadnieniem *Jak szybko powinien być dyspozytor informowany o opóźnieniu pociągu?* Propozycja niekoniecznie musi być dobra czy poprawna w kontekście zagadnienia, z którym jest powiązana. Propozycje umożliwiają dogłębne eksplorowanie przestrzeni rozwiązań: często w czasie burzy mózgów nawet bezsensowne propozycje stają się źródłem nowych pomysłów, które w innych okolicznościach być może nie miałyby szansy zaistnienia. Różne propozycje związane z tym samym zagadnieniem mogą się zająć, tak jak na przykład dla zagadnienia *Jak przechowywać trwale dane?* propozycje *Należy użyć relacyjnej bazy danych* oraz *Należy użyć relacyjnej bazy danych dla danych strukturalnych i „płaskich” plików dla obrazów i dźwięków*. Propozycje służą do reprezentowania zarówno wybranych rozwiązań dla zagadnień, jak i odrzuconych ewentualności rozwiązania.

Jedna propozycja może odnosić się do kilku zagadnień, na przykład propozycja *Należy wykorzystać architekturę „model-widok-kontroler”* może być adresowana do zagadnień *Jak odseparować obiekty interfejsu od obiektów encji?* oraz *Jak zapewnić spójność między wieloma*

widokami tego samego obiektu?. Propozycja może stać się także źródłem nowego zagadnienia: propozycja *Należy wykorzystywać automatyczne odświeżanie* jako odpowiedź na zagadnienie *Jak minimalizować wycieki pamięci?* prowadzi natychmiast do zagadnienia *Jak minimalizować degradację reaktywności systemu spowodowaną automatycznym zarządzaniem pamięcią?*. Rozważając sensowność propozycji w kontekście danego zagadnienia, należy brać pod uwagę także wszystkie jego zagadnienia wynikowe.

Na diagramach UML propozycje reprezentowane są jako instancje klasy `Proposal`. W klasie tej (podobnie jak w klasie `Issue`) definiowane są atrybuty `subject` i `description`. Zgodnie z konwencją, nazwa propozycji powinna mieć formę krótkiej frazy czasownikowej. Propozycje jako obiekty klasy `Proposal` powiązane są z odnosnymi zagadnieniami, jako obiektami klasy `Issue`, za pomocą dwojakiego typu skojarzeń: skojarzenie `addressed by` wskazuje zagadnienie, dla którego dana propozycja jest odpowiedzią, zaś skojarzenie `raises` wskazuje jedno lub kilka zagadnień generowanych przez daną propozycję.

Powróćmy do CTC. Rozważając zagadnienie związane z interfejsem dyspozytora, mamy do wyboru dwie propozycje: graficzny interfejs typu „wskaz i kliknij” (`point&click`) oraz interfejs tekstowy (`text-based`), w ramach którego odcinki izolowane reprezentowane są w postaci znaków semigraficznych. Propozycja interfejsu tekstowego generuje zagadnienie wynikowe dotyczące wyboru konkretnej emulacji terminala. Wzajemne powiązanie tych zagadnień i propozycji widoczne jest na rysunku 12.4.



Rysunek 12.4. Przykład propozycji i zagadnienia wynikowego. Propozycje i zagadnienie wynikowe generowane przez jedną z nich wyróżnione zostały pogrubioną ramką

Propozycja powinna zawierać jedynie meritum proponowanego rozwiązania, w oderwaniu od jego oceny, zalet i wad, te bowiem są domeną innych elementów UML — kryteriów i argumentów.

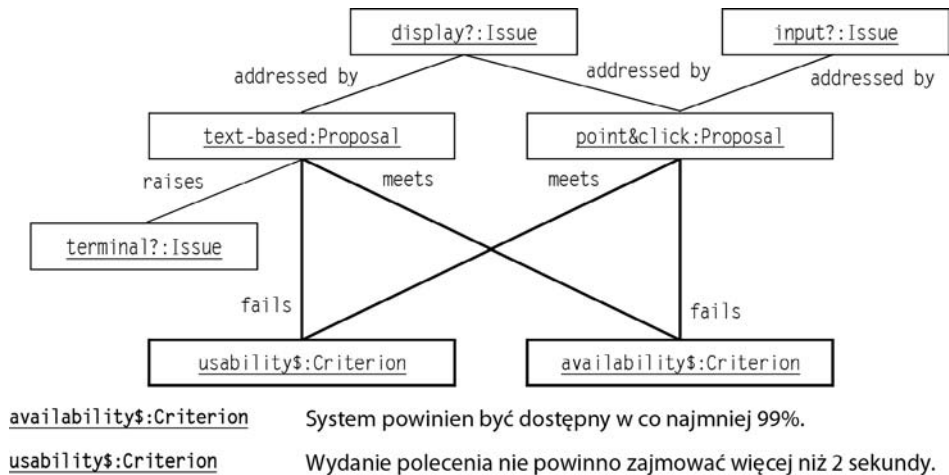
12.3.4. Wartościowanie elementów przestrzeni rozwiązań: kryteria i argumenty

Kryterium wyraża pożądaną jakość propozycji przynależnej do określonego zagadnienia. Cele projektowe, takie jak na przykład czas reakcji systemu czy jego niezawodność, to kryteria związane z celami projektowymi. Cele menedżerskie, w postaci minimum kosztów i minimum ryzyka, to kryteria związane z zagadnieniami wynikającymi z zarządzania projektem. Zbiór kryteriów wskazuje kierunki oceniania poszczególnych propozycji. Propozycja spełniająca określone kryterium kwalifikowana jest jako *oceniona pozytywnie*, propozycja niespełniająca go — jako *oceniona negatywnie*, w kontekście tegoż kryterium. Dane kryterium może być współdzielone przez wiele zagadnień.

Na diagramach UML kryteria reprezentowane są w postaci instancji klasy *Criterion*. Klasa ta, podobnie jak klasa *Issue*, posiada atrybuty *subject* i *description*. Atrybut *subject* zawsze wyrażany jest w ukierunkowaniu pozytywnym, to znaczy musi wyrażać kryterium, które poszczególne propozycje powinny *maksymalizować* — czyli na przykład *szybkość*, *reaktywność*, *niski koszt*, a nie *czas* czy *koszt* (pożądanymi cechami są bowiem *maksymalna* szybkość, *maksymalna* reaktywność i *maksymalnie* niski koszt, ale nie maksymalny czas i maksymalny koszt). Kryteria (jako obiekty klasy *Criterion*) wiązane są z odpowiednimi propozycjami (obiektami klasy *Proposal*) za pomocą skojarzeń *assessment*. Skojarzenie posiada atrybut *value*, wyrażający propozycję w kontekście danego kryterium („spełniająca” — *meets* — albo „niespełniająca” — *fails*), oraz atrybut *weight*, wyrażający znaczenie („wagę”) propozycji w kontekście tegoż kryterium. Zgodnie z konwencją, na końcu nazwy kryterium dołącza się znak dolara (\$) dla zaznaczenia, że kryteria stanowią **miarę dobroci** i nie powinny być mylone z zagadnieniami czy argumentami.

W związku z interfejsem dyspozytora CTC formułujemy dwa kryteria: dostępności (*availability*) jako нефункциjne wymagania jak najdłuższej pracy bez awarii i użyteczności (*usability*) rozumianej w tym przypadku jako minimalizacja czasu wydawania poprawnych poleceń (patrz rysunek 12.5). Kryteria te wyprowadzone zostały wprost z нефункциjnych wymagań dla systemu. W kontekście tych kryteriów rozpatrujemy obie propozycje dotyczące wariantów technologii interfejsu (GUI albo tekstowy), i tak interfejs GUI okazuje się niezgodny z kryterium dostępności, jest bowiem bardziej skomplikowany, a więc trudniejszy do przetestowania i bardziej podatny na błędy; interfejs GUI okazuje się jednak lepszy z perspektywy kryterium użyteczności, ze względu na wygodniejszą formę wydawania poleceń i wprowadzania danych. Tak oto spotykamy się ze sprzecznością wymagającą kompromisu: każda z propozycji okazuje się maksymalizować jedno kryterium przy minimalizowaniu drugiego. Należy zatem zdecydować, któremu z kryteriów przypisuje się większą wagę (*weight*).

Argument to opinia wyrażona przez osobę, zgadzającą się lub nie z propozycją, kryterium czy oceną. Argumenty są odzwierciedleniem debaty poświęconej poszukiwaniu rozwiązań; definiują adekwatność miary dobroci, a czasem prowadzą do podejmowania decyzji. Na diagramach UML argumenty reprezentowane są przez instancje klasy *Argument*, posiadającej atrybuty *subject* i *description*. Argumenty powiązane są z encją, której dotyczą, za pomocą skojarzeń *is supported* lub *is opposed*, zależnie od tego, czy przemawiają na jej rzecz, czy przeciwko niej.



Rysunek 12.5. Przykład oceny propozycji przez kryteria. Kryteria wyróżnione zostały pogrubioną ramką. Spełnienie kryterium przez propozycję reprezentowane jest przez etykietę *meets* skojarzenia *assessment*, niespełnienie — przez etykietę *fails*

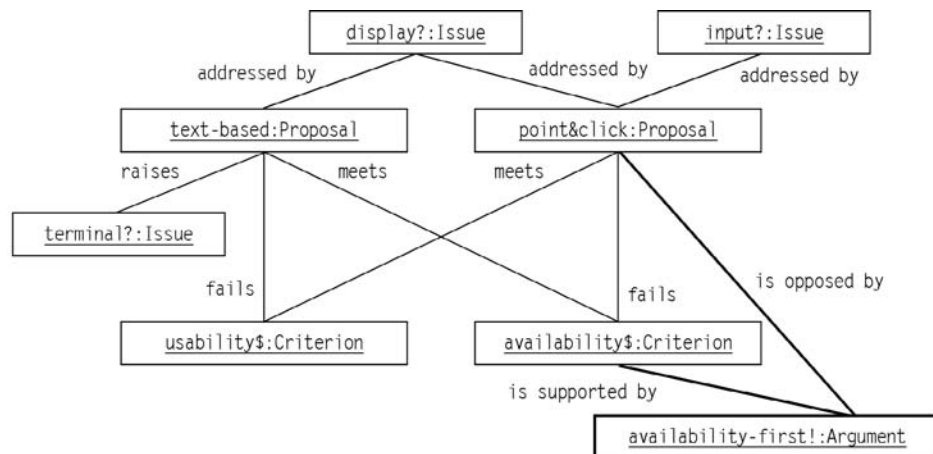
Stając wobec wyboru między dostępnością a użytecznością systemu CTC, zauważamy, że korzyści z większej użyteczności systemu mogą być okupione uszczerbkiem na jego dostępności, co może się przekładać na poważne awarie niwelujące korzyści wynikające z wygodniejszej i sprawniejszej obsługi systemu — jest to argument przemawiający zarówno na rzecz kryterium dostępności (*availability*), jak i przeciwko interfejsowi GUI (*point&click*). Relację tę przedstawiliśmy na rysunku 12.6.

Definiując kryteria, oceniając propozycje, argumentując „za” i „przeciw”, dokonujemy wartościowania elementów w przestrzeni rozwiązań. Kolejnym krokiem jest wybór jednego z tych rozwiązań w celu rozstrzygnięcia problemu i zamknięcia zagadnienia.

12.3.5. Kolapsacja przestrzeni rozwiązań: rozstrzygnięcie

Rozstrzygnięcie (*resolution*) reprezentuje ewentualność wybraną jako sposób zamknięcia zagadnienia. Decyzja rozstrzygnięcia wpływa na jeden z modeli systemu lub model zadań. Rozstrzygnięcie jest wynikiem rozpatrywania wielu propozycji i jednocześnie podsumowaniem ich uzasadnienia. Na diagramach UML rozstrzygnięcia reprezentowane są przez instancje klasy *Resolution*, definiującej atrybuty *subject*, *description*, *justification* i *status*. Każde rozstrzygnięcie, jako obiekt klasy *Resolution*, powiązane jest z odnośnymi propozycjami (jako obiektami klasy *Proposal*) za pomocą skojarzeń *based-on*. Jednocześnie każde rozstrzygnięcie powiązane jest za pomocą skojarzenia *resolves* z dokładnie jednym zagadnieniem (jako obiektem klasy *Issue*), dla którego jest rozwiązaniem.

Nieustanne zmiany w projekcie mogą powodować, że rozstrzygnięcie, które aktualnie jest rozwiązaniem pewnego zagadnienia, przestanie nim być, gdy zamknięte (*closed*) zagadnienie zostanie ponownie otwarte (*open*). Z tą okolicznością wiąże się atrybut *status* klasy *Resolution*; gdy odnośne zagadnienie jest zamknięte, czyli gdy rozstrzygnięcie pozostaje jego



availability-
first!:Argument

Interfejs „wskaz i kliknij” jest bardziej skomplikowany w implementacji niż interfejs tekstowy. Jest także trudniejszy w testowaniu, bowiem znacznie szerszy jest repertuar akcji, jakie może wykonywać dyspozytor. Wiąże się więc z ryzykiem wprowadzenia do systemu fatalnych błędów, których konsekwencje mogą całkowicie zniwelować wszelkie korzyści wynikające z wygodniejszej obsługi.

Rysunek 12.6. Przykład argumentu (reprezentujący go obiekt został wyróżniony pogrubioną ramką)

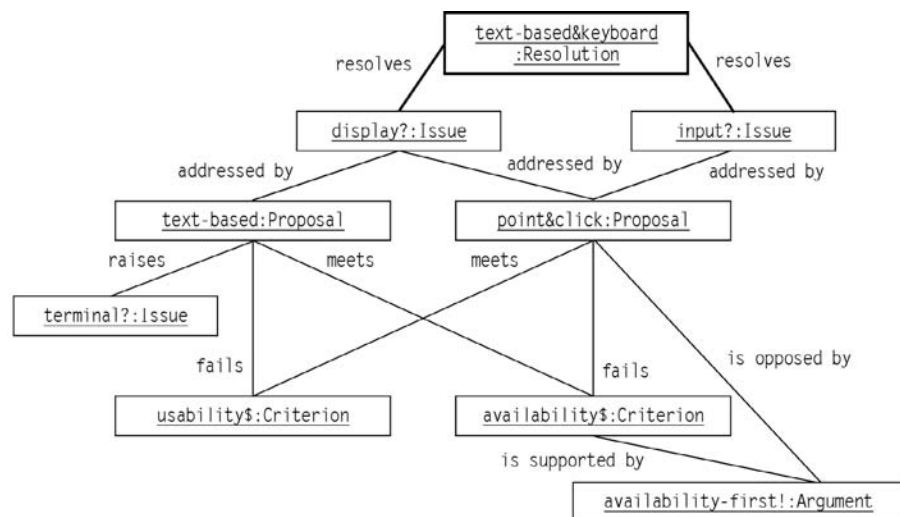
rozwiązaniem, atrybut status ma wartość *active*; w przeciwnym razie ma on wartość *obsolete*. Z zamkniętym zagadnieniem skojarzone jest dokładnie jedno rozstrzygnięcie o statusie *active* i dowolna liczba (lub zero) rozstrzygnięć o statusie *obsolete*.

Ostatecznie więc zdecydowaliśmy się na wybór tekstowego interfejsu dla naszego systemu CTC. Decyzja taka podyktowana została większym priorytetem kryterium dostępności w stosunku do kryterium użyteczności; w konsekwencji dyspozytor nie będzie mógł oglądać tak dużej ilości danych jednocześnie jak w przypadku interfejsu GUI, a wprowadzanie danych za pomocą klawiatury będzie trwało dłużej i będzie mniej wygodne w porównaniu z klikaniem. W zamian jednak zyskujemy większą pewność funkcjonowania całego systemu. Na diagramie UML (patrz rysunek 12.7) wybrane przez nas rozwiązanie reprezentowane jest jako obiekt klasy *Resolution* skojarzony z dwoma obiektami klasy *Issue*.

Dodanie rozstrzygnięcia do modelu zagadnień oznacza formalne zakończenie dyskusji nad danym zagadnieniem. Wobec iteratywnego charakteru procesu tworzenia systemu mogą pojawić się potrzeby otwarcia któregoś z zamkniętych zagadnień i ponownego ocenienia odrzuconych ewentualności. Gdy jednak proces ten zostanie zakończony, większość zagadnień musi być zamknięta, a otwarte powinny być jawnie opisane w dokumentacji w postaci listy znanych problemów.

12.3.6. Implementowanie rozstrzygnięć: elementy działania

Rozstrzygnięcie implementowane jest w postaci **elementów działania** (*action items*). Element działania to zadanie, do którego przydzielona jest osoba odpowiedzialna i dla którego określono datę zakończenia. Elementy działania nie są elementami racjonalizacji *per se*, lecz raczej



text-based &
keyboard:Resolution

Wybraliśmy tekstowy sposób prezentowania danych i wprowadzanie danych za pomocą klawiatury. Odpowiednia emulacja terminala musi zapewnić czytelne wyświetlanie stanu odcinków izolowanych za pomocą znaków semigraficznych.

Nasza decyzja podyktowana została prostotą i większą niezawodnością interfejsu tekstowego w porównaniu z interfejsem GUI. Zdajemy sobie sprawę, że jej podjęcie okupione zostanie mniejszą wygodą w zakresie użytkowania systemu przez dyspozytora, a wprowadzanie przez niego danych za pomocą klawiatury będzie wolniejsze i bardziej podatne na błędy.

Rysunek 12.7. Przykład zamkniętego zagadnienia. Jego rozstrzygnięcie wyróżnione jest pogrubioną ramką

częścią modelu zadań (patrz rozdział 14. „Zarządzanie projektem”); mimo to, opisujemy je w tym miejscu, ponieważ są ściśle powiązane z modelami zagadnień. Na diagramach UML elementy działania reprezentowane są jako instancje klasy *ActionItem* definiującej atrybuty *subject*, *description*, *owner*, *deadline* i *status*. Atrybut *owner* reprezentuje osobę odpowiedzialną za wykonanie zadania związanego z elementem działania, atrybut *status* może przyjmować cztery wartości reprezentujące stan wspomnianego zadania: *todo* („do wykonania”), *notDoable* („niewykonalne”), *inProgress* („trwające”) i *done* („wykonane”). Rozstrzygnięcie powiązane jest z odnośnym elementem działania poprzez skojarzenie *is implemented by*. Na rysunku 12.8 widoczny jest element działania generowany przez rozstrzygnięcie z rysunku 12.7.

Zakończyliśmy opis notacji wykorzystywanej do reprezentowania racjonalizacji w postaci modelu zagadnień i jego integracji z modelem zadań, przyjrzyjmy się więc funkcjonującym w praktyce wybranym systemom zarządzania racjonalizacją.

12.3.7. Przykłady modeli zagadnień i ich realizacje

Kolekcjonowanie racjonalizacji w postaci modeli zagadnień zostało oryginalnie zaproponowane przez W. Kunza i H. Rittela. Od tego czasu opracowano wiele różnych modeli na potrzeby inżynierii oprogramowania i nie tylko. Opiszemy w skrócie cztery z nich: IBIS (*Issue-Based Information System* [Kunz i Rittel, 1970]), DRL (*Decision Representation Language* [Lee, 1990]), QOC (*Questions, Options, and Criteria* [MacLean i in., 1991]) i NFR Framework [Chung i in., 1999].



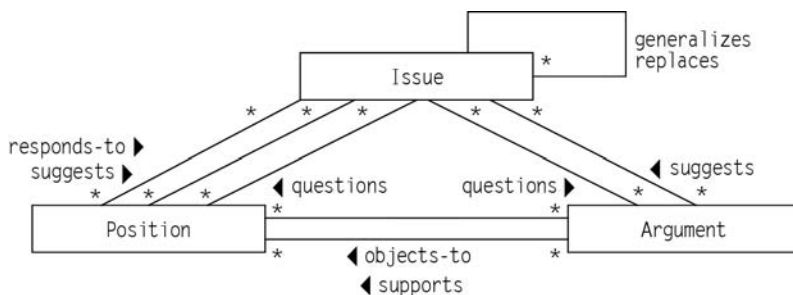
Rysunek 12.8. Przykład implementacji rozstrzygnięcia. Element działania wyróżniony jest pogrubioną ramką

Issue-Based Information System

Issue-Based Information System (IBIS) to system dedykowany problemom o złej strukturze i problemom zagmatwanym, nietypowym (w odróżnieniu od problemów typowych, „oswojonych”): takich problemów nie sposób „ruszyć” w sposób algorytmiczny, można je rozwiązywać jedynie w drodze dyskusji i debat.

Model zagadnień systemu IBIS (patrz rysunek 12.9) tworzony jest przez trzy klasy węzłów: Issue, Position i Argument kojarzonych ze sobą za pomocą siedmiu różnych skojarzeń: supports („wspiera”), objects-to („sprzeciwia się”), replaces („zastępuje”), responds-to („odpowiada”), generalizes („uogólnia”), questions („kwestionuje”) i suggests („sugeruje”). Węzeł klasy Issue reprezentuje problem projektowy. Propozycje jego rozwiązywania reprezentowane są przez węzły klasy Position (podobnej do klasy Proposal opisywanej w sekcji 12.3.3). Węzły klasy Argument reprezentują wartościowanie przez programistów poszczególnych propozycji i powiązane są z węzłami klasy Position za pomocą skojarzeń supports („wspiera”) i objects-to („sprzeciwia się”). Dany argument (Argument) może być powiązany z kilkoma propozycjami.

Oryginalny model systemu IBIS nie zawiera węzłów reprezentujących kryteria i rozstrzygnięcia.

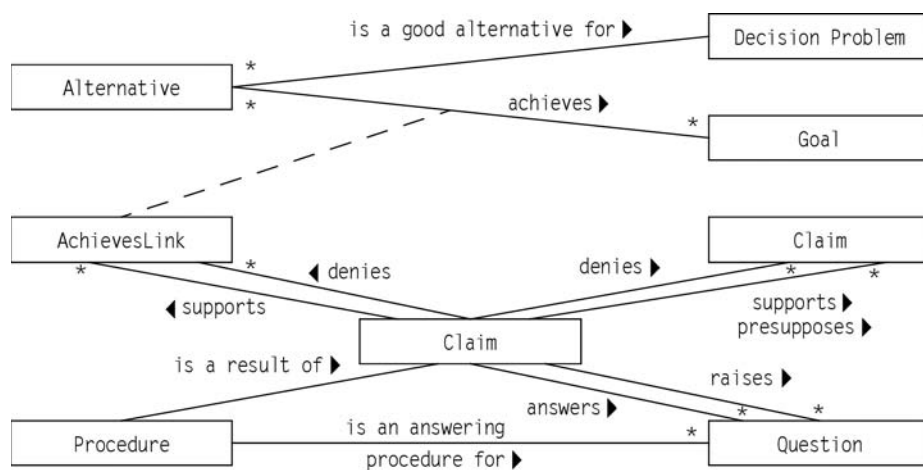


Rysunek 12.9. Model systemu IBIS

IBIS wspierany jest przez narzędzie hipertekstowe o nazwie gIBIS, opisane przez J. Conklina i K. C. Burgessa-Yakemovica [Conklin i Burgess-Yakemovic, 1991], i wykorzystywany do kolekcjonowania racjonalizacji w trakcie spotkań osobistych. Stanowi jednocześnie bazę dla większości późniejszych rozwiązań bazujących na modelach zagadnień, między innymi DRL i QOC.

Decision Representation Language

System **Decision Representation Language (DRL)** pomyślany został jako narzędzie wspomagające kolekcjonowanie racjonalizacji projektowania, rozumianej jako reprezentacja jakościowych elementów podejmowania decyzji — rozpatrywanych ewentualności rozwiązania, ich ocen i kształtujących te oceny argumentów oraz kryteriów oceniania. DRL wspierany jest przez narzędzie o nazwie SYBIL, umożliwiające użytkownikowi śledzenie zależności między elementami racjonalizacji w sytuacji ponownej ewaluacji poszczególnych ewentualności rozwiązań. DRL stanowi rozwinięcie systemu IBIS, przez wzbogacenie modelu o dwie klasy węzłów reprezentujące cele projektowe (Goal) i procedury (Procedure). Z perspektywy DRL konstruowanie racjonalizacji związanej z artefaktem jest zadaniem porównywalnym z tworzeniem samego artefaktu. Podstawową wadą DRL jest jego złożoność — siedem klas węzłów i piętnaście klas skojarzeń, co widać na rysunku 12.10 — oraz dodatkowy wysiłek konieczny dla nadania racjonalizacji odpowiedniej struktury.

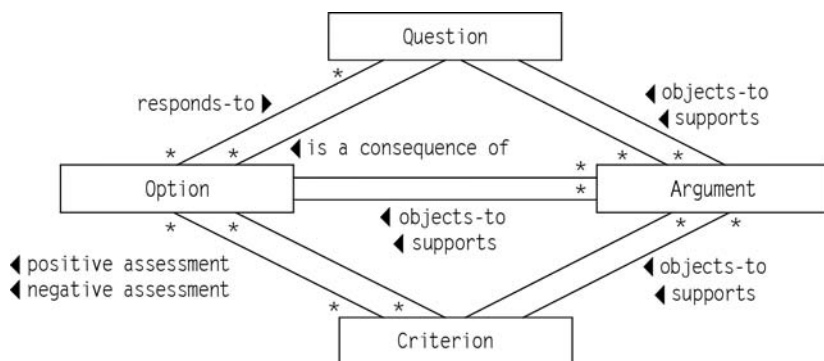


Rysunek 12.10. Model systemu DRL

Questions, Options, and Criteria

Innym efektem rozszerzenia systemu IBIS jest system o nazwie **Questions, Options, and Criteria (QOC)** („pytania, opcje i kryteria”). Węzły klasy **Question** („pytania”) składają się na reprezentację rozwiązywanego problemu. Propozycje jego rozwiązań reprezentowane są przez „opcje”, czyli węzły klasy **Option** (odpowiadającej klasie **Proposal** opisywanej w sekcji 12.3.3).

Opcje mogą generować kolejne pytania; są też oceniane (dodatnio lub ujemnie) według kryteriów (Criteria) stanowiących relatywną miarę dobroci definiowaną przez programistów. Argumenty (Argument) mogą wspierać lub kwestionować pytania, opcje i kryteria oraz skojarzenia między nimi. Model systemu QOC przedstawiony jest schematycznie na rysunku 12.11.



Rysunek 12.11. Model systemu QOC

QOC i IBIS różnią się jednak zasadniczo na poziomie procesów. Zadaniem systemu IBIS jest kolekcjonowanie elementów racjonalizacji na bieżąco, gdy pojawiają się w toku dyskusji i rejestrowane są za pomocą wspomnianego narzędzia gIBIS. W systemie QOC struktura modelu tworzona jest natomiast jako wynik refleksji nad bieżącym stanem projektu. Konceptyjna separacja faz konstrukcji i argumentacji akcentuje systematyczność i strukturalny charakter racjonalizacji, w przeciwieństwie do budowania jej „na gorąco”. Z perspektywy modelu QOC racjonalizacja jest opisem eksplorowania przestrzeni projektowej przez programistów, system IBIS postrzega natomiast racjonalizację jako chronologiczny zapis analizy prowadzącej do określonego projektu. W praktyce każde z tych podejść okazuje się przydatne do kolekcjonowania racjonalizacji w wystarczającym kształcie. (Aktywnościami związanymi z kolekcjonowaniem racjonalizacji i zarządzaniem nią zajmujemy się w sekcji 12.4).

NFR

Framework NFR³, opisany przez L. Chunga, B. A. Nixona, E. Yu i J. Mylopoulosa [Chung i in., 1999], w przeciwieństwie do trzech poprzednio omawianych modeli zagadnień dedykowany jest inżynierii wymagań, udostępnia bowiem metody śledzenia wymagań pozafunkcyjnych związanych z poszczególnymi decyzjami, ocenianych propozycji rozwiązań i interakcji między wspomnianymi wymaganiami. Wymagania pozafunkcyjne traktowane są jako *cele* do osiągnięcia; jako że wymagania pozafunkcyjne są ze swej natury wysokopoziomowe i subiektywne — a przez to trudno się nimi operuje na poziomie modeli zagadnień — wspomniane cele mogą być doskonalone przez podział na podcele. Cele i podcele reprezentowane są w modelu jako węzły grafu, relacje dekompozycji na podcele reprezentowane są przez skierowane krawędzie tego grafu. Model NFR dostarcza dwa typy dekompozycji. Oto one.

³ Skrót od *Non-Functional-Requirements* — „wymagania pozafunkcyjne” — *przyp. tłum.*

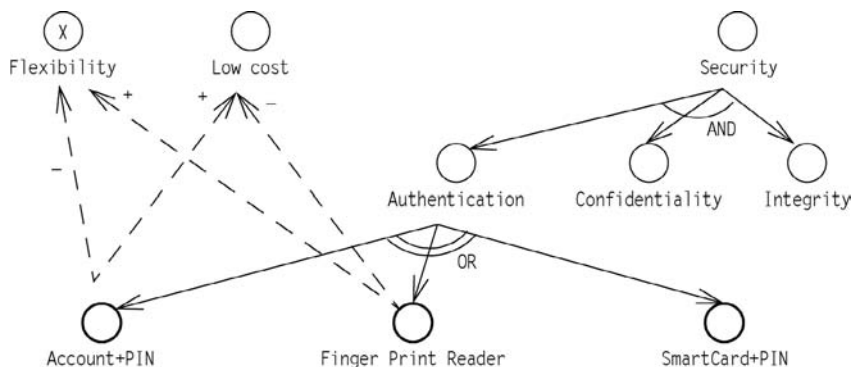
- *Dekompozycja koniunkcyjna (AND decomposition)* — by spełniony był określony cel, muszą być spełnione *wszystkie* podcele stanowiące wynik jego dekompozycji.
- *Dekompozycja dysjunkcyjna (OR decomposition)* — by spełniony był określony cel, wystarczy spełnienie *co najmniej jednego* spośród podcelów stanowiących wynik jego dekompozycji.

Tak więc cele wysokopoziomowe (specyfikowane przez klienta i użytkowników) dekomponowane są na bardziej szczegółowe podcele. Co ciekawe, nie musi to być podział ściśle drzewiasty — jeden podcel może być podporządkowany kilku celom macierzystym. NFR udostępnia ponadto nowe klasy skojarzeń do reprezentowania rozmaitych powiązań, przykładowo dany cel może wspierać inny cel lub kolidować z nim. Ponieważ wymagania pozafunkcyjne rzadko mają postać umożliwiającą kwalifikowanie binarne („spełnione-niespełnione”), skojarzenia między celami (podcelami) reprezentują także stopień, w jakim jeden cel koresponduje z drugim lub z nim koliduje. Cel uważa się za „usatysfakcjonowany” (*satisfied*, nie mylić z *satisfied* — „spełniony”)⁴, jeśli wybrana ewentualność spełnia go w ramach minimum celów wspierających wyznaczonego przez skojarzenie; w przeciwnym razie cel uważany jest za „stłamszony” (*denied*). Ów limit formułowany jest w postaci pięciu następujących wag przypisywanych skojarzeniu: *makes* („powoduje”), *helps* („wspomaga”), *neutral* („nie ma związku”), *hurts* („kłóci się”) i *breaks* („zaprzecza”).

Węzły najwyższego poziomu reprezentują cele wysokopoziomowe specyfikowane przez klienta. W wyniku ich sukcesywnego doskonalenia do postaci celów bardziej szczegółowych uzyskujemy zbiór celów reprezentujących poszczególne cechy systemu — cele te nazywane są celami *realizacyjnymi* (*operationalizing*), dla odróżnienia od celów reprezentujących oryginalne wymagania pozafunkcyjne.

Na rysunku 12.12 widoczny jest fragment grafu ukazujący warianty mechanizmu uwierzytelniania transakcji bankomatowych. Celami wysokopoziomowymi są tu: elastyczność (*Flexibility*), niski koszt (*Low cost*) i bezpieczeństwo (*security*). Ogólnie pojęte bezpieczeństwo dekomponowane jest koniunkcyjnie na składowe: uwierzytelnienie (*Authentication*), poufność (*Confidentiality*) i integralność (*integrity*) — bezpieczny system może zezwalać na dostęp do kont jedynie uprawnionym użytkownikom, dokonywane transakcje muszą być tajne, a każda transakcja musi zakończyć się z zachowaniem integralności stanu kont (awaria systemu nie może spowodować zagubienia ani wykreowania kwoty). Uwierzytelnienie (jako cel) jest dalej dekomponowane dysjunkcyjnie ze względu na środki uwierzytelnienia, którymi mogą być: numer konta wraz z numerem PIN (*Account+PIN*), karta inteligentna wraz z numerem PIN (*SmartCard+PIN*) lub linie papilarne (*FingerPrint*). Każdy z tych trzech ostatnich podcelów jest celem realizacyjnym, na diagramie został więc wyróżniony pogrubioną obwódką. Każdy z nich pozostaje w określonej relacji z innymi celami; na diagramie ograniczyliśmy się do pokazania, jak ma się uwierzytelnianie za pomocą numerów konta i PIN oraz uwierzytelnianie za pomocą linii papilarnych do wymagań elastyczności i niskiego kosztu.

⁴ „Spełnienie” to właściwość kategoryczna: cel może być spełniony albo nie. „Usatysfakcjonowanie” to wielkość mierzalna: dany cel jest tym bardziej „usatysfakcjonowany”, im więcej podcelów jest z nim zgodnych. Odpowiednio duży wskaźnik usatysfakcjonowania jest więc „rozmytym” odpowiednikiem spełnienia — *przyp. tłum.*



Rysunek 12.12. Dekompozycja celów projektowych w ramach frameworku NFR na przykładzie mechanizmu uwierzytelniania transakcji bankomatowych

Gdy w wyniku kompozycji zidentyfikowanych zostanie kilka celów realizacyjnych, programiści wybierają i wartościują wybrane ich podzbiory; na podstawie skojarzeń między celami można wówczas wyrokować o „usatysfakcjonowaniu” lub „stłamszeniu” poszczególnych celów wysokopoziomowych. W przykładzie na rysunku 12.12 jako rozwiązanie wybrany został pierwszy z wymienionych środków uwierzytelnienia (Account+PIN); jak widzimy, „usatysfakcjonowane” zostały cele Authentication i Low cost, zaś „usatysfakcjonowanie” celu Security uzależnione jest od „usatysfakcjonowania” celów Confidentiality i Integrity. Nie jest natomiast „usatysfakcjonowany” cel Flexibility.

Dla typowego, niebanalnego systemu przedstawiony graf dekompozycji rozrasta się i komplikuje do tego stopnia, że do oceniania i porównywania różnych ewentualności rozwiązania konieczne jest użycie narzędzi wspomagających. Jedną z zalet frameworku NFR, której nie będziemy szczegółowo opisywać, jest możliwość wielokrotnego wykorzystywania dokonanej dekompozycji; umożliwia to programistom grupowanie wyników poprzednich eksploracji i ponowne ich wartościowanie w świetle nowych kryteriów.

12.4. Aktywności racjonalizacji — od zagadnień do decyzji

Właściwe zarządzanie racjonalizacją ułatwia programistom radzenie sobie z koniecznymi zmianami. Dokumentując uzasadnienie każdej decyzji, mogą oni łatwiej przeanalizować podjęte wcześniej istotne decyzje pod kątem ich aktualności w obliczu nowych wymagań klienta czy zmian w środowisku. By jednak modele racjonalizacji mogły być rzeczywiście użyteczne, zawarta w nich informacja musi być aktualna, musi posiadać odpowiednią strukturę i musi być łatwo dostępna. W tej sekcji opiszemy związane z tym aktywności, czyli:

- kolekcjonowanie racjonalizacji w ramach zebrań (patrz sekcja 12.4.2),
- rewidowanie modeli racjonalizacji i ich sukcesywne ulepszanie (patrz sekcja 12.4.3),
- wzbogacanie racjonalizacji o nowe elementy podczas wprowadzania zmian do systemu (patrz sekcja 12.4.4),
- rekonstruowanie racjonalizacji ukrywającej się za modelami systemu (patrz sekcja 12.4.5).

Najbardziej krytyczne elementy racjonalizacji rodzą się na etapie projektowania systemu: decyzje podejmowane na tym etapie mają wpływ na każdy podsystem i ich zmiana jest zwykle kosztowna, szczególnie gdy odbywa się w późnych stadiach procesu. Co więcej, racjonalizacja kryjąca się za dekompozycją systemu jest zazwyczaj bardzo skomplikowana, jako że rozciąga się na dużą liczbę zagadnień, takich jak odwzorowanie w węzły sprzętowe, przechowywanie trwałych danych, kontrola dostępu, globalne sterowanie przepływem i warunki graniczne (o czym pisaliśmy w rozdziale 7. „Projekt systemu: realizacja celów projektowych”).

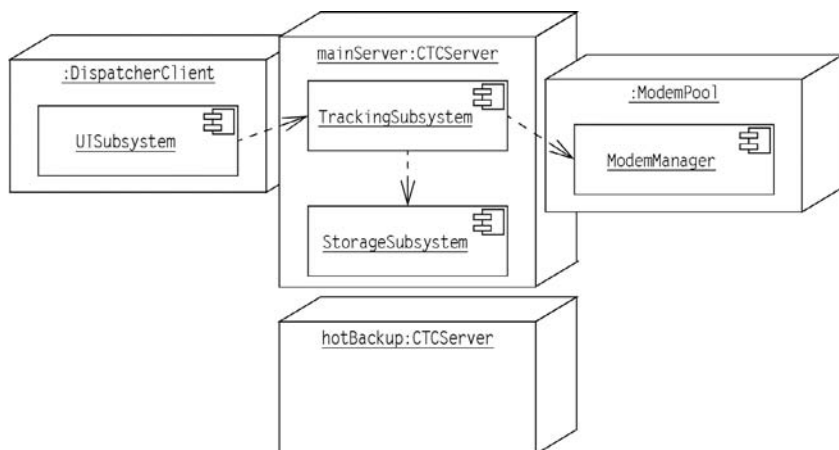
Z tych właśnie względów w tym rozdziale skoncentrujemy się właśnie na etapie projektowania systemu; opisywane techniki kolekcjonowania racjonalizacji wyglądają podobnie na wszystkich innych etapach, począwszy od zbierania wymagań, aż po testy pilotażowe. Późniejsze aktywności opisywać będziemy w kontekście realnego przykładu, jakim jest system CTC; rozpoczniemy więc od przedstawienia projektu tego systemu.

12.4.1. Projekt systemu CTC

Ogólną charakterystykę systemu CTC przedstawiliśmy w sekcji 12.3.1. Wyobraźmy sobie, że uczestniczymy w procesie inżynierii wtórnej, czyli w unowocześnianiu tego systemu, polegającym na zastąpieniu centralnego komputera *mainframe* siecią stacji roboczych. Zamierzamy także rozszerzyć funkcje systemu o takie mechanizmy jak kontrola dostępu i generalnie lepsze wsparcie dla bezpieczeństwa i użyteczności. Znajdujemy się właśnie w środku etapu projektowania systemu i na razie sformułowaliśmy (na podstawie wymagań pozafunkcyjnych) kilka celów projektowych (które wymieniamy, począwszy od najważniejszych).

- **Dostępność.** Awarie systemu nie mogą się zdarzać częściej niż jedna na miesiąc, a powrót systemu do normalnej pracy po awarii nie powinien trwać dłużej niż 10 minut.
- **Bezpieczeństwo.** Żadna encja zlokalizowana poza nastawniami nie może uzyskiwać dostępu do informacji o stanie zarządzanych odcinków izolowanych ani też nie może mieć możliwości manipulowania jakimikolwiek urządzeniami.
- **Użyteczność.** Przeszkolony dyspozytor może pomyłkowo wydawać błędne polecenia nie częściej niż dwa razy dziennie.

Każdy dyspozytor posiada własny węzeł kliencki, zaś całością systemu zarządza para dublujących się serwerów (patrz rysunek 12.13 i tabela 12.1). Serwery te odpowiedzialne są także za przechowywanie trwałych danych; dane te przechowywane są w formie niezależnych plików, które można kopiować offline i następnie importować do bazy danych w celu przetwarzania ich zawartości. Komunikacja systemu z urządzeniami (stacjami przydrożnymi) odbywa się za pośrednictwem modemów zarządzanych przez dedykowaną maszynę. Oprogramowanie *middleware* realizuje dwa rodzaje komunikacji między podsystemami: wywoływanie metod realizujących żądania oraz powiadamianie o zdarzeniach istotnych dla poszczególnych podsystemów. Przed nami zadanie rozwiązania zagadnień związanych z kontrolą dostępu oraz mechanizmami zabezpieczającymi dyspozytorów przed ingerowaniem w ich sekcje dyspozycyjne przez innych dyspozytorów.



Rysunek 12.13. Dekompozycja systemu CTC. Zarządzanie stanem systemu jest zadaniem serwera mainServer, który w razie awarii zastępowany jest przez serwer hotBackup. Serwer wysyła polecenie i odbiera informację o stanie odcinków za pośrednictwem modemów, którymi zarządza węzeł ModemPool

Tabela 12.1. Opis podsystemów systemu CTC z rysunku 12.13

DispatcherClient	Każdemu dyspozytorowi przydzielony jest węzeł DispatcherClient, realizujący interfejs użytkownika.
CTCServer	Zarządza stanem systemu: wysyła dane do urządzeń w terenie i odbiera informacje o ich stanie za pośrednictwem węzła ModemPool. Odpowiedzialny także za przechowywanie trwałych danych (między innymi adresów sieciowych i nazw urządzeń, przydziału dyspozytorów do sekcji dyspozycyjnych, rozkładów jazdy pociągów i tym podobnych). Redundancja w postaci pracy dublujących się serwerów ma na celu zwiększenie dostępności systemu.
ModemPool	Zarządza modemami realizującymi komunikację z urządzeniami w terenie.
ModemManager	Zarządza połączeniami z urządzeniami w terenie i transmituje do nich polecenia.
StorageSubsystem	Zarządza stanem trwałych danych systemu.
TrackingSubsystem	Zarządza stanem poszczególnych odcinków izolowanych, na podstawie informacji uzyskiwanych od urządzeń w terenie. Wysyła (za pośrednictwem węzła ModemManager) polecenia do urządzeń w terenie, bazując na poleceniach otrzymywanych od węzła UISubsystem.
UISubsystem	Odpowiedzialny za wyświetlanie stanu odcinków izolowanych na użytek dyspozytora oraz kontrolowanie poprawności poleceń wydawanych przez dyspozytora przed przekazaniem ich do węzła CTCServer.

12.4.2. Kolekcjonowanie racjonalizacji w ramach zebrań

Zebrań umożliwia programistom prezentowanie, negocjowanie i rozwiązywanie zagadnień w warunkach osobistej komunikacji. Fizyczna obecność dyskutantów jest tu bardzo ważna, dostarcza bowiem sygnały komunikacji niewerbalnej, w postaci oceny nastawienia innych uczestników do problemu i ich motywacji. Negocjowanie i podejmowanie decyzji „na

odległość”, na przykład ze pośrednictwem e-maili, jest mniej efektywne, bo łatwiej wówczas o nieporozumienia, tak więc spotkania „twarzą w twarz” stanowią bardziej naturalny punkt wyjścia do kolekcjonowania racjonalizacji.

Procedury organizowania zebrań i związane z nimi dokumenty (agendy i protokoły) opisaliśmy już w rozdziale 3. „Organizacja projektu i komunikacja”. Agenda, publikowana przed zebraniem, opisuje status problemu i jego aspekty przeznaczone do przedyskutowania. Przebieg zebrania dokumentowany jest w formie protokołu, który opublikowany zostaje wkrótce po zakończeniu zebrania. Wykorzystując koncepcje modelowania zagadnień opisane w sekcji 12.3, tworzymy agendę w kategoriach zagadnień, na temat których chcemy dyskutować i rozwiązania których chcemy poszukiwać. Jako cel zebrania określamy rozwiązanie tychże zagadnień, a także wszystkich innych podzagadnień, które wynikną w trakcie dyskusji. Protokołowi zebrania nadajemy strukturę uwzględniającą **propozycje** zgłaszane w trakcie zebrania, uzgodnione **kryteria** ich oceniania oraz **argumenty** wysuwane na poparcie poszczególnych propozycji i przeciwko nim. Podjęte decyzje formułujemy w kategoriach rozstrzygnięć i implementujących je elementów działania. Przegląd statusu tych ostatnich będzie prawdopodobnie jednym z punktów następnych zebrań.

Jako przykład rozpatrzmy zagadnienia związane z kontrolą dostępu w systemie CTC. Musimy w związku z tym zorganizować spotkanie z udziałem zespołu architektonicznego oraz programistów odpowiedzialnych za podsystemy `UISubsystem`, `TrackingSubsystem` i `NotificationService`. Alice, facilitator zespołu architektonicznego, publikuje poniższą agendę.

AGENDA: Integracja kontroli dostępu i powiadamiania

Gdzie i kiedy

Data: 13 września

Początek: 16:30

Koniec: 17:30

Pokój: Train Hall, 3420

Role

Pierwszy facilitator: Alice

Chronometrzysta: Dave

Protokolant: Ed

1. Cel

Pierwszą rewizję projektów odwzorowania sprzętowo-programowego i systemu przechowywania danych mamy już za sobą, konieczne jest teraz zdefiniowanie modelu kontroli dostępu i jego integracji z istniejącymi podsystemami, między innymi `NotificationService` i `TrackingSubsystem`.

2. Oczekiwane wyniki

Rozwiązanie zagadnień związanych z integracją kontroli dostępu i powiadamiania.

3. Rozpowszechniana informacja [czas: 15 minut]

AI [1]: Dave: Analiza modelu kontroli dostępu realizowanej przez *middleware*.

4. Dyskusja [czas: 35 minut]

I [1]: Czy dyspozytor ma prawo oglądać sekcje dyspozycyjne kontrolowane przez innych dyspozytorów?

I [2]: Czy dyspozytor ma prawo wysyłania poleceń do urządzeń znajdujących się w sekcjach dyspozycyjnych kontrolowanych przez innych dyspozytorów?

I [3]: W jaki sposób kontrola dostępu powinna być zintegrowana z podsystemami `TrackingSubsystem` i `NotificationService`?

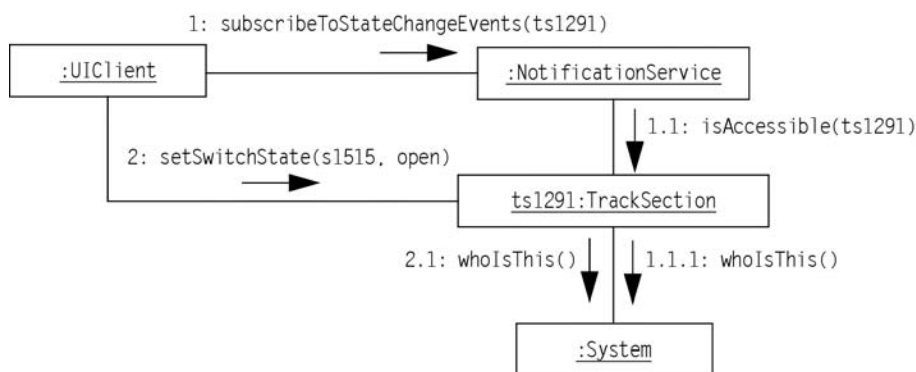
5. Podsumowanie [czas: 5 minut]

Przegląd i przyporządkowanie nowych elementów działania.

Uwagi krytyczne

W czasie zebrania dyskutowany będzie element działania stanowiący wynik poprzedniego zebrania zespołu architektonicznego: AI[1]: *Analiza modelu kontroli dostępu realizowanej przez middleware*. *Middleware* dostarcza podstawowe bloki dla uwierzytelniania i szyfrowania informacji, nie wnosi nic poza tym do modelu kontroli dostępu. Zagadnienia I[1] i I[2] zostaną rozwiązane dość szybko w oparciu o wiedzę z dziedziny aplikacyjnej: każdy dyspozytor może oglądać wszystkie sekcje dyspozycyjne, lecz możliwość manipulowania urządzeniami ograniczona jest do jego własnej sekcji. Zagadnienie I[3] (*W jaki sposób kontrola dostępu powinna być zintegrowana z podsystemami TrackingSubsystem i NotificationService?*) nie jest jednak tak oczywiste i stanowić będzie temat do dyskusji.

Trwa zebranie. Dave, programista odpowiedzialny za komponent *NotificationService*, proponuje zintegrowanie kontroli dostępu z klasą *TrackSection* (patrz rysunek 12.14 i tabela 12.2). Klasa *TrackSection* powinna w związku z tym utrzymywać listę uprawnień, określającą uprawnienia poszczególnych dyspozytorów do odczytywania i modyfikowania stanu poszczególnych sekcji dyspozycyjnych. Podsystem zainteresowany zdarzeniami zachodzącymi w ramach klasy *TrackSection* powinien subskrybować powiadamianie o tych zdarzeniach za pomocą usługi *NotificationService*. Usługa ta powinna sprawdzać, czy dyspozytor żądający informacji o danej sekcji dyspozycyjnej ma uprawnienia do (przynajmniej) odczytywania tego stanu.

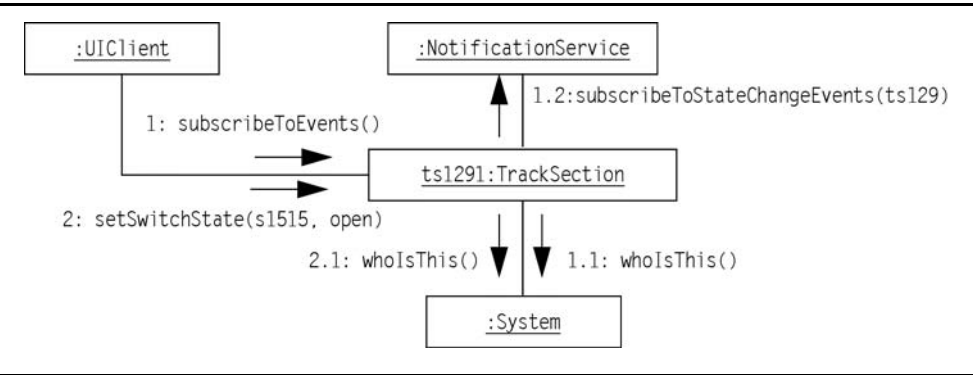


Rysunek 12.14. Propozycja P[1]: Dostęp do klasy *TrackSection* kontrolowany jest za pomocą listy uprawnień utrzymywanej przez ten komponent. W oparciu o tę listę usługa *NotificationService* sprawdza, czy żądający informacji dyspozytor ma prawo do jej uzyskiwania

Alice, programistka odpowiedzialna za podsystem *TrackSubsystem*, którego częścią jest klasa *TrackSection*, proponuje odwrócenie zależności między klasą *TrackSection* a usługą *NotificationService* (patrz rysunek 12.15 i tabela 12.3). Zgodnie z jej propozycją, podsystem *UIClient* nie powinien mieć bezpośredniego dostępu do usługi *NotificationService*, nawet przy subskrybowaniu informacji. W celu dokonania subskrypcji podsystem *UIClient* wywoła metodę *subscribeToEvents()* klasy *TrackSection*, która to metoda po pozytywnym zweryfikowaniu uprawnień wywoła operację *subscribeToStateChangeEvents()* usługi *NotificationService*. Dzięki temu uzyskamy scentralizowanie w ramach jednej klasy

Tabela 12.2. Opis elementów modelu widocznego na rysunku 12.14

NotificationService	Usługa propagująca informację o zmianie stanu sekcji dyspozycyjnej (TrackSection). W celu otrzymywania informacji dotyczącej określonej sekcji za pośrednictwem tej usługi zainteresowany podsystem powinien tę informację subskrybować. Może to jednak uczynić tylko podsystem posiadający uprawnienia do odczytywania informacji na temat wspomnianej sekcji. Uprawnienia te odczytywane są za pomocą operacji isAccessible() klasy TrackSection.
System	System odpowiedzialny za bezpieczne sprawdzanie, na podstawie informacji przekazanej przez komponent UIClient, który dyspozytor żąda informacji. Obiekt TrackSection weryfikuje uprawnienia bieżącego dyspozytora za pomocą operacji whoIsThis().
TrackSection	Klasa reprezentująca sekcję dyspozycyjną, złożoną z odcinków reprezentowanych przez obiekty TrackCircuits i powiązane z nimi obiekty Devices. Na poziomie klasy TrackSection realizowana jest kontrola dostępu, w oparciu o listę utrzymywaną przez każdy obiekt tej klasy.
UIClient	Podsystem odpowiedzialny za wyświetlanie stanu sekcji dyspozycyjnych i przyjmowanie poleceń od dyspozytora.



Rysunek 12.15. Propozycja P[2]: klient zostaje pozbawiony dostępu do usługi NotificationService, całość operacji związanych z kontrolą dostępu skupiona zostaje w klasie TrackSection, za pośrednictwem której klient subskrybuje żadaną informację

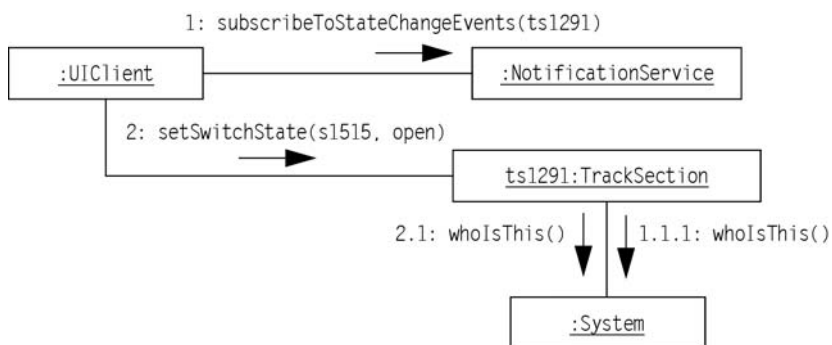
wszystkich „chronionych”⁵ operacji i całokształtu kontroli dostępu. Ponadto, gdy zmienia się uprawnienia dostępu, klasa TrackSection może we własnym zakresie anulować subskrypcje pozostające w sprzeczności z nowymi uprawnieniami.

⁵ Autorzy używają tu przymiotnika „chroniona” (*protected*) na określenie faktu, że wykonanie danej operacji uzależnione jest od uprawnień podsystemu wywołującego tę operację, czyli poprzedzone weryfikacją tych uprawnień (operacja „chroniona” jest przez mechanizm kontroli dostępu). Nie należy więc tym razem kojarzyć słowa „chroniona” z innym jego znaczeniem — poziomem widzialności *protected* elementu klasy (patrz sekcja 9.3.2) — *przyp. tłum.*

Tabela 12.3. Opis elementów modelu widocznego na rysunku 12.15; zmiany w stosunku do tabeli 12.2 wyróżnione zostały kursywą

NotificationService	Usługa propagująca informację o zmianie stanu sekcji dyspozycyjnej (TrackSection). W celu otrzymywania informacji dotyczącej określonej sekcji za pośrednictwem tej usługi zainteresowany podsystem powinien tę informację subskrybować. Może to jednak uczynić tylko podsystem posiadający uprawnienia do odczytywania informacji na temat wspomnianej sekcji. Uprawnienia te odczytywane są za pomocą operacji <code>isAccessible()</code> klasy <code>TrackSection</code> . <i>Usługa dostępna jest bezpośrednio jedynie dla klasy <code>TrackSection</code>, podsystem <code>UIClient</code> nie ma do niej dostępu, subskrybuje on żadaną informację przez wywołanie metody <code>subscribeToEvents()</code> klasy <code>TrackSection</code>.</i>
TrackSection	Klasa reprezentująca sekcję dyspozycyjną, złożoną z odcinków reprezentowanych przez obiekty <code>TrackCircuits</code> i powiązane z nimi obiekty <code>Devices</code> . Na poziomie klasy <code>TrackSection</code> realizowana jest kontrola dostępu, w oparciu listę utrzymywaną przez każdy obiekt tej klasy. <i>Klient, żądający informacji o stanie sekcji, powinien tę informację subskrybować, wywołując metodę <code>subscribeToEvents()</code> klasy <code>TrackSection</code>. Klasa <code>TrackSection</code> po pozytywnym zweryfikowaniu uprawnień podsystemu żądającego informacji wywołuje operację <code>subscribeToTrackSectionEvents()</code> usługi <code>NotificationService</code>.</i>

Propozycja, którą zgłosił Ed, opiera się na ważnym spostrzeżeniu, iż kontroli dostępu powinny podlegać jedynie próby modyfikowania stanu sekcji dyspozycyjnych, ponieważ z założenia oglądanie stanu wszystkich sekcji ma być dostępne dla wszystkich dyspozytorów. Zakładając, że modyfikacja stanu sekcji dokonuje się wyłącznie przez wywołanie odpowiednich metod, i biorąc pod uwagę fakt, że usługa `NotificationService` wykorzystywana jest jedynie do powiadamiania o zmianach stanu, a nie uczestniczy w modyfikacji wspomnianego stanu, łatwo zauważyć, iż usługa ta nie pozostaje w żadnej relacji do mechanizmu kontroli dostępu. Nie trzeba zatem integrować jej z tym mechanizmem. Otrzymujemy tym samym propozycję będącą ulepszeniem propozycji Dave’a (patrz rysunek 12.16 i tabela 12.4).



Rysunek 12.16. Propozycja P[3]: jedynie próby modyfikowania stanu obiektu `TrackSection` podlegają kontroli dostępu, więc usługa `NotificationService` przestaje być zależna od tego mechanizmu

Tabela 12.4. Opis elementów modelu widocznego na rysunku 12.16; zmiany w stosunku do tabeli 12.2 wyróżnione zostały *przekreśloną kursywą*

NotificationService	Usługa propagująca informację o zmianie stanu sekcji dyspozycyjnej (TrackSection). W celu otrzymywania informacji dotyczącej określonej sekcji za pośrednictwem tej usługi zainteresowany podsystem powinien tę informację subskrybować. <i>Może to jednak uczynić tylko podsystem posiadający uprawnienia do odczytywania informacji na temat wspomnianej sekcji. Uprawnienia te odczytywane są za pomocą operacji isAccessible() klasy TrackSection.</i>
---------------------	--

Zespół architektoniczny zdecydował o wyborze propozycji Eda, dzięki jej prostocie. Ed jako protokolant sporządza przedstawiony poniżej chronologiczny protokół.

CHRONOLOGICZNY PROTOKÓŁ: Integracja kontroli dostępu i powiadamiania	
Gdzie i kiedy	Role
Data: 13 września	Pierwszy facilitator: Alice
Początek: 16:30	Chronometrażysta: Dave
Koniec: 18:00	Protokolant: Ed
Pokój: Train Hall, 3420	
1. Cel	
Pierwszą rewizję projektów odwzorowania sprzętowo-programowego i systemu przechowywania danych mamy już za sobą, konieczne jest teraz zdefiniowanie modelu kontroli dostępu i jego integracji z istniejącymi podsystemami, między innymi NotificationService i TrackingSubsystem.	
2. Oczekiwane wyniki	
Rozwiązanie zagadnień związanych z integracją kontroli dostępu i powiadamiania.	
3. Rozpowszechniana informacja	
AI [1]: Dave: Analiza modelu kontroli dostępu realizowanej przez <i>middleware</i> . Status: <i>Middleware</i> udostępnia silne mechanizmy szyfrowania i uwierzytelniania, poza tym nie wprowadza żadnych ograniczeń do modelu dostępu; konieczne jest więc zaimplementowanie reguł dostępu na serwerze.	
4. Dyskusja	
I [1]: Czy dyspozytor ma prawo oglądać sekcje dyspozycyjne kontrolowane przez innych dyspozytorów? Ed: Tak, wynika to ze specyfikacji CTC.	
I [2]: Czy dyspozytor ma prawo wysyłania poleceń do urządzeń znajdujących się w sekcjach dyspozycyjnych kontrolowanych przez innych dyspozytorów? Zoe: Nie, tylko dyspozytor przydzielony do danej sekcji dyspozycyjnej ma prawo manipulować urządzeniami tej sekcji. Zauważmy przy tym, że możliwa jest dynamiczna zmiana przydziału dyspozytorów do sekcji. Ed: To też wynika ze specyfikacji CTC.	

I[3]: W jaki sposób kontrola dostępu powinna być zintegrowana z podsystemami `TrackingSubsystem` i `NotificationService`?

Dave: W klasie `TrackSection` zaimplementowana jest lista uprawnień. Usługa powiadamiania wykorzystuje tę listę do sprawdzania, które podsystemy mają prawo uzyskiwać informację o stanie tej sekcji.

Alice: Prawdopodobnie powinniśmy odwrócić kierunek zależności między usługą powiadamiania a klasą `TrackSection`. Klient `UIClient` nie powinien kontaktować się bezpośrednio z usługą powiadamiania, lecz tylko z klasą `TrackSection`, która przed udostępnieniem informacji usłudze powiadamiania zweryfikuje uprawnienia podsystemu żądającego tej informacji. W ten oto sposób skupimy w jednym miejscu wszystkie mechanizmy zależne od kontroli dostępu.

Dave: Dzięki takiemu rozwiązaniu klasa `TrackSection` będzie mogła automatycznie zweryfikować istniejące subskrypcje w przypadku zmodyfikowania listy uprawnień.

Ed: Zauważcie, że wszyscy dyspozytorzy mogą oglądać wszystkie sekcje dyspozycyjne, a więc kontrola dostępu konieczna jest tylko przy próbie modyfikowania ich stanu. Usługa powiadamiania może więc działać bez kontekstu kontroli dostępu.

Alice: Ale uwzględnienie możliwości powiązania usługi powiadamiającej z kontrolą dostępu daje bardziej ogólne rozwiązanie.

Ed: Jednocześnie bardziej skomplikowane. Oddzielmy więc w tej chwili powiadamianie od kontroli dostępu i powróćmy do problemu, gdy zmienią się wymagania.

Alice: Dobrze, muszę zatem zrewidować API podsystemu `TrackingSubsystem`.

5. Podsumowanie

AI[2]: Alice: Zaprojektowanie kontroli dostępu dla podsystemu `TrackingSubsystem` w oparciu o uwierzytelnianie i szyfrowanie udostępniane przez *middleware*.

Ed utworzył swój protokół poprzez wstawienie do agendy informacji o przebiegu dyskusji dotyczącej poszczególnych zagadnień. Zapis tej dyskusji ma postać chronologicznej listy wypowiedzi poszczególnych dyskutantów. Większość tych wypowiedzi łączy propozycje i argumenty na ich rzecz (oraz przeciwko niektórym innym propozycjom), aby więc protokół był zgodny z modelem zagadnień, Ed nadaje mu odpowiednią strukturę, czego efekt widoczny jest poniżej.

STRUKTURALNY PROTOKÓŁ: Integracja kontroli dostępu i powiadamiania

Gdzie i kiedy

Data: 13 września

Początek: 16:30

Koniec: 18:00

Pokój: Train Hall, 3420

Role

Pierwszy facilitator: Alice

Chronometrażysta: Dave

Protokolant: Ed

1. Cel

Pierwszą rewizję projektów odwzorowania sprzętowo-programowego i systemu przechowywania danych mamy już za sobą, konieczne jest teraz zdefiniowanie modelu kontroli dostępu i jego integracji z istniejącymi podsystemami, między innymi `NotificationService` i `TrackingSubsystem`.

2. Oczekiwane wyniki

Rozwiązanie zagadnień związanych z integracją kontroli dostępu i powiadamiania.

3. Rozpowszechniana informacja

AI [1]: Dave: Analiza modelu kontroli dostępu realizowanej przez *middleware*.

Status: *Middleware* udostępnia silne mechanizmy szyfrowania i uwierzytelniania, poza tym nie wprowadza żadnych ograniczeń do modelu dostępu; konieczne jest więc zaimplementowanie reguł dostępu na serwerze.

4. Dyskusja

I [1]: Czy dyspozytor ma prawo oglądać sekcje dyspozycyjne kontrolowane przez innych dyspozytorów?

R [1]: Tak (wynika to ze specyfikacji CTC i potwierdzone jest przez Zoe, użytkownika testującego).

I [2]: Czy dyspozytor ma prawo wysyłania poleceń do urządzeń znajdujących się w sekcjach dyspozycyjnych kontrolowanych przez innych dyspozytorów?

R [2]: Nie, tylko dyspozytor przydzielony do danej sekcji ma prawo manipulować urządzeniami tej sekcji. Zauważmy przy tym, że możliwa jest dynamiczna zmiana przydziału dyspozytorów do sekcji (wynika to ze specyfikacji CTC i potwierdzone jest przez Zoe, użytkownika testującego).

I [3]: W jaki sposób kontrola dostępu powinna być zintegrowana z podsystemami *TrackingSubsystem* i *NotificationService*?

P [3.1]: W klasie *TrackSection* zaimplementowana jest lista uprawnień. Dany podsystem, aby subskrybować powiadamianie o zdarzeniach tej sekcji, wysyła żądanie do usługi powiadamiania, która z kolei kontaktuje się z przedmiotową sekcją w celu zweryfikowania uprawnień wspomnianego podsystemu.

P [3.2]: W klasie *TrackSection* skupione zostają wszystkie chronione operacje. Podsystem zamierzający subskrybować informacje o stanie danej sekcji, kontaktuje się z tą właśnie sekcją, która po zweryfikowaniu jego uprawnień wywołuje usługę powiadamiania w celu dokonania rzeczzonej subskrypcji.

A [3.1] na rzecz P [3.2]: Chronione operacje i kontrola dostępu zostają zintegrowane w ramach jednej klasy.

P [3.3]: Nie trzeba kontrolować dostępu związanego z odczytywaniem informacji, *UIClient* może więc żądać subskrypcji od usługi powiadamiania bez względu na swe uprawnienia — usługa powiadamiania nie musi w ogóle interesować się listami uprawnień.

A [3.2] na rzecz P [3.3]: Dyspozytorzy mogą bez ograniczeń oglądać wszystkie sekcje (patrz R [1]).

A [3.3] na rzecz P [3.3]: Prostota.

R [3]: P [3.3]. Patrz AI [2].

5. Podsumowanie

AI [2]: Alice: Zaprojektowanie kontroli dostępu dla podsystemu *TrackingSubsystem* w oparciu o uwierzytelnianie i szyfrowanie udostępniane przez *middleware*, zgodnie z rozstrzygnięciem R [3] tego protokołu.

Reasumując, rezultatem spotkania poświęconego kontroli dostępu było ustalenie, że:

- każdy dyspozytor może oglądać stan wszystkich sekcji, lecz modyfikować może wyłącznie stan swej własnej sekcji,

- kontrola dostępu odbywa się na podstawie listy uprawnień skojarzonej z każdym obiektem klasy `TrackSection`,
- usługa powiadamiania (`NotificationService`) została odseparowana od kontroli dostępu, bowiem informacja o stanie dowolnej sekcji jest dostępna dla każdego dyspozytora.

Skupiając się na modelu zagadnień, możemy ponadto zauważyć, że:

- rozważano integrację usługi powiadamiania (`NotificationService`) z mechanizmami kontroli dostępu,
- scentralizowanie wszystkich chronionych metod w klasie `TrackSection` było akceptowaną regułą.

Dwie ostatnie informacje są typowymi informacjami wchodzącym w skład racjonalizacji; zazwyczaj informacje tego typu uważane są za mało istotne, ich utrwalenie w modelu okazuje się jednak korzystne w dłuższej perspektywie, gdy pojawia się konieczność zmian.

12.4.3. Asynchroniczne kolekcjonowanie racjonalizacji

Dyskusje na zebraniach opierają się o informację kontekstową. Uczestnicy zebrania dysponują pokaznym zasobem wiedzy na temat systemu, jego przeznaczenia i projektu. Facilitator z konieczności ukierunkowuje przebieg zebrania na wąski zakres zagadnień oczekujących na rozwiązanie. I tak na przykład wszyscy uczestnicy opisywanego w poprzedniej sekcji zebrania z pewnością znają przeznaczenie systemu CTC, jego elementy funkcjonalne, cele projektowe i aktualny kształt jego dekompozycji na podsystemy. Protokół z zebrania odzwierciedla jedynie dyskutowane na tym zebraniu zagadnienia, pomijając cały kontekst dyskusji. I niestety, wskutek tego kontekst ów ulega z czasem zatraceniu i protokoły stają się coraz mniej zrozumiałe.

Temu właśnie problemowi można zaradzić przy użyciu modelowania zagadnień. W rozdziale 3. „Organizacja projektu i komunikacja” opisywaliśmy wykorzystywanie oprogramowania wspomagającego pracę grupową (*groupware*) w warunkach komunikacji asynchronicznej. Integrując przygotowanie do zebrania i jego przebieg z komunikacją asynchroniczną, zyskujemy możliwość utrwalania także informacji kontekstowej.

Załóżmy na przykład, że programistka imieniem Mary, odpowiedzialna za podsystem `UISubsystem`, nie mogła uczestniczyć w zebraniu poświęconym kontroli dostępu. Zapoznała się z agendą tego zebrania i protokołem (które dostępne były na grupie dyskusyjnej zespołu architektonicznego) i choć znakomicie rozumie wynik zebrania, dyskusja na temat usługi powiadamiania (`NotificationService`) wydaje się jej nie do końca jasna: zgodnie z argumentem `A[3.3]` dla propozycji `P[3.3]`, ponieważ każdy dyspozytor może obserwować stan każdej sekcji, wszystkie zdarzenia w systemie są publicznie widoczne, nie ma więc potrzeby ujmować informacji o nich w ramy mechanizmu kontroli dostępu. Aby jednak kontrola taka była skuteczna w odniesieniu do *modyfikowania* stanu sekcji, konieczne jest założenie, że zdarzenia zachodzące w jednym podsystemie nie mogą powodować zmiany stanu innych podsystemów, w tym poszczególnych sekcji `TrackSection`. Mary chce się upewnić, że taki właśnie jej tok

myślenia jest prawidłowy i w związku z tym wysła na grupę dyskusyjną widoczny poniżej post, w którym dodatkowo proponuje, by w konsekwencji zabronić usłudze TrackingService subskrybowania jakichkolwiek zdarzeń.

Grupy dyskusyjne: ctc.architecture.discuss

Temat:	Data
I[1]: Czy dyspozytor ma prawo oglądać sekcje dyspozycyjne kontrolowane przez innych dyspozytorów?	14 wrz
I[2]: Czy dyspozytor ma prawo wysyłania poleceń do urządzeń znajdujących się w sekcjach dyspozycyjnych kontrolowanych przez innych dyspozytorów?	14 wrz
I[3]: W jaki sposób kontrola dostępu powinna być zintegrowana z podsystemami TrackingSubsystem i NotificationService?	14 wrz
P[3.1]: W klasie TrackSection zaimplementowana jest lista uprawnień.	14 wrz
P[3.2]: W klasie TrackSection skupione zostają operacje subskrypcyjne.	14 wrz
+A[3.1]: Rozszerzalność.	14 wrz
+A[3.2]: Scentralizowanie wszystkich chronionych operacji.	14 wrz
P[3.3]: Usługa NotificationService nie jest związana z kontekstem kontroli dostępu.	14 wrz
+A[3.3]: Dyspozytorzy mogą oglądać wszystkie sekcje.	14 wrz
+A[3.3]: Prostota.	14 wrz

Od: Mary

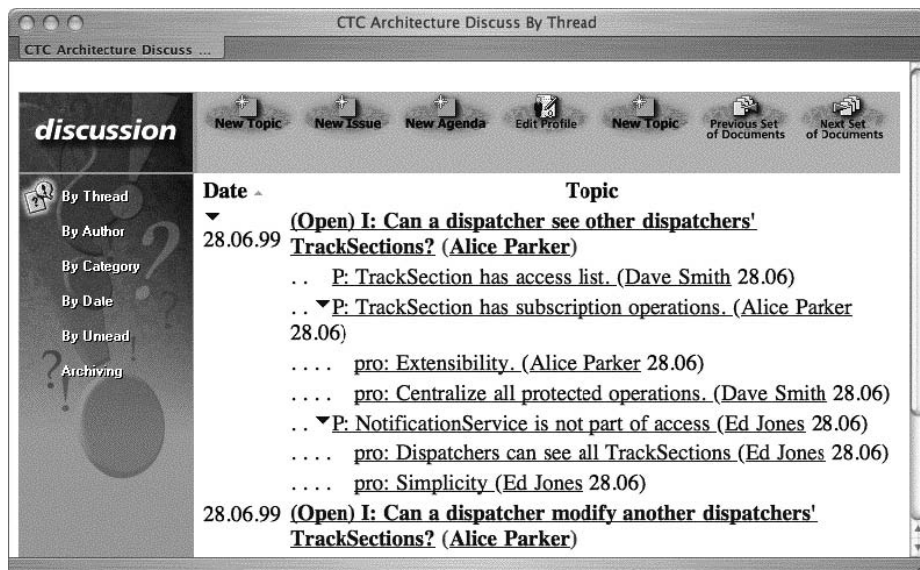
Grupy dyskusyjne: ctc.architecture.discuss

Temat: Zagadnienie wynikowe: Czy dopuszczalne jest realizowanie żądań przy użyciu powiadamiania?

Data: 15 wrz, czw, 13:12:48 -0400

I[4] w odpowiedzi na A[3.3]: w związku z listami uprawnień w kontekście możliwości:
 > Dyspozytorzy mogą oglądać stan wszystkich sekcji, mają więc dostęp do wszystkich zdarzeń. Zakładamy zatem, że stan sekcji TrackSection nie jest sterowany zdarzeniami i zdarzenia wykorzystywane są tylko do informowania podsystemów o zmianie stanu w innych podsystemach. Czy dla większego bezpieczeństwa nie powinniśmy zabronić usłudze TrackingService subskrybowania jakichkolwiek informacji o zdarzeniach?

Wykorzystywanie tego samego modelu zagadnień zarówno na zebraniach, jak i na potrzeby dyskusji online umożliwia integrację informacji związanej z racjonalizacją. Mimo iż integrację tę zapewnić można przy użyciu prostych technologii, takich jak grupy dyskusyjne, poszczególne elementy — modele zagadnień, agendy oraz protokoły zebrań i związane z nimi komunikaty — mogą być integrowane także za pomocą zaawansowanych narzędzi *groupware*, jak Lotus Notes czy wieloużytkowa baza zagadnień dostępna w sieci WWW (czego przykład widzimy na rysunku 12.17). Ustanawiając odpowiednie procedury wspomnianej integracji, możemy kolekcjonować obszerną informację związaną z racjonalizacją. I tu pojawia się kolejne wyzwanie — utrzymywanie tej informacji w stanie aktualnym, stosownie do dokonywanych zmian.



Rysunek 12.17. Przykład bazy zagadnień — szablon bazy danych (LN IBIS w Domino Lotus Notes). W tej bazie za pomocą formularzy WWW programiści mogą rejestrować zagadnienia, propozycje, argumenty i rozstrzygnięcia

12.4.4. Racjonalizacja dyskutowanych zmian

Modele racjonalizacji pomagają programistom uporać się z problemami wynikającymi z konieczności wprowadzania zmian do systemu; niestety, sama racjonalizacja też jest przedmiotem zmian, gdy rozważamy adekwatność podjętych decyzji do nowych warunków. Innymi słowy, projektując rozwiązanie problemu pojawiającego się na przykład w związku ze zmianami w wymaganiach klienta, musimy nie tylko dostarczyć racjonalizacji dla nowych zmian wprowadzanych do systemu, lecz także powiązać nowe elementy racjonalizacji z tymi, które kryją się za obecnym kształtem systemu.

Załóżmy dla przykładu, że klient kontraktujący system CTC zmienił swe wymagania dotyczące kontroli dostępu: dotychczas wszyscy dyspozytorzy mogli bez ograniczenia oglądać stan wszystkich sekcji, klient uznał jednak, że dla danego dyspozytora interesujący może być co najwyżej stan sekcji przylegających do jego własnej sekcji — stan innych sekcji jest dla niego obojętny z punktu widzenia wykonywanych zadań.

Stajemy zatem przed koniecznością zmodyfikowania projektu mechanizmów kontroli dostępu, w związku z czym organizowane jest zebranie zespołu architektonicznego. Należy przede wszystkim przeanalizować racjonalizację stojącą za obecnym kształtem wspomnianych mechanizmów; Alice, główny facilitator zespołu architektonicznego, publikuje widoczną poniżej agendę.

AGENDA: Zrewidowanie kontroli dostępu**— dyspozytorzy mogą oglądać tylko przylegające sekcje dyspozycyjne****Gdzie i kiedy****Data:** 13 października**Początek:** 16:30**Koniec:** 17:30**Pokój:** Signal Hall, 2300**Role****Pierwszy facilitator:** Alice**Chronometrażysta:** Dave**Protokolant:** Ed**1. Cel**

Klient żąda, by dyspozytorzy mogli oglądać tylko sekcje przylegające do ich własnych sekcji.

2. Oczekiwane wyniki

Zrewidowanie mechanizmów kontroli dostępu w związku z powyższą zmianą w wymaganiach klienta.

3. Rozpowszechniana informacja [czas: 15 minut]

AI [1]: Dave: Odtworzenie racjonalizacji kryjącej się za obecnym projektem kontroli dostępu.

4. Dyskusja [czas: 35 minut]

I [1]: W jaki sposób zrewidować istniejące mechanizmy kontroli dostępu, by ograniczyć dyspozytorom dostęp wyłącznie do sekcji przylegających do ich własnych sekcji?

5. Podsumowanie [czas: 5 minut]

Przegląd i przyporządkowanie nowych elementów działania.

Uwagi krytyczne.

W trakcie zebrania Dave prezentuje racjonalizację dyskutowaną na poprzednim zebraniu i na forum grupy dyskusyjnej zespołu architektonicznego. Zespół architektoniczny potwierdza nieaktualność założenia, że dyspozytorzy mogą bez ograniczeń oglądać wszystkie sekcje — zgodnie z obecnymi wymaganiami możliwość ta ograniczona jest tylko do sekcji przylegających. Analizując protokół z poprzedniego zebrania, za najbardziej przydatną dla nowych warunków uznają oni propozycję P[2] (patrz rysunek 12.15): wszystkie chronione operacje scentralizowane są w klasie `TrackSection` i to ona staje się głównym obiektem wszelkich zmian wynikających z nowych wymagań dotyczących kontroli dostępu. Niestety, implementacja istniejącego projektu jest już mocno zaawansowana i przyjęcie propozycji P[2] jako rozstrzygnięcia wiązałoby się z drastycznymi zmianami istniejącego kodu, czego programiści chcieliby raczej uniknąć. Alicja optuje więc za propozycją P[1] (patrz rysunek 12.14): nie potrzeba wprowadzać zmian do podsystemu `UIClient`, bowiem interfejsy klas `TrackSection` i `NotificationService` pozostają niezmienione. Zmiany wymaga natomiast implementacja usługi `NotificationService`: w celu weryfikacji praw dostępu usługa ta kontaktuje się z klasą `TrackSection`; gdy natomiast wprowadzone zostaną zmiany do listy uprawnień utrzymywanej przez obiekt klasy `TrackSection`, ten kontaktuje się z usługą `NotificationService`, by anulować (ewentualne) nieprzysługujące już subskrypcje. Wadą takiego rozwiązania jest cykliczna zależność obu klas `TrackSection` i `NotificationService`, jednak modyfikacja istniejącego kodu sprowadzona zostaje do rozmiarów minimalnych.

I ta właśnie propozycja wybrana zostaje jako rozstrzygnięcie przez zespół architektoniczny. Ed sporządza widoczny poniżej strukturalny protokół (jego chronologiczny pierwotny wzór darujemy sobie ze względu na oszczędność miejsca).

STRUKTURALNY PROTOKÓŁ: Zrewidowanie kontroli dostępu — dyspozytorzy mogą oglądać tylko przylegające sekcje

Gdzie i kiedy

Data: 13 października

Początek: 16:30

Koniec: 17:30

Pokój: Signal Hall, 2300

Role

Pierwszy facilitator: Alice

Chronometrażysta: Dave

Protokolant: Ed

1. Cel

Klient żąda, by dyspozytorzy mogli oglądać tylko sekcje przylegające do ich własnych sekcji.

2. Oczekiwane wyniki

Zrewidowanie mechanizmów kontroli dostępu w związku z powyższą zmianą w wymaganiach klienta.

3. Rozpowszechniana informacja

AI [1]: Dave: Odtworzenie racjonalizacji kryjącej się za obecnym projektem kontroli dostępu.

Wynik: Odtworzenie zagadnień I [1] (z 13 września) i I [2] (z 15 września):

I [1]: *Jak powinno się zintegrować kontrolę dostępu z klasami TrackSection i NotificationService?(protokół z 14 września)*

P[3.1]: *W klasie TrackSection zaimplementowana jest lista uprawnień. Dany podsystem, aby subskrybować powiadamianie o zdarzeniach tej sekcji, wysłał żądanie do usługi powiadamiania, która z kolei kontaktuje się z przedmiotową sekcją w celu zweryfikowania uprawnień wspomnianego podsystemu.*

P[3.2]: *W klasie TrackSection skupione zostają wszystkie chronione operacje. Podsystem zamierzający subskrybować informacje o stanie danej sekcji kontaktuje się z tą właśnie sekcją, która po zweryfikowaniu jego uprawnień wywołuje usługę powiadamiania w celu dokonania rzeczowej subskrypcji.*

A[3.1] na rzecz P[3.2]: *Rozszerzalność.*

A[3.2] na rzecz P[3.2]: *Scentralizowanie wszystkich chronionych operacji w jednej klasie.*

P[3.3]: *Nie ma potrzeby kontrolowania dostępu związanego z odczytywaniem informacji, UIClient może więc żądać subskrypcji od usługi powiadamiania bez względu na swe uprawnienia — usługa powiadamiania nie musi w ogóle interesować się listami uprawnień.*

A[3.3] na rzecz P[3.3]: *Prostota.*

R[3]: *P[3.3]. Patrz AI [2].*

I [2]: *Czy dopuszczalne jest realizowanie żądań przy użyciu powiadamiania? (post Mary z 15 września na grupie dyskusyjnej)*

R[2]: *Mechanizm powiadamiania może być wykorzystywany wyłącznie do informowania o zmianach stanu podsystemów. Obiekty TrackSection — i w ogóle cały podsystem TrackingSubsystem — nie mogą zmieniać swego stanu wskutek powiadomień.*

4. Dyskusja

I [1]: Jak należy zrewidować projekt kontroli dostępu, by ograniczyć dostępność sekcji dla dyspozytora tylko do przyległych sekcji?

- P[1.1]: Skupienie chronionych operacji, w tym subskrybowania, w ramach klasy `TrackSection`, jak w P[3.2] w protokole z 13 września.
A[1.1] przeciwko P[1.1]: To wymagałoby modyfikacji wszystkich podsystemów subskrybujących powiadamianie o zdarzeniach, ponieważ operacja subskrybowania zostałaby przeniesiona z klasy `NotificationService` do klasy `TrackSection`.
- P[1.2]: Usługa `NotificationService` wysyła do obiektu `TrackSection` żądanie zweryfikowania uprawnień, natomiast obiekt `TrackSection` wysyła do usługi `NotificationService` żądania anulowania subskrypcji, jeśli wynika to ze zmienionych uprawnień dostępu — jak w P[3.1] w protokole z 13 września.
A[1.2] na rzecz P[1.2]: Minimalne zmiany w istniejącej implementacji.
A[1.2] przeciwko P[1.2]: Cykliczna zależność między klasami.
- R[1]: P[1.2], patrz AI[2] i AI[3]

5. Podsumowanie

- AI[2]: Alice: Zmiana implementacji klasy `TrackSection` pod kątem odwoływania subskrypcji tych dyspozytorów, którzy tracą do niej uprawnienia w wyniku zmian na liście dostępu.
- AI[3]: Dave: Zmiana usługi `NotificationService` — w celu weryfikowania uprawnień zwraca się ona do docelowego obiektu `TrackSection`.

Protokół ten przydatny jest do dwóch celów: określa racjonalizację nowej zmiany i jednocześnie odnosi ją do istniejącej racjonalizacji. Tę ostatnią przywołuje jako cytat z protokołu dokumentującego poprzednie zebrania oraz postu na grupie dyskusyjnej zespołu architektonicznego. Wspomniany protokół opublikowany zostanie na rzeczonej grupie dyskusyjnej, by programiści nieobecni na zebraniu mogli przedyskutować jego treść, zamykając w ten sposób cykl rejestrowania i wyjaśniania informacji składającej się na racjonalizację. Gdy używane są narzędzia hipertekstowe, odniesienie do istniejącej racjonalizacji ma formę hiperłącza, co umożliwia wygodne nawigowanie po powiązanych egzemplarzach informacji.

Nawet jeśli do rejestrowania i śledzenia poszczególnych zagadnień wykorzystywana jest zaawansowana baza danych, gromadzona w niej informacja może szybko przerodzić się w ogromny chaos, bez wyraźnej struktury. Co więcej, nie wszystkie zagadnienia są w tej bazie rejestrowane, bo nie wszystkie są przedmiotem dyskusji na zebraniach: wiele z nich rozstrzyganych jest w ramach nieformalnych konwersacji lub nawet przez pojedynczych programistów, bez jakichkolwiek konsultacji. Konieczne zatem staje się rekonstruowanie umykającej w ten sposób informacji i integrowanie jej z bieżącą racjonalizacją.

12.4.5. Rekonstruowanie racjonalizacji

Rekonstruowanie racjonalizacji to odmienna metoda jej kolekcjonowania. Zamiast rejestrowania na bieżąco, informacja składająca się na racjonalizację jest systematycznie rekonstruowana na podstawie modeli systemu, zapisków komunikacyjnych (e-maili, postów na grupach dyskusyjnych) i faktów zapamiętanych jedynie przez programistów. W ten sposób racjonalizacja rozwijana jest w sposób bardziej systematyczny, we wczesnych fazach projektu zużywanych jest mniej zasobów, co umożliwia programistom szybsze uzyskiwanie rozwiązań.

Ponadto odseparowanie aktywności projektowych od kolekcjonowania racjonalizacji umożliwia programistom krytyczne i bardziej obiektywne przyglądanie się efektom swych prac. Rekonstruowanie racjonalizacji opiera się jednak głównie na tych propozycjach, które wybrane zostały jako rozstrzygnięcia; propozycje odrzucone są szybko zapominane przez programistów i ich utrwalanie jest znacznie trudniejsze.

Załóżmy na przykład, że nie prowadziliśmy kolekcjonowania racjonalizacji w związku z tworzeniem systemu CTC i jedyna informacja związana z tym systemem zawarta jest w jego projekcie, którego fragment dotyczący kontroli dostępu prezentuje się następująco.

4. Kontrola dostępu

Kontrola dostępu w systemie CTC realizowana jest na poziomie sekcji dyspozycyjnych (`TrackSection`): dyspozytor (`Dispatcher`) przydzielony do danej sekcji może modyfikować ich stan, czyli przedstawiać sygnały i zwrotnice oraz operować innymi urządzeniami. Dyspozytor może również oglądać stan sekcji przylegających do jego sekcji macierzystej, bez możliwości modyfikowania ich stanu. Dyspozytor może w ten sposób obserwować pociągi zbliżające się do kontrolowanej przez niego sekcji.

Kontrola dostępu zaimplementowana jest w postaci list uprawnień, utrzymywanych przez poszczególne obiekty `TrackSection` reprezentujące konkretne sekcje. Dla danej sekcji lista ta zawiera identyfikację dyspozytora uprawnionego do modyfikowania tej sekcji („pisarza”) oraz identyfikację dyspozytorów uprawnionych do obserwowania ich stanu („czytników”)⁶. Ze względów ogólności (rozszerzalności) lista zaimplementowana jest w ten sposób, że może zawierać wielu „czytników” i wielu „pisarzy”. Lista utrzymywana przez daną sekcję sprawdzana jest przed wykonaniem każdej operacji odczytującej lub modyfikującej jej stan.

Gdy podsystem żąda subskrybowania określonej informacji, dotyczącej stanu danej sekcji, wywołwana przez niego usługa `NotificationService` kontaktuje się z tą sekcją w celu zweryfikowania praw wspomnianego podsystemu do odczytywania jej stanu. Gdy modyfikuje się listę uprawnień związaną z daną sekcją, sekcja ta wywołuje usługę `NotificationService` w celu anulowania nieuprawnionych subskrypcji.

Powyższe rozwiązanie przedstawione jest na rysunku 12.14.

Rekonstruując racjonalizację na podstawie dokumentacji i przeglądów, zapisujemy każde zagadnienie w formie tabeli z dwiema kolumnami, zawierającymi propozycje (lewa kolumna) i argumenty „za” i „przeciw” tym propozycjom (prawa kolumna). W tabeli 12.5 widoczny jest efekt takiej rekonstrukcji w odniesieniu do zintegrowania kontroli dostępu z mechanizmami powiadamiania. Identyfikujemy dwa możliwe rozwiązania: P[1], zgodnie z którym klasa `TrackSection` eksportuje wszystkie operacje wymagające kontroli dostępu, oraz P[2], polegające na delegowaniu przez usługę `NotificationService` kontroli dostępu do obiektu `TrackSection`. Wyliczamy zalety i wady każdego z tych rozwiązań (w prawej kolumnie), a na dole tabeli umieszczamy uzasadnienie decyzji o wyborze konkretnego rozwiązania.

⁶ To analogia do paradygmatu „czytników i pisarzy”, o tyle jednak daleka, że nie mamy tu do czynienia z rywalizowaniem o dostęp do zasobów — *przyp. tłum.*

Tabela 12.5. Zrekonstruowana racjonalizacja kontroli dostępu związanego z powiadamianiem w systemie CTC

I[1]: Jak powinno się zintegrować kontrolę dostępu z klasą <code>TrackSection</code> i usługą powiadamiania? Kontrola dostępu w systemie CTC realizowana jest na poziomie sekcji dyspozycyjnych (<code>TrackSection</code>): dyspozytor (<code>Dispatcher</code>) przydzielony do danej sekcji może modyfikować ich stan, czyli przestawiać sygnały i zwrotnice oraz operować innymi urządzeniami. Dyspozytor może również oglądać stan sekcji przylegających do jego sekcji macierzystej, bez możliwości modyfikowania ich stanu. Dyspozytor może w ten sposób obserwować pociągi zbliżające się do kontrolowanej przez niego sekcji.	
P[1]: Klasa <code>TrackSection</code> odpowiedzialna jest za modyfikowanie stanu sekcji oraz subskrypcje związane z powiadamianiem. Kontrola dostępu zaimplementowana jest w formie listy uprawnień utrzymywanej przez obiekty klasy <code>TrackSection</code>. Uprawnienia sprawdzane są dla każdej operacji odczytującej lub modyfikującej stan wspomnianych obiektów. Podsystem zamierzający subskrybować powiadomienia o zdarzeniach wywołuje odpowiednie metody klasy <code>TrackSection</code>, które z kolei przekazują wywołanie do usługi powiadamiania (<code>NotificationService</code>) pod warunkiem pozytywnego zweryfikowania uprawnień do subskrypcji. Rozwiązanie to przedstawione jest na rysunku 12.15.	Na rzecz propozycji: • Scentralizowane rozwiązanie — wszystkie chronione metody związane z klasą <code>TrackSection</code> zgrupowane są w tej klasie.
P[2]: Klasa <code>TrackSection</code> odpowiedzialna jest wyłącznie za modyfikowanie stanu sekcji, usługa powiadamiania (<code>NotificationService</code>) zarządza tylko powiadamianiem. Podobnie jak P[1], z tą jednak różnicą, że podsystem żąda subskrypcji bezpośrednio od usługi powiadamiania, która odwołuje się do obiektu <code>TrackSection</code> w celu zweryfikowania uprawnień do tej subskrypcji.	Na rzecz propozycji: • Interfejs niezależny od kontroli dostępu: interfejsy klas <code>NotificationService</code> i <code>TrackSection</code> są takie same jak w przypadku nieobecności kontroli dostępu. • Przeciwno propozycji: • Cykliczna zależność między klasami <code>NotificationService</code> i <code>TrackSection</code>: obiekt <code>TrackSection</code> wywołuje usługę powiadamiającą w celu zasygnalizowania zdarzenia, zaś usługa powiadamiająca odwołuje się do obiektu <code>TrackSection</code> w celu zweryfikowania uprawnień do subskrypcji.
R[1]: P[2]. Propozycja P[1] mogłaby być lepszym rozwiązaniem, ze względu na wyłączenie usługi powiadamiania z kontekstu kontroli dostępu, gdyby nie konieczność związanej z tym znaczącej przeróbki istniejącego kodu. W przypadku propozycji P[2] przeróbka ta jest znacznie mniejsza.	

Przedstawione rekonstruowanie racjonalizacji jest mniej kosztowne niż aktywności uprzednio opisywane. Jest jednak znacznie trudniejsze pod względem dokumentowania odrzuconych propozycji i powodów ich odrzucenia, szczególnie gdy podejmowane decyzje były wielokrotnie rewidowane. Rozstrzygnięciu z tabeli 12.5 towarzyszy wyjaśnienie, iż jest ono

wyborem gorszej (nieoptymalnej) propozycji z określonych względów praktycznych (czyli przeróbki kodu znacznie bardziej ograniczonej w porównaniu z wyborem propozycji optymalnej). O takich subtelnościach często jednak zapomina się z czasem.

Rekonstruowanie racjonalizacji może być efektywnym narzędziem przeglądu systemu, szczególnie pod względem wyszukiwania decyzji niespójnych z celami projektowymi. I jeżeli nawet zmiana tych decyzji okazuje się niewykonalna w późnych stadiach projektu, nabyta przy tej okazji wiedza okaże się pożyteczna dla nowych programistów oraz programistów dokonujących rewizji systemu w następnych iteracjach.

Wzajemne proporcje między kolekcjonowaniem, utrzymywaniem i rekonstruowaniem racjonalizacji są różne dla odmiennych projektów i muszą zostać starannie określone przez menedżera projektu. Spotyka się dość często kolekcjonowanie, przy znacznym wysiłku, potężnej dawki informacji w większości bezużytecznej lub trudno dostępnej dla programistów, którzy mogliby ją spożytkować. Zajmiemy się tym problemem w następnej sekcji.

12.5. Kierownicze aspekty zarządzania racjonalizacją

Gdy menedżer projektu żąda od programistów szczegółowego uzasadnienia podejmowanych przez nich decyzji projektowych, często traktowany jest jak intruz, a same techniki związane z racjonalizacją napotykają opór ze strony programistów i rychło przeradzają się w biurokrację. Stanowi to duże wyzwanie dla menedżera, odpowiedzialnego za najważniejsze aspekty zarządzania racjonalizacją, między innymi:

- jej dokumentowanie (patrz sekcja 12.5.1),
- przypisywanie odpowiedzialności w zakresie kolekcjonowania i pielęgnowania modeli racjonalizacji (patrz sekcja 12.5.2),
- zapewnienie komunikacji dotyczącej modeli racjonalizacji (patrz sekcja 12.5.3),
- negocjowanie zagadnień (patrz sekcja 12.5.4),
- rozwiązywanie konfliktów (patrz sekcja 12.5.5).

Jak poprzednio, koncentrujemy się na racjonalizacji związanej z etapem projektowania systemu, opisywane techniki przydatne są jednak równie dobrze na wszystkich pozostałych etapach.

12.5.1. Dokumentowanie racjonalizacji

Modele racjonalizacji (na przykład modele zagadnień) różnią się pod względem struktury od modeli systemu (modelu przypadków użycia, modelu klas, kodu źródłowego). Jako że modele typowego systemu są rozległe i skomplikowane, programiści wykorzystują rozmaite techniki radzenia sobie z ich złożonością, organizując je w warstwy i hierarchie, wiążąc je z przypadkami użycia, czy też kojarząc dokumentację obiektów systemu z ich kodem źródłowym. Modele racjonalizacji są z natury bardziej objętościowe od modeli systemu, obejmują bowiem wszystkie propozycje związane z rozwiązywaniem poszczególnych problemów, argumenty „za” i „przeciw” tym propozycjom, podjęte decyzje oraz ich uzasadnienia. Ponieważ podejmowane decyzje bywają rewidowane i zmieniane, modele racjonalizacji także nieustannie ewoluują. Niewłaściwe jest zatem myślenie o racjonalizacji w kategoriach jedynie dokumentu,

gdyż każdy taki dokument szybko stawałby się nieaktualny i nieadekwatny do pozostałych dokumentów. Odpowiedniejszym podejściem jest utrzymywanie racjonalizacji w formie repozytorium, nieustannie aktualizowanego i uzupełnianego o nowe zagadnienia i decyzje. Zawartość tego repozytorium musi być jednak utrzymywana w aktualnym stanie i łatwo dostępna dla wszystkich zainteresowanych. Wymaganie to prowadzi wprost do ścisłego zintegrowania repozytorium racjonalizacji z innymi narzędziami i procesami programistycznymi. Przykładem narzędzia realizującego taką integrację jest *REQuest*; narzędzie to umożliwia formułowanie przypadków użycia w kontekście ich racjonalizacji, o czym piszą A. H. Dutoit i B. Paech [Dutoit i Paech, 2002]. Na rysunku 12.18 widzimy przykładowy ekran tej aplikacji; lewa kolumna zapewnia dostęp do poszczególnych wymagań systemu, w podziale na przypadki użycia i wymagania pozafunkcyjne, zaś w prawej kolumnie prezentowane są elementy racjonalizacji związane z poszczególnymi wymaganiami, w kategoriach rozszerzonego modelu QOC. Przyciski w górnej części lewej kolumny umożliwiają programistom formułowanie pytań w różnych kategoriach (klarowności, kompletności, spójności, prawidłowości uformowania, poprawności, uzasadnienia) i kojarzenie ich z aktualnie wyświetlanym elementem.

Zielone, żółte i czerwone wskaźniki towarzyszące elementom racjonalizacji wymienionym w lewej kolumnie oznaczają status zagadnienia reprezentowanego przez ten element. Klikając wspomniany wskaźnik, otrzymujemy opis odnośnego zagadnienia w prawej kolumnie.

The screenshot shows the REQuest application window titled "REQuest: MeetingScheduler System". The interface is divided into two main panes. The left pane, titled "MeetingScheduler Specification", displays the system specification, including a path, actions, questions, and a detailed section for "3.1. Use Cases". The right pane, titled "MeetingScheduler Questions For Handle Replies By Date", shows a table of questions with columns for Type, Subject, Status, Author, and Date.

MeetingScheduler Specification

Path: System > Specification
 Actions: [Refresh] [All] [Index] [Printable Version] [Help]
 Questions: [Unclear?] [Incomplete?] [Inconsistent?] [Ill structured?] [Incorrect?]

3. Specification [Edit] [Attach Diagram] [Help]
 This section describes the specification of the system in terms of use cases, services, and quality constraints. Use cases describe sequence of interactions between actors and the system. Services are features that the system provides that can be used to realize use cases. Quality constraints are constraints on use cases or services that must be met by the system.
 Created on 7/25/01 12:49 PM by allen.

3.1. Use Cases [New Use Case] [Help]
 User task Manage Interaction Among Participants is realized by the use cases:

Use Case	Author	Date	Action
Cancel Meeting	lieblb	1/24/01 1:06 PM	[Delete]
Handle Replies	dutoit	7/25/01 12:57 PM	[Delete]
React to Replan Request	wildmosm	1/24/01 12:55 PM	[Delete]
Remind Participant	lieblb	1/24/01 1:11 PM	[Delete]
Set Meeting	wildmosm	1/24/01 11:16 AM	[Delete]

User task Plan Meeting is realized by the use cases:

MeetingScheduler Questions For Handle Replies By Date

[Show All Questions] [Printable Version]

Type	Subject	Status	Author	Date
Question	Relationship with Request Meeting and Plan Meeting	Resolved	dutoit	1/24/01 10:15 AM
Question	is conflict resolution part of this use case	Resolved	paech	1/23/01 10:42 AM
Question	Handle Replies	Open	novak	1/23/01 9:03 AM
Question	Connection failure	Open	novak	1/23/01 8:43 AM

Rysunek 12.18. *REQuest* — przykład repozytorium racjonalizacji zintegrowanego z narzędziem inżynierii wymagań: elementy modelu racjonalizacji kojarzone są z poszczególnymi wymaganiami. W lewej kolumnie wyświetlane są elementy modelu wymagań, prawa kolumna dedykowana jest modelowi zagadnień

Zintegrowanie elementów specyfikacji z widokami elementów racjonalizacji daje programistom łatwy dostęp do aktualnej informacji; zapewnienie jej aktualności w dłuższej perspektywie, poprzez konsolidowanie i pielęgnowanie jej elementów, jest zadaniem wynikającym z roli *redaktora racjonalizacji*; tę i inne role związane z zarządzaniem racjonalizacją opiszemy w następnej sekcji.

12.5.2. Przypisywanie odpowiedzialności

Właściwe przyporządkowanie odpowiedzialności związanej z kolekcjonowaniem i pielęgnowaniem racjonalizacji jest najbardziej krytyczną decyzją menedżera, decydującą o rzeczywistej użyteczności modeli racjonalizacji. Zarządzanie racjonalizacją sprowadzone do postaci biurokratycznej procedury uzasadniania przez każdego programistę każdej podejmowanej przez niego decyzji nie spełni swego zadania. Zamiast tego modele racjonalizacji powinny być utrzymywane przez niewielką grupę osób — „historyków systemu” — na podstawie wszelkiej informacji, jaka będzie dla nich dostępna ze strony programistów: szkiców dokumentów projektowych, postów na grupach dyskusyjnych i tym podobnych. Ponieważ gromadzona w ten sposób informacja jest jedną z najważniejszych, jakiej potrzebują programiści w zmaganiach się z konsekwencjami nieuchronnych zmian, w interesie samych programistów leży troska o zapewnienie jej należytej szczegółowości i kompletności.

W związku ze wspomnianą na wstępie odpowiedzialnością za aktualność i użyteczność informacji składającej się na racjonalizację, menedżer projektu wyznacza następujące role.

- **Protokolant** odpowiedzialny jest za kolekcjonowanie na bieżąco elementów racjonalizacji pojawiających się w ramach zebrań; kolekcjonowanie to polega na sporządzaniu chronologicznego zapisu dyskusji, który to zapis po zakończeniu zebrania stanowi podstawę do sporządzenia strukturalnego protokołu dokumentującego zagadnienia, propozycje, argumenty i rozstrzygnięcia. Pisaliśmy o tym w sekcji 12.4.2.
- **Redaktor racjonalizacji** odpowiedzialny jest za gromadzenie i organizację informacji związanej z racjonalizacją. Informacja ta obejmuje między innymi strukturalne protokoły z zebrań, prototypy, oceny poszczególnych technologii przez programistów, szkice modeli systemu oraz dokumenty projektowe (lub ich szkice). Dla zebranej informacji tworzony jest indeks według poszczególnych zagadnień. Redaktor racjonalizacji wyręcza programistów w porządkowaniu i strukturalizowaniu informacji składającej się na racjonalizację, ograniczając ich rolę do udostępniania tej informacji.
- **Weryfikator racjonalizacji** analizuje informację zebraną przez redaktora i identyfikuje napotkane w niej luki, korzystając z tych samych źródeł informacji, co redaktor. Nie jest to rola o charakterze menedżerskim, weryfikator powinien przekonać programistów, iż uzyskanie od nich rzetelnych informacji opłaci się w dłuższej perspektywie. Role redaktora i weryfikatora często powierzane są tej samej osobie.

Rozmiar projektu determinuje liczbę protokolantów, redaktorów i weryfikatorów. Przy przydzielaniu tych ról pomocne mogą okazać się następujące heurystyki.

- *Jeden protokolant dla każdego zespołu.* Zebrania są zwykle organizowane przez zespoły zajmujące się poszczególnymi podsystemami lub zespół międzyfunkcyjny. Rolę protokolanta mogą rotacyjnie pełnić poszczególni programiści, co przyczynia się do równomiernego rozłożenia tego czasochłonnego obowiązku w całym cyklu realizacji projektu.
- *Jeden i ten sam redaktor racjonalizacji przez cały czas realizacji projektu.* Ta rola wymaga pełnoetatowego zaangażowania, w przeciwieństwie więc do rotacyjnej roli protokolanta wymaga zachowania spójności i z tego względu powinna być przypisana pojedynczej osobie. W przypadku małych projektów rolę tę pełnić może architekt systemu (patrz rozdział 6. „Projektowanie systemu — dekompozycja na podsystemy”).

- *Zwiększenie liczby weryfikatorów racjonalizacji po dostarczeniu systemu.* Po dostarczeniu systemu klientowi zmniejsza się liczba programistów, którzy muszą być bezpośrednio zaangażowani w projekt. Programistom odciążonym można powierzyć role weryfikatorów, którzy powinni wydobyć tyle informacji związanej z racjonalizacją, ile to tylko możliwe — elementy tej informacji wciąż są jeszcze żywe w umysłach programistów i należy je utrwalić, zanim naturalną kolejną rzeczycę odejdą w zapomnienie.

12.5.3. Heurystyki komunikowania racjonalizacji

Duża część informacji komunikowanej w związku z projektem to informacja związana z racjonalizacją, każdy argument jest bowiem z definicji racjonalny, co wyjaśnialiśmy w sekcji 12.2. Programiści wymieniają argumenty dotyczące celów projektowych, istotności związanych z nimi zagadnień, ich ewaluacji i tak dalej. Na racjonalizację systemu składa się ogrom skomplikowanej informacji, znacznie obszerniejszej niż sam system. Informacja ta wymieniana jest najczęściej w kręgu małych forów, między innymi na zebraniach zespołu i w ramach nieformalnych konwersacji przy automacie z kawą. Podstawowym wyzwaniem pod adresem zarządzania racjonalizacją jest udostępnienie tej informacji zainteresowanym uczestnikom, z zachowaniem treści, lecz bez przeładowywania formy. Opisaliśmy już kilka technik kolekcjonowania i strukturalizowania racjonalizacji, między innymi przez modelowanie zagadnień w protokołach i reprezentowanie racjonalizacji w formie repozytoriów. Uzupełnieniem tych technik mogą być poniższe heurystyki, które sprawiają, że struktura racjonalizacji staje się bardziej czytelna i łatwiej po niej nawigować.

- *Wybieraj nazwy zagadnień w sposób spójny.* Zagadnienia powinny być opatrywane spójnymi nazwami, unikalnymi w obrębie protokołów, grup dyskusyjnych, komunikatów e-mail i dokumentów. Dla ułatwienia identyfikacji w nazwie każdego zagadnienia powinien być obecny jego numer i krótką, sugestywną frazę określającą jego kategorię (na przykład „zagadnienie_dostęp/powiadamy”).
- *Scentralizuj zagadnienia.* Mimo iż zagadnienia mogą być dyskutowane w różnych kontekstach, jeden z nich (grupę dyskusyjną, bazę zagadnień) należy specjalnie wyróżnić jako ich centralne repozytorium. Baza zagadnień powinna być utrzymywana przez redaktora racjonalizacji, lecz każdy programista powinien mieć możliwość jej odczytywania i rozszerzania o nowe pozycje. Ułatwi to programistom szybkie wyszukiwanie niezbędnych informacji.
- *Zapewnij powiązania zagadnień z odnośnymi elementami systemu.* Większość zagadnień odnosi się do konkretnych elementów modeli systemu (przypadków użycia, obiektów, podsystemów). Identyfikowanie zagadnień związanych z konkretnym elementem systemu jest sprawą oczywistą, relacja odwrotna — identyfikowanie elementu systemu, do którego odnosi się konkretne zagadnienie — jest zdecydowanie bardziej problematyczna. By relację ta stała się bardziej użyteczna i czytelna, należy wiązać każde zagadnienie w momencie jego powstawania z elementem systemu, którego dotyczy.

- *Zarządzając zmianami, pamiętaj o racjonalizacji.* Racjonalizacja systemu ewoluuje wraz z jego modelami, zatem powinna podlegać zarządzaniu konfiguracją w takim samym stopniu jak wspomniane modele.

Kolekcjonowanie i strukturalizowanie racjonalizacji nie tylko usprawnia jej komunikowanie, lecz także ułatwia komunikację związaną z modelami systemu. Zintegrowanie obu tych kategorii informacji ułatwia programistom utrzymywanie spójności między nimi.

12.5.4. Modelowanie i negocjowanie zagadnień

Najważniejsze decyzje związane z projektem są w większości przypadków wynikiem negocjacji — różne strony, reprezentujące różne, często sprzeczne interesy wypracowują konsensus w sprawie wybranego aspektu systemu. Analiza wymagań wiąże się z negocjowaniem kształtu funkcjonalnego systemu z klientem, projektowanie systemu wymaga negocjowania interfejsów między programistami, integrowanie podsystemów wymaga rozwiązywania konfliktów między programistami. Modele zagadnień są wygodną formą reprezentowania informacji wymienianej w czasie negocjowania, mogą być jednak równie użyteczne w dziele ułatwiania samych negocjacji.

Tradycyjne negocjowanie, polegające na trwaniu z uporem na swych pozycjach, jest często czasochłonne i bezowocne, szczególnie gdy stanowiska negocjujących stron są wyraźnie sprzeczne. Cały wysiłek kierowany jest wówczas na akcentowanie zalet własnego stanowiska, przy jednoczesnym wytykaniu wad stanowiska prezentowanego przez drugą stronę. Jeżeli nawet obie strony zainteresowane są osiągnięciem porozumienia, perspektywa zmiany prezentowanego nastawienia postrzegana bywa jako ryzyko utraty wiarygodności. Takie negocjacje z trudem (jeżeli w ogóle) posuwają się do przodu, prowadząc do wypracowania *jakiegoś* kompromisu.

Jedną ze znanych metodologii pokonywania opisanych trudności jest tak zwana harwardzka metoda negocjowania, opisana w książce R. Fishera, W. Ury'ego i B. Pattona [Fisher i in., 1991], koncentrująca przebieg negocjacji na ich przedmiocie, a nie na negocjujących stronach. W przełożeniu na modelowanie zagadnień podstawowe założenia tej metody można sformułować następująco.

- *Oddzielaj propozycje od formułujących je programistów.* Programiści wkładają tak wiele wysiłku w wypracowywanie konkretnych propozycji i w efekcie tak silnie się z nimi identyfikują, że krytykowanie danej propozycji odbierane jest jak osobisty atak na jej autora. Wyraźne oddzielenie propozycji od osoby jej autora ułatwia dyskusowanie i być może odrzucenie. Można ten cel osiągnąć, wypracowując poszczególne propozycje kolektywnie (nie jednoosobowo) lub angażując obie negocjujące strony w ich opracowywanie. Programiści łatwiej rezygnują ze swych propozycji, jeśli te nie wiążą się jeszcze z zaangażowaniem znacznych zasobów, stąd oczywisty wniosek, by negocjowanie propozycji odbywało się jeszcze przed rozpoczęciem implementowania ich następstw.
- *Koncentruj się na kryteriach, nie propozycjach.* Programiści bronią swych propozycji (i podważają propozycje drugiej strony), działając zgodnie z pewnymi kryteriami — wysuwane przez nich propozycje są zapewne zgodne z ich własnymi kryteriami,

lecz niekoniecznie z kryteriami przyjmowanymi przez innych programistów. Konflikt w negocjacjach jest więc najczęściej tak naprawdę konfliktem między kryteriami — gdy kryteria staną się w pełni jawne, łatwiejsze będzie wypracowanie kompromisu. Gdy uzgodniony zostanie zestaw kryteriów satysfakcjonujący wszystkie negocjujące strony, negocjowanie poszczególnych propozycji nabierze bardziej obiektywnego charakteru i stanie się daleko mniej kontrowersyjne.

- *Uwzględniaj wszystkie kryteria, zamiast faworyzować jedno wybrane.* Różnica kryteriów przyjętych przez poszczególne strony negocjacji wynika wprost z różnicy interesów tych stron: kryteria wydajnościowe motywowane są przez względy użyteczności systemu, kryteria modyfikowalności — przez względy jego pielęgnacji i tak dalej. Nawet jeśli pewne kryteria wyraźnie górują nad innymi pod względem priorytetu, zupełne ignorowanie tych o niższym prioryecie może równać się ignorowaniu interesów jednej lub kilku negocjujących stron.

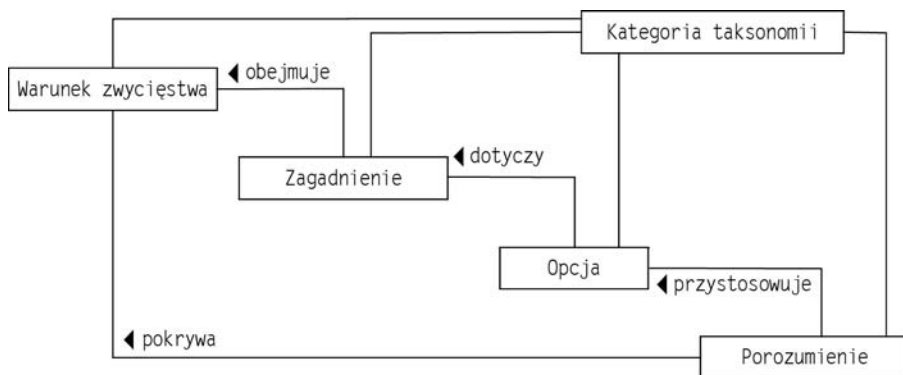
Postrzeganie realizacji projektu w kategoriach negocjacji uwypatnia jego aspekt społeczny. Programiści są przecież ludźmi, a zatem — poza oczywistym nastawieniem technicznym — prezentują określony stosunek emocjonalny do poszczególnych ewentualności rozwiązania określonego problemu. Przekłada się to w prostej linii na interpersonalne relacje między programistami i stanowić może genezę rozmaitych konfliktów natury osobistej. Sprowadzenie zagadnienia do obiektywnego, czysto technicznego wymiaru — co osiągnąć można właśnie dzięki ich modelowaniu — zdecydowanie sprzyja łagodzeniu opisanego zjawiska.

Przykładem tak ukierunkowanego modelu jest *WinWin* — narzędzie wspomagające zbieranie wymagań i ułatwiające negocjowanie stron prezentujących odmienne punkty widzenia, które w swej pracy opisali B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah i R. Madachy [Boehm i in., 1998]. U podstaw jego powstania legła znana prawda, iż warunkiem koniecznym pomyślności w realizacji projektu jest zadowolenie jego głównych uczestników. Często bowiem postęp w negocjacjach uwarunkowany jest nie tyle rozstrzygnięciem kompromisów między negocjującymi stronami, ile właśnie rozpoznaniem kryteriów przyjmowanych przez negocjujące strony. Jawność wszystkich wspomnianych kryteriów ułatwia rozpoznawanie konfliktów między nimi i poszukiwanie kompromisowych rozwiązań.

WinWin wykorzystuje model podobny do QOC (patrz rysunek 12.19), od którego różni się uporządkowaniem węzłów w hierarchie taksonomii. Oryginalne węzły kryteriów modelu QOC tutaj nazywane są „warunkami zwycięstwa” (*Win Condition*) i reprezentują kryteria sukcesu poszczególnych stron negocjacji. Zgodnie z modelem, proces rozpoczyna się od zidentyfikowania głównych „graczy” i ich „warunków zwycięstwa”; skonfliktowane ze sobą „warunki zwycięstwa” modelowane są w postaci zagadnień. Gdy osiągnięte zostanie porozumienie (*Agreement*), jest ono wprowadzane do bazy *WinWin* i kojarzone ze wspomnianymi warunkami. Ułatwia to wypracowywanie i dokumentowanie osiągniętych kompromisów.

12.5.5. Strategie rozwiązywania konfliktów

Niestety, czasami zdarza się tak, że wypracowanie kompromisu w negocjacjach okazuje się nieosiągalne. W takiej sytuacji trzeba sięgnąć po określone strategie rozwiązywania konfliktów. Najgorszymi bowiem decyzjami projektowymi są te, które nie zostały podjęte wskutek



Rysunek 12.19. Model zagadnień systemu WinWin

braku porozumienia i braku wspomnianych strategii. Odkładanie istotnych decyzji na późniejsze fazy projektu może wiązać się z kolosalnymi nawet kosztami zmian w projekcie i kolekcjonowania kryjącej się za nimi racjonalizacji.

Spośród wielu istniejących strategii rozwiązywania konfliktów ograniczymy się do pięciu poniższych.

- **Większość ma rację.** Głosowanie na zasadzie „decyduje większość” może przełamać impas i przyczynić się do podjęcia decyzji. Różne narzędzia wspomagające współpracę grupową umożliwiają przypisanie wag istotności poszczególnym argumentom w modelu zagadnień i wyłonienie „zwycięskiej” propozycji na podstawie formuły arytmetycznej, o czym można przeczytać w pracy M. Purvisa, M. Purvis i P. Jonesa [Purvis i in., 1996]. Zakłada się przy tym równouprawnienie wszystkich dyskutantów i statystyczne oczekiwanie, że grupa zwykle podejmuje właściwe decyzje.
- **Właściciel ma ostatnie słowo.** Zgodnie z tą strategią rozstrzygający głos należy do „właściciela” zagadnienia, czyli osoby, która je sformułowała.
- **Menedżer ma zawsze rację.** Ta strategia jest szczególnie przydatna w warunkach hierarchicznej organizacji projektu: gdy grupa nie potrafi wypracować konsensusu, jej menedżer narzuca swą decyzję, wypracowaną na podstawie argumentów prezentowanych przez członków tej grupy. Zakłada się tu, że menedżer jest w stanie należycie zrozumieć wspomniane argumenty.
- **Ekspert ma zawsze rację.** W ramach tej strategii rozstrzygnięcia dokonuje niezależny ekspert, niezaangażowany w debatę, na podstawie oceny sytuacji i swego doświadczenia. Przykładowo na etapie analizy wymagań ekspertem takim może być testujący użytkownik, któremu prezentuje się poszczególne propozycje rozwiązania zagadnienia. Wadą tej strategii jest fakt, że ów ekspert może mieć nikłą wiedzę na temat innych zagadnień i generalnie na temat kontekstu diskutowanego zagadnienia.
- **Niech czas zdecyduje.** Gdy dane zagadnienie pozostaje nierozwiązane, upływający czas wywiera coraz większą presję na znalezienie rozstrzygnięcia i podjęcie decyzji. Wraz z upływem czasu mogą pojawić się nowe okoliczności — zdefiniowanie nowych

aspektów systemu czy podjęcie ważnych decyzji — które znacznie ułatwią rozwiązanie patowego dotąd problemu. Niebezpieczeństwem tej strategii jest zagrożenie, iż wymusza ona podejmowanie decyzji krótkowzrocznych, optymalizujących kryteria krótkoterminowe — na przykład łatwość implementacji — a ignoruje cele dalekosiężne, takie jak modyfikowalność i pielęgnowalność.

Strategie „Większość ma rację” i „Właściciel ma ostatnie słowo” są najgorszymi z możliwych, bowiem oznaczają ignorowanie racji sporej (być może) liczby uczestników. Strategie „Menedżer ma zawsze rację” i „Ekspert ma zawsze rację” prowadzą do lepszego rezultatu, pod warunkiem wszakże, iż menedżer (ekspert) ma wystarczającą wiedzę na temat problemu będącego przedmiotem sporu. Strategia „Niech czas zdecyduje” oznacza tymczasową ucieczkę od problemu, na dłuższą metę niosącą ryzyko kosztownych przeróbek.

W praktyce najpierw należy dążyć do osiągnięcia konsensusu. Gdy dążenia okazują się bezowocne, należy pozostawić decyzję menedżerowi lub ekspertowi. Gdy ci nie potrafią do końca właściwego wyboru, pozostaje głosowanie jako ostateczność.

12.6. Literatura uzupełniająca

Większość prac związanych z modelowaniem zagadnień ma swe źródło w publikacji W. Kunza i H. Rittela [Kunz i Rittel, 1970], choć opisuje ona wykorzystywanie modelu IBIS w kontekście negocjowania złożonych problemów politycznych. Zastosowanie modelu IBIS na gruncie inżynierii oprogramowania nastąpiło w późnych latach 80. ubiegłego stulecia, w postaci narzędzia *gIBIS*, omówionego w pracy J. Conklina i K. C. Burgessa-Yakemovica [Conklin i Burgess-Yakemovic, 1991], a wykorzystywanego z powodzeniem w przemyśle do wielu analiz przypadków. Narzędzie *gIBIS* stało się też punktem wyjścia dla innego (komercyjnego) narzędzia *QuestMap*, wykorzystywanego do kolekcjonowania i strukturalizowania racjonalizacji w czasie zebrań.

Na początku lat 90. ubiegłego wieku alternatywą IBIS dla stał się QOC, koncentrujący się na systematycznym wartościowaniu propozycji według przyjętych kryteriów (zamiast okazjonalnie formułowanych argumentów), o czym piszą A. MacLean, R. M. Young, V. Bellotti i T. Moran [MacLean i in., 1991]. Oba modele podzieliły odtąd rynek i IBIS wykorzystywany jest w kontekście racjonalizacji gromadzonej „w locie”, podczas gdy QOC jest bardziej przydatny do długofalowego reprezentowania informacji związanej z racjonalizacją.

W literaturze przedmiotu spotyka się niewiele przykładów zastosowania racjonalizacji w tworzeniu oprogramowania — do nielicznych należą książki T. P. Morana i J. M. Carrolla [Moran i Carroll, 1996] oraz A. H. Dutoita, R. McCalla, I. Mistrika i B. Peacha [Dutoit i in., 2006]. Efektywne wykorzystywanie racjonalizacji w procesie tworzenia oprogramowania jest trudnym zadaniem ze względów zarówno technicznych (modele racjonalizacji są obszerne, trudne w zarządzaniu i wykorzystywaniu), jak i pozatechnicznych — kolekcjonowanie racjonalizacji wiąże się z wysiłkiem, którego beneficjentami stają się głównie inne osoby, o czym piszą A. H. Dutoit i B. Paech [Dutoit i Paech, 2001]. Istnieją jednak liczne fakty, ilustrujące wyjątkowe znaczenie gromadzenia i integrowania racjonalizacji dla powodzenia rozmaitych przedsięwzięć. W tym rozdziale ograniczyliśmy się do trzech narzędzi wspierających stosowanie racjonalizacji w inżynierii oprogramowania: *WinWin* [Boehm i in., 1998], *NFR Framework* [Chung i in., 1999] i *REQuest* [Dutoit i Paech, 2002].

12.7. Ćwiczenia

- 12.1. W sekcji 12.3 omawialiśmy zagadnienie związane z kontrolą dostępu i powiadamianiem w systemie CTC. Podaj przykład innego zagadnienia, jakie może wyniknąć w związku z tworzeniem takiego systemu, przedstaw możliwe propozycje, kryteria i argumenty. Zaproponuj oraz uzasadnij rozstrzygnięcie. Oto dwa możliwe przykłady wspomnianych zagadnień:
- Jak zapewnić spójność między serwerem głównym (mainServer) a zapasowym (hotBackup)?
 - Jak wykrywane będą awarie serwera głównego i jak zaimplementowane będzie przejęcie pracy przez serwer zapasowy?
- 12.2. Wyobraź sobie, że tworzysz program narzędziowy wspomagający modelowanie w języku UML i postanowiłeś ściśle zintegrować zarządzanie racjonalizacją z innymi mechanizmami modelowania. Opisz, w jaki sposób programista wykorzystujący Twoje narzędzie kojarzył będzie zagadnienia z innymi elementami modelu. Narysuj diagram klas przedstawiający model zagadnień i wspomniane skojarzenia.
- 12.3. Poniższy fragment jest wyjątkiem dokumentu SDD dla systemu FRIEND opisującym w języku naturalnym racjonalizację decyzji wyboru relacyjnej bazy danych jako magazynu przechowywania obiektów trwałych. Narysuj model zagadnień obejmujący zagadnienia, propozycje, argumenty i kryteria zawarte w tym opisie (podobnie jak uczyniliście to w sekcji 12.3).

Jednym z fundamentalnych problemów przy projektowaniu podsystemu przechowywania danych jest wybór silnika bazodanowego. Początkowo jedno z wymagań pozafunkcyjnych nakazywało użycie w tej roli obiektowej bazy danych, jako alternatywę wymieniono natomiast relacyjną bazę danych, „płaskie” pliki oraz kombinację tych dwóch mechanizmów. Obiektowa baza danych daje wymierne korzyści w postaci automatycznego zarządzania skomplikowanymi powiązaniami między przechowywanymi obiektami, jest jednak mało wydajna w stosunku do dużych rozmiarów danych i częstych operacji wyszukiwania, poza tym dostępne obecnie produkty tej kategorii nie integrują się dobrze z mechanizmem CORBA, ze względu na brak wsparcia cech specyficznych dla niektórych języków programowania, między innymi skojarzeń w języku Java. Relacyjne bazy danych oferują rozwiązanie bardziej niezawodne, bardziej wydajne, a także większy wybór produktów i większe wsparcie w postaci licznych narzędzi. Ponadto relacyjne bazy danych integrują się bezproblemowo z mechanizmem CORBA. Relacyjne bazy danych nie zapewniają jednak bezpośredniego wsparcia dla implementacji skomplikowanych powiązań między danymi. Zaproponowano także specjalne potraktowanie danych tworzonych jednorazowo i odczytywanych bardzo rzadko, lecz koniecznych do przechowywania przez dłuższy czas — na przykład odczytów wskazań czujników, nagrywanych rozmów czy migawek z kamer: ze względu na skąpe powiązania tych danych z innymi elementami najbardziej odpowiednim mechanizmem ich przechowywania byłyby niezależne pliki, zdolne przechowywać niejednorodne dane dużych rozmiarów i łatwo poddające się archiwizowaniu. System niezależnych plików wymagałby jednak napisania „od zera” wszelkiego kodu związanego z zarządzaniem ich zawartością, w tym kodu odpowiedzialnego za serializację danych i szeregowanie dostępu. Ostatecznie wybraliśmy relacyjną bazę danych, bazując na wymaganiach związanych z mechanizmem CORBA i stosunkowo prostym charakterem powiązań między elementami przechowywanych danych.

- 12.4. Narysuj model QOC równoważny grafowi celów frameworku NFR z rysunku 12.12. Przedyskutuj wady i zalety zastosowania QOC i NFR dla reprezentowania racjonalizacji w procesie zbierania i analizowania wymagań.
- 12.5. Wyobraź sobie, że integrujesz system raportowania błędów z narzędziem zarządzania konfiguracją w celu śledzenia wykrywanych błędów, ich poprawiania, żądania nowych cech i rozszerzeń. Integrację tę chcesz zrealizować za pomocą modelu zagadnień. Narysuj diagram klas tego modelu, uwzględniający elementy zarządzania konfiguracją, elementy raportowania błędów i dyskusje związane z tymi elementami.

Bibliografia

- [Boehm i in., 1998] B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, R. Madachy *Using the WinWin spiral model: A case study*, „IEEE Computer” 31(7): str. 33 – 44, 1998.
- [Chung i in., 1999] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos *Non-Functional Requirements in Software Engineering*, Kluwer Academic, Boston, 1999.
- [Conklin i Burgess-Yakemovic, 1991] J. Conklin, K. C. Burgess-Yakemovic *A process-oriented approach to design rationale*, „Human-Computer Interaction”, t.6, str. 357 – 391, 1991.
- [Dutoit i Paech, 2001] A.H. Dutoit, B. Paech „Rationale management in software engineering” w: S. K. Chang (red.) *Handbook of Software Engineering and Knowledge Engineering*, t.1, World Scientific Publishing, 2001.
- [Dutoit i Paech, 2002] A. H. Dutoit, B. Paech *Rationale-based use case specification*, „Requirements Engineering Journal”, 7(1): str. 3 – 19, 2002.
- [Dutoit i in., 2006] A. H. Dutoit, R. McCall, I. Mistrik, B. Paech (red.) *Rationale Management in Software Engineering*, Springer, Heidelberg, 2006.
- [Fisher i in., 1991] R. Fisher, W. Ury, B. Patton *Dochodząc do TAK. Negocjowanie bez poddawania się*, Polskie Wydawnictwo Ekonomiczne, Warszawa, 2000.
- [Kunz i Rittel, 1970] W. Kunz, H. Rittel „Issues as elements of information systems”, *Working Paper Nr 131*, Institut für Grundlagen der Planung, Universität Stuttgart, Germany, 1970.
- [Lee, 1990] J. Lee „A qualitative decision management system” w: P. H. Winston, S. Shellard (red.) *Artificial Intelligence at MIT: Expanding Frontiers*, t.1, str. 104 – 133, MIT Press, Cambridge, MA, 1990.
- [MacLean i in., 1991] A. MacLean, R. M. Young, V. Bellotti, T. Moran „Questions, options, and criteria: Elements of design space analysis”, „Human-Computer Interaction”, t.6, str. 201 – 250, 1991.
- [Moran i Carroll, 1996] T. P. Moran, J. M. Carroll (red.) *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[Purvis i in., 1996]

M. Purvis, M. Purvis, P. Jones „A group collaboration tool for software engineering projects”, *Conference proceedings of Software Engineering: Education and Practice (SEEP'96)*, Dunedin, NZ, styczeń 1996.

Skorowidz

@see, 429
«boundary», 106
«control», 106, 216
«create», 224
«entity», 106
«extent», 106
«include», 106
«invariant», 408
«postcondition», 408
«precondition», 408
«sut», 540
«testCase», 540
«testComponent», 540
«testContext», 540
«testObjective», 540

A

Abstract Factory, 772
Abstract Window Toolkit, 276
abstrakcyjne typy danych, 72
action items, 561, 646
activities, 640, 646
activity-centered, 685
Ada, 606
adaptacyjne tworzenie oprogramowania, 737
Adapter, 365, 366, 371, 773
 delegowanie, 373
 dziedziczenie, 373
Adaptive Software Development, 737
adaptowalność, 162
agenda przeglądu klienckiego, 131
agenda spotkania, 142
agile, 637, 673
Agile Alliance Manifesto, 737
agregacja, 92, 231
 agregacja współdzielona, 231
 kompozycyjna, 231
agregat CM, 602, 603
agregat zarządzania konfiguracją, 602
Airbus A320, 598
akceptacja systemu, 672
akcje, 99
aktorzy, 79, 80
aktywności, 43, 47, 68, 100, 124, 640, 689
 partycje aktywności, 103
aktywności analizy wymagań, 217
aktywności etapu projektowania systemu, 305
aktywności inżynierii oprogramowania, 49
 analiza, 51
 implementowanie, 53
 projekt systemu, 51
 projektowanie obiektów, 53
 testowanie, 54
 zbieranie wymagań, 50
aktywności klasycznego zarządzania projektem, 653
aktywności organizacyjne, 146
 dołączanie do infrastruktury komunikacyjnej, 146
 dołączanie do zespołu, 146
 udział w zebraniach zespołu, 147
aktywności projektowania systemu, 288, 303
aktywności racjonalizacji, 567
aktywności realizacji celów projektowych, 306
aktywności specyfikowania interfejsów, 416
aktywności zbierania wymagań, 166
aktywności „zwinnej” realizacji projektu, 673
aktywny przegląd projektu, 507
ALAP, 650
Albrecht A. J., 666
alfa-testowanie, 530
algorytm szyfrowania, 318
alternatives, 551
Ambler S., 739, 771
analityczny model obiektowy, 64, 213, 214, 401
analitik, 239
analiza, 48, 49, 51, 212
analiza opisu przypadków użycia, 219
analiza wymagań, 159, 211, 236
 aktywności, 217, 237
 analityczny model obiektowy, 214
ARENA, 245
cele, 212
diagramy sekwencji, 224
dokument analizy wymagań, 238
dokumentowanie, 238
identyfikacja agregacji, 231
identyfikacja atrybutów, 232
identyfikacja obiektów brzegowych, 220, 250
identyfikacja obiektów encji, 218, 245

- analiza wymagań
 - identyfikacja obiektów sterujących, 222, 251
 - identyfikacja skojarzeń, 228
 - iteracje modelu analitycznego, 241
 - komunikacja, 240
 - koncepty, 214
 - konsolidacja modelu analitycznego, 254
 - model dynamiczny, 214
 - modelowanie interakcji między obiektami, 228, 252
 - modelowanie relacji dziedziczenia między obiektami, 234
 - modelowanie zachowania poszczególnych obiektów uzależnionego od ich stanu, 233
 - obiekty brzegowe, 215
 - obiekty encji, 215
 - obiekty sterujące, 215
 - odzorowywanie przypadków użycia w obiekty, 224
 - przeglądy modelu analitycznego, 235
 - przydzielanie odpowiedzialności, 239
 - uzgodnienie modelu analitycznego z klientem, 243
 - weryfikacja modelu analitycznego, 254
 - zarządzanie analizą wymagań, 237
- analiza zorientowana obiektowo, 76
- AND decomposition, 566
- Andres C., 485, 633, 725
- API, 123, 270, 357
- Application Programmer Interface, 270, 357
- arbiter, 540
- Arbiter, 540
- architecture-centric management, 655
- architekt, 239, 331
- architekt systemu, 123
- architektura, 266
- architektura czterowarstwowa, 286, 287
- architektura filtr-potok, 286
- architektura klient-serwer, 283, 284
- architektura MVC, 281
- architektura oprogramowania, 279, 296
- architektura otwarta, 275
- architektura P2P, 284
- architektura peer-to-peer, 284
- architektura repozytoryjna, 279, 280
- architektura SOA, 348
- architektura trójwarstwowa, 285
- architektura warstwowa, 268
- architektura zamknięta, 275
- ARENA, 57, 190, 245, 303
 - analiza wymagań, 245
 - awarie, 346
 - deklaracja problemu, 190
 - odzorowywanie modelu na kod, 478
 - projektowanie systemu, 334
 - specyfikowanie interfejsów, 433
 - statystyki systemu, 478
 - wielokrotne wykorzystywanie rozwiązań wzorcowych, 390
 - wzorce projektowe, 390
- argumentacja, 552
- arguments, 552
- argumenty, 559
- argumenty komunikatu, 95
- Ariane 501, 114
- ArrayList, 74
- artifact road map, 728
- As late as possible, 650
- As soon as possible, 650
- ASAP, 649, 650
- ASD, 737
- asSet(), 414
- asynchroniczna komunikacja grupowa, 145
- asynchroniczne kolekcjonowanie racjonalizacji, 577
- asynchroniczne mechanizmy komunikacyjne, 138, 140
- ATTRACT, 743, 762
 - cel projektu, 743
 - kontrola, 745
 - modelowanie, 745
 - planowanie, 744
 - powtarzalność, 744
 - procesy cyklu życiowego, 745
 - projekt, 743
 - środowisko projektu, 744
 - wnioski, 746
 - wynik, 745
- atrybut klasy, 74
- atrybuty, 232, 247, 469
- audyt, 601
- audytor, 629
- automatyczne weryfikowanie spełnienia kontraktów, 465
- automatyzacja testowania, 538
- awaria, 324, 325, 346, 494, 499, 500
- AWT, 276

B

- Babatunda A. O., 740
- Babich W. A., 602, 632
- bag, 413
- Baker P., 543
- ball, 271
- ball-and-socket, 270
- Baron J. T. T., 151
- baseline, 597
- Basic Support for Cooperative Work, 145
- Bass L., 297, 348
- baza danych, 311, 469
 - tabele, 469

Beck K., 228, 258, 485, 633, 725
 Beedle M., 674, 678, 725
 benchmark test, 530
 Bersoff E. H., 602
 beta-testowanie, 526, 530
 bezpieczeństwo, 36, 162, 290, 568
 biblioteka, 603, 606
 biblioteka dynamiczna, 606
 biblioteka kontrolowana, 606
 biblioteka statyczna, 606
 biblioteki klas, 383
 bieżące kolekcjonowanie racjonalizacji, 553
 big-bang testing, 521
 Binder R. V., 542
 Birrer E. T., 382
 blackboard systems, 280
 blackbox tests, 504
 Blaha M., 311, 348, 449, 472
 blueprint, 46
 błąd roku 1900, 36
 błąd roku przestępnego, 36
 błędny stan, 494, 499, 500, 502
 błędy, 368, 494
 błędy w oprogramowaniu, 324
 BNF, 609
 Boehm B., 590, 669, 677, 687, 701
 Booch G., 107, 108, 162, 215, 703
 Borghoff U. W., 151
 Borning A., 440
 bottom-up testing, 521, 522
 brak klienta, 718
 breakage, 668
 Bridge, 368, 774
 Brooks F. P., 664, 678
 Brown W. J., 771
 Bruegge B., 382
 brygady tygrysów, 529
 BSCW, 145
 bucket theory of the mind, 41
 budowanie infrastruktury, 643
 budowanie wieloetapowe, 631
 budżet, 36
 bug, 494
 Burgess-Yakemovic K. C., 592
 buried association, 472
 burn down chart, 676
 burza mózgów, 133, 228, 241, 242, 333
 Buschmann F., 296, 394, 771

C

cache'owanie wyników czasochłonnych obliczeń, 457
 callback, 285, 383
 całościowe zarządzanie jakością, 740
 Campbell R. H., 382
 Capability Maturity Model, 632, 695

Carr M. J., 669
 Carroll J. M., 169, 205, 592
 CASE, 55, 452, 748
 Catalysis, 49
 cechy specyfikacji, 164
 cele analizy, 212
 cele projektowe systemu, 290
 cele testowania, 540
 Centralized Traffic Control, 555
 Champty J., 166
 change request, 602
 change set, 608
 chaordic system, 766
 Charette R. N., 669
 chief programmer organization, 679
 Christerson M., 107
 chroniona operacja, 405
 chroniony atrybut, 405
 chronometrażysta, 142
 Chung L., 565
 ciągi, 411, 413
 Class, Responsibilities i Collaborators, 228
 ClearCase, 610, 611
 Clements P., 297, 348
 CM aggregate, 602
 CMM, 695
 Cockburn A., 173, 206, 725, 737
 Cocoa, 378
 COCOMO II, 677, 727
 Cohn M., 677, 738
 Collection, 74
 Command, 775
 competitor testing, 630
 Composite, 776
 Computer Aided Software Engineering, 55
 Concurrent Version System, 611, 633
 configuration, 602
 configuration item, 602
 configuration management aggregate, 602
 Conklin J., 592
 consequent issues, 557
 Constantine L. L., 107, 206, 296
 continuous integration, 630
 Contract4J, 440
 CORBA, 276, 277, 283
 core use cases, 703
 correction, 499
 CRC, 228
 criteria, 552
 critical path, 649
 cross reference, 429
 CruiseControl, 631
 Crystal Clear, 737
 CTC, 555, 568
 kontrola dostępu, 583
 opis systemu, 569

Cunningham W., 228, 258
 Curtis B., 723
 CVS, 608, 609, 610, 611, 633
 cykl życiowy oprogramowania, 56, 637, 683, 686
 aktywności, 689
 dojrzałość modeli, 695
 grupy procesów, 688
 IEEE 1074, 687, 688
 iteracyjne modele ukierunkowane na aktywności,
 701
 modele, 685, 698
 modele ukierunkowane na aktywności, 685
 modele ukierunkowane na encje, 685, 706
 modelowanie cyklu życiowego, 690
 postrealizacja, 693
 prerealizacja, 691
 proces, 688
 procesy integralne, 694
 procesy międzyrealizacyjne, 694
 realizacja, 692
 sekwencyjne modele ukierunkowane
 na aktywności, 699
 tworzenie systemu, 692
 ujęcie produktowe, 686
 zależności czasowe między aktywnościami, 686
 zarządzanie projektem, 690
 cykle, 701, 703
 czas, 43
 czas przestoju, 649
 czas reakcji, 162
 czas trwania projektu, 719
 częstotliwość zmian, 719
 członkowie projektu, 116
 czynniki ryzyka, 670, 671
 czytelność modelu projektu systemu, 328

D

dane, 303
 dane trwałe, 309
 Dart S., 601
 dashboard, 631
 Date C. J., 469
 De Marco T., 107, 218
 dead reckoning, 684
 deadlock, 284
 debugowanie, 495, 496
 Decision Representation Language, 562, 564
 decisions, 552
 decyzje, 101, 120, 552
 decyzje implementacyjne, 72
 defekty, 494
 definiowanie
 atributy, 236
 cele projektowe, 290

problem, 642
 projekt, 116
 projekt wysokopoziomowy, 656
 skojarzenia, 236
 wyjątki związane z naruszeniem kontraktów, 448
 zakres projektu, 643
 założenia kontroli dostępu, 312, 340
 definiowany model kontroli procesów, 740
 deklaracja problemu, 129, 130, 642, 654, 655
 dekompozycja dysjunkcyjna, 566
 dekompozycja hierarchiczna, 275
 dekompozycja koniunkcyjna, 566
 dekompozycja systemu, 263, 269, 278, 295, 332, 401
 delegowanie, 359, 363
 delegowanie implementacji, 363
 delegowanie we wzorcach projektowych, 364
 deliverable, 640, 648
 delty, 608
 Deming E. W., 740
 demonstrowanie prototypów, 666
 dependability, 162
 design by contracts, 440
 diagramy aktywności, 68, 79, 101
 decyzje, 101
 partycje aktywności, 103
 rozwidlenia, 102
 węzły kontrolne, 101
 zastosowania, 103
 złączenia, 102
 diagramy De Marco, 72
 diagramy interakcji, 64, 67, 79, 95
 argumenty, 95
 komunikaty, 95
 zastosowanie, 97
 diagramy klas, 52, 64, 65, 78, 86, 255, 328
 agregacja, 92
 dziedziczenie, 93
 klasy, 87
 klasy skojarzeniowe, 88
 krotność skojarzenia, 89
 kwalifikowanie, 92
 łącza, 88
 metody, 94
 obiekty, 87
 operacje, 94
 redukcja krotności, 92
 role, 89
 skojarzenia, 88
 skojarzenia kwalifikowane, 92
 zastosowania, 94
 diagramy komunikacyjne, 97, 282
 diagramy PERT, 532
 diagramy przypadków użycia, 64, 65, 78, 79
 aktorzy, 79
 przypadki użycia, 79
 relacja dziedziczenia, 84

- relacja komunikacji, 82
- relacja rozszerzania, 83
- relacja zawierania, 82
- scenariusze, 85
- zależności komunikacyjne, 82
- diagramy sekwencji, 52, 67, 224, 225, 228, 252
- diagramy sieciowe, 640, 649
- diagramy stanów, 67, 79, 98, 233
 - akcje, 99
 - aktywności, 100
 - przejście między stanami, 98
 - przejście wewnętrzne, 100
 - stan, 98
 - zagnieżdżone maszyny stanów, 100
 - zagnieżdżony stan, 100
 - zastosowania, 101
- diagramy UML, 43
- diagramy wdrażania, 304
- Dijkstra E. W., 296, 440
- Directory, 91
- dojrzałość modeli cyklu życiowego, 695
- dokładność, 162
- dokonywanie podziału pracy, 657
- dokument analizy wymagań, 188, 238
 - system obecnie użytkowany, 189
 - system proponowany, 189
 - wprowadzenie, 188
- dokument projektu obiektów, 402, 425, 428
 - budowanie, 426
 - interfejsy klas, 429
 - pakiety, 429
 - utrzymywanie materiału źródłowego, 431
 - wstęp, 428
- dokument projektu systemu, 328, 329
 - architektura istniejącego systemu, 329
 - model projektu nowego systemu, 329
 - usługi oferowane przez podsystemy, 330
 - wprowadzenie, 329
- dokument RAD, 238
- dokument SDD, 328
- dokument umowy projektu, 642
- dokumentacja powtarzalnego rozwiązania, 388
- dokumentalista techniczny, 390, 432
- dokumentowanie
 - analiza wymagań, 238
 - projektowanie obiektów, 425
 - projekt systemu, 328
 - racjonalizacje, 585
 - testowanie, 532
 - transformacje, 475
 - wykorzystywanie gotowych rozwiązań, 388
 - zarządzanie konfiguracją, 627
 - zbieranie wymagań, 188
- dołączanie do infrastruktury komunikacyjnej, 118, 146

- dołączanie do zespołu, 118, 146
- DOORS, 187
- doskonalenie przypadków użycia, 160, 173, 198
- dostarczenie systemu, 644
- dostępność, 162, 335, 568
- doświadczenie uczestników, 717
- Douglass B. P., 258
- Doyle M., 151
- DRL, 562, 564
- dry run, 150
- Dutoit A. H., 586, 592
- Duval P., 630
- dwukierunkowe skojarzenia „jeden do jednego”, 459
- dynamiczna kontrola dostępu, 317
- dynamiczne witryny WWW, 385
- dynamika zmian, 433
- dyskusja rozproszonych uczestników w czasie rzeczywistym, 144
- dziedziczenie, 73, 93, 216, 359, 360
 - dziedziczenie implementacyjne, 360, 361, 362
 - dziedziczenie interfejsu, 362
 - dziedziczenie kontraktów, 424
 - dziedziczenie specyfikacyjne, 360, 362
 - dziedziczenie we wzorcach projektowych, 364
 - subklasy, 93
 - superklasy, 93
- dziedzina aplikacyjna, 76, 130
- dziedzina realizacyjna, 76

E

- earned value, 667
- egoless programming, 679, 732
- egzemplarze, 72
- Eiffel, 440, 465
- ekspert komponentu, 389
- Ekspert ma zawsze rację, 591
- ekspert wzorca projektowego, 389
- ekstender klasy, 403
- elementy działania, 561, 646
- elementy konfiguracji, 602, 603
- elementy ryzyka, 669
- Elssamadisy A., 771
- e-mail, 140
- emisja, 117, 134, 602, 605, 616
- Engineering Change Proposal, 602
- engineering workflows, 704
- entity-centered, 685
- Erl T., 348
- Erman L. D., 280
- erroneous state, 494, 499
- error, 494
- error guessing, 512
- etap zbierania wymagań, 50
- ewentualności, 551

ewolucja elementu konfiguracji, 608
 ExecuteTrip, 289
 exists(), 415
 eXtreme Programming, 485, 520, 526, 633, 725, 731

F

FAA, 599
 Fabryka abstrakcyjna, 376, 390, 772
 delegowanie, 376
 dziedziczenie, 376
 zastosowanie, 390
 Facade, 777
 facilitator, 142
 Facilitator, 143
 Fagan M. E., 496, 541
 failure, 494, 499
 faks, 138
 fałsyfikacja modeli, 77, 495
 Fasada, 295, 777
 FAT, 91
 fault, 494, 499
 Fayad M. E., 382
 faza definiowania projektu, 116
 Feature-Driven Design, 725
 Feiler P., 771
 Felsing J., 725
 field tests, 530
 Files, 91
 filtrowanie pakietów, 316
 filtry, 286
 Finish no earlier than, 650
 Finish no later than, 650
 firewall, 315
 Fisher R., 151, 589
 flat staffing, 664
 Flower M., 108
 Floyd R. W., 440
 fly-by-wire, 598
 FNET, 650
 FNLT, 650
 follproof, 496
 forAll(), 415
 foreign key, 470
 formowanie zespołów, 643, 663
 formularz formalizujący żądanie zmiany, 138
 formułowanie ograniczeń, 423
 forward engineering, 427, 452, 706
 Fowler M., 394, 450, 457, 737, 771
 Frames, 394
 framework, 383, 384
 frameworki aplikacyjne, 381, 382
 frameworki białoskrzynkowe, 382
 frameworki czarnoskrzynkowe, 383
 frameworki infrastrukturalne, 382

 frameworki pośredniczące, 382
 NFR, 565
 Freeman-Benson B., 440
 FRIEND, 79, 167, 743, 746, 762
 aktywności cyklu życiowego projektu, 751
 cel projektu, 746
 dokumenty systemowe, 749
 iteracje, 753
 kontrola, 750
 modelowanie, 748
 planowanie, 747
 powtarzalność, 747
 procesy cyklu życiowego, 750
 scenariusze, 170
 środowisko projektu, 747
 wnioski, 753
 wynik, 752
 zespoły funkcyjne, 750
 funkcje projektowe, 646
 FURPS, 162
 FURPS+, 162

G

Gaffney J. E. Jr., 666
 gałęzie, 607
 Gamma E., 283, 295, 308, 394, 771
 Garlan D., 296
 generalizacja, 216, 234, 360
 generowanie dokumentu ODD, 428
 gIBIS, 592
 Gladwin T., 719
 Gleick J., 766
 globalna tabela dostępu, 314
 globalny przepływ sterowania, 319, 341
 Glover A., 630
 główny architekt, 432, 477
 gniazda TCP/IP, 276
 gniazdo, 270
 Goldberg A., 394
 gra planistyczna, 733
 gradual staffing, 664
 graf PERT, 126
 graf przepływów, 513
 graficzne reguły reprezentowania widoków, 71
 graniczne przypadki użycia, 323
 Grenning J., 677, 738
 grep, 287
 groupware, 577, 578
 grupa procesów, 688
 grupa produktów, 640
 grupowanie klas, 105
 grupowanie przypadków użycia, 104
 grupy dyskusyjne, 140
 Guttag J., 440

H

Halstead M. H., 666
Hammer M., 166
Hamu D. S., 382
Harel D., 98
harmonogram, 115, 126, 640, 646
 graf PERT, 126
 ścieżka krytyczna PERT, 127
 wykres Gantta, 126
harmonogram przeglądów, 118, 150
HashMap, 74
Hayes-Roth F., 280
HEARSAY II, 280
Helm R., 283
Henderson V. D., 602
hermetyzacja algorytmów, 780
hermetyzacja hierarchii, 378
hermetyzacja kontekstu, 373
hermetyzacja kosztownych obiektów, 779
hermetyzacja niekompatybilnych komponentów, 371
hermetyzacja platformy, 376, 772
hermetyzacja podsystemów, 295, 777
hermetyzacja przechowywania danych, 368
hermetyzacja przepływu sterowania, 377, 775
heurystyki Abbotta, 218
heurystyki identyfikowania wzorców projektowych, 381
heurystyki komunikowania racjonalizacji, 588
heurystyki metodologiczne, 765
heurystyki pisanie przypadków użycia, 175
heurystyki pisanie scenariuszy, 175
heurystyki pomocne w czytelnym formułowaniu ograniczeń, 423
heurystyki pomocne w grupowaniu obiektów w podsystemy, 295
heurystyki pomocne w identyfikowaniu atrybutów, 233
heurystyki pomocne w identyfikowaniu obiektów, 181
heurystyki pomocne w identyfikowaniu obiektów brzegowych, 221
heurystyki pomocne w identyfikowaniu obiektów encji, 220
heurystyki pomocne w identyfikowaniu obiektów sterujących, 223
heurystyki pomocne w identyfikowaniu skojarzeń, 230
heurystyki pomocne w odwzorowywaniu naruszenia kontraktów w wyjątki, 468

heurystyki weryfikowania kompletności zestawu przypadków użycia, 182
heurystyki wspomagające rysowanie diagramów sekwencji, 227
heurystyki wspomagające zarządzanie gałęziami, 621
heurystyki wyboru między relacjami zawierania i rozszerzania, 179
heurystyki wyboru wzorców projektowych, 379, 781

hierarchiczna organizacja projektu, 120
Highsmith J., 725, 735, 737
Hoare C. A. R., 440
Hock D., 766
Hofmeister C., 297
hook methods, 382
Hopper G. M., 541
HTTP, 283, 384

I

IBIS, 562, 563, 592
iContract, 440, 468
idealny tydzień, 733
identyfikacja
 agregacje, 231
 agregaty CM, 613
 aktorzy, 160, 167, 192
 atributy, 232
 cele projektowe, 290, 335
 elementy konfiguracji, 600, 613
 frameworki aplikacyjne, 381
 obiekty brzegowe, 220, 250
 obiekty encji, 218, 220, 245
 obiekty modelu analitycznego, 179
 obiekty sterujące, 222, 251
 obiekty trwałe, 310, 339
 podsystemy, 294, 336
 przypadki użycia, 160, 171, 195
 relacje, 198
 relacje między aktorami a przypadkami użycia, 176
 relacje między przypadkami użycia, 160
 scenariusze, 160, 169, 192
 skojarzenia, 228
 trwałe dane, 309
 umiejętności, 660
 usługi, 321, 343
 warunki graniczne, 323, 345
 wymagania pozafunkcyjne, 160, 182, 184, 204
identyfikatory wersji, 606
identyfikowalna specyfikacja, 165
identyfikowalność, 160, 187
idiotoodporność, 496
IEEE 1074, 687, 688, 698, 709
implementator, 123, 403
implementowanie, 53
includes(), 414
infrastruktura komunikacyjna, 118, 660
inicjowanie projektu, 690
Inline Class Refactoring, 457
inspection, 495
inspekcja kodu, 132, 495, 666
inspekcja komponentu, 506, 507
inspekcja problemu, 117
instalowanie systemu, 644, 672

instancja klasy, 74
 instancje aktorów uczestniczących, 86
 int, 72
 integracja ciągła, 630, 633
 integrowanie, 553
 integrowanie pionowe, 521, 525
 integrowanie poziome, 521
 interakcje między uczestnikami projektu, 120
 interdyscyplinarność, 22
 interfejs API, 123
 interfejs podsystemów, 270
 interfejs programisty, 270, 357
 interfejs użytkownika, 378
 interfejs wzorca, 365
 interfejsy, 285, 399
 internacjonalizacja, 162
 internal work products, 648
 intersection(), 414
 inv., 415
 inżynier API, 123
 inżynieria, 35
 inżynieria interfejsu, 165
 inżynieria odwracająca, 427, 449, 452, 706
 inżynieria oprogramowania, 35, 38, 266, 716
 aktywności, 49
 konceptje, 43
 modelowanie, 38, 39
 niepowodzenia, 36
 pozyskiwanie wiedzy, 38, 41
 proces sterowany racjonalizacją, 38
 racjonalizacja, 42
 rozwiązywanie problemów, 38, 40
 inżynieria pierwotna, 165
 inżynieria postępująca, 427, 448, 452, 706
 inżynieria wahadłowa, 706
 inżynieria wtórna, 165
 inżynieria wymagań, 157
 inżynierskie przepływy pracy, 704
 Islam N., 382
 issue, 551
 Issue-Based Information System, 562, 563
 issue-based life cycle model, 707
 iteracje modelu analitycznego, 241
 burza mózgów, 242
 dojrzewanie, 242
 ustalenie, 242
 iteracje projektowania systemu, 333
 iteracyjne modele ukierunkowane na aktywności, 701
 jednolity proces wytwarzania oprogramowania, 703
 model spiralny, 701
 iteracyjne planowanie, 738

J

JAA, 599
 Jacobson I., 107, 108, 162, 205, 215, 709
 JAD, 160, 185
 agenda sesji, 185
 badania, 185
 definiowanie projektu, 185
 dokument końcowy, 185
 przygotowania, 185
 sesja, 185
 słownik menedżera, 185
 wstępna specyfikacja, 185
 wywiady, 185
 JAMES, 136, 743, 754, 762
 aktywności, 757
 cel projektu, 754
 iteracje, 756
 kontrola, 759
 modelowanie, 756
 planowanie, 755
 powtarzalność, 756
 procesy cyklu życiowego, 757
 środowisko projektu, 755
 wnioski, 760
 wynik, 760
 Java, 72, 258, 269
 klasy abstrakcyjne, 74
 Java RMI, 276, 283
 JavaCard, 758
 Javadoc, 429, 440
 jContractor, 440
 jeden do jednego, 90
 jeden na wiele, 90
 jednokierunkowe skojarzenia „jeden do jednego”, 459
 jednolity proces, 532, 725
 jednolity proces wytwarzania oprogramowania, 703
 jednostki pracy, 640
 jednoznaczność specyfikacji, 164
 Jensen R. W., 687
 język
 Ada, 606
 Eiffel, 440
 Java, 72
 język z wbudowaną kontrolą typów danych, 72
 OCL, 107, 359, 407
 SQL, 470
 UML, 43, 63, 64
 Johnson R., 283
 Joint Application Design, 160, 185
 Jones T. C., 496
 Jonsson P., 107
 JUnit, 538, 539

K

- kamienie milowe, 117, 118, 665
- karty CRC, 228, 258
- karty inteligentne, 754
- katalog główny, 603, 606
- katastrofa Ariane, 114
- katastrofa Challengeera, 638
- Katzenbach J. R., 677
- Kay A., 394
- Kayser T. A., 151, 662
- Kelly J. F., 740
- Kemerer C. F., 624
- Key Process Areas, 696
- kierownicze aspekty zarządzania konfiguracją, 627
- klasy, 65, 73, 86, 268
 - atrybuty, 74, 232
 - dziedziczenie, 73, 93
 - ekstender, 403
 - implementator, 403
 - instancja, 74
 - operacje, 74
 - użytkownik, 403
- klasy abstrakcyjne, 73
- klasy bazowe, 360
- klasy bezpośrednio powiązane, 412
- klasy implementacyjne, 366
- klasy klienckie, 365, 403
- klasy pochodne, 360
- klasy powiązane pośrednio, 412
- klasy rozszerzające, 367
- klasy skojarzeniowe, 88, 89, 464
- klasy zdarzeniowe, 75
- klasyczne zarządzanie projektem, 653
- Klepp A., 407
- klient, 124, 239, 283
- klient pełnomocnik, 718
- klient prezentacji, 286
- klient-serwer, 283, 284
- klimat technologiczny, 718
- klucze główne, 470
- klucze kandydackie, 470
- klucze obce, 470
- kluczowe obszary procesu, 696
- know-how, 722
- Knuth D. E., 607
- kolapsacja, 456
- kolekcje, 413
- kolekcje OCL, 411
- kolekcjonowanie racjonalizacji, 562, 569
- komentarze Javadoc, 429, 440
- komisja, 119
- kompletność modelu, 235
- kompletność modelu projektu systemu, 327
- kompletność specyfikacji, 164
- komponenty, 306, 384
- komponenty fizyczne, 269
- komponenty logiczne, 269
- komponenty testowe, 540
- Kompozyt, 378, 379, 776
- kompromisy projektowe, 293
- komunikacja, 22, 55, 79, 113, 117
 - analiza wymagań, 240
 - emisje, 117
 - inspekcja problemu, 117
 - mechanizmy, 138
 - projektowanie systemu, 331
 - przeglądy klienckie, 117
 - przeglądy partnerskie, 117
 - przeglądy projektowe, 117
 - rozwiązywanie problemów, 117
 - zdarzenia nieprzewidywalne, 117
 - zdarzenia przewidywalne, 117
 - zebrania statusowe, 117
 - żądanie wyjaśnień, 117
 - żądanie zmian, 117
- komunikacja między uczestnikami, 432
- komunikacja partnerska, 122
- komunikacja planowa, 118, 128, 139
 - burza mózgów, 133
 - deklaracja problemu, 129
 - emisje, 134
 - przeglądy klienckie, 131
 - przeglądy partnerskie, 132
 - przeglądy post mortem, 134
 - przeglądy projektowe, 132
 - przeglądy statusowe, 133
- komunikacja pozaplanowa, 118, 135, 136
 - rozwiązywanie problemów, 137
 - żądanie wyjaśnień, 136
 - żądanie zmian, 137
- komunikatory, 140
- komunikaty, 75, 76, 94, 95
 - argumenty, 95
- komunikowanie, 120
- koncepcja wielokrotnego wykorzystania, 359
- konceptcje komunikacyjne projektu, 128
- konceptcje organizacyjne projektu, 119
- konceptcje zbierania wymagań, 161
- konfiguracja, 602, 604
- konfiguracja oprogramowania, 56
- konfiguracja sprzętowa, 306
- konfiguracyjne przypadki użycia, 345
- konflikty, 620, 623
- konserwacja systemu, 54
- konsolidacja modelu, 236
- konsolidacja modelu analitycznego, 254
- konsultanci, 124
- kontakt z użytkownikami, 718
- kontekst testowania, 540
- kontrakty, 406
- kontrola dostępu, 304, 312, 340

kontrola projektu, 644, 665, 675
 demonstrowanie prototypów, 666
 inspekcje kodu, 666
 kamienie milowe, 665
 metryki, 666
 przeglądy projektu, 666
 zarządzanie ryzykiem, 669
 zebrania, 665
 kontrola zmian, 601
 kontroler, 281, 629
 konwersacja okazjonalna, 140
 kończenie pracy systemu, 345
 kończenie projektu, 671, 677
 akceptacja systemu, 672
 instalowanie systemu, 672
 refleksje post mortem, 673
 koszty operacyjne, 335
 kółko-gniazdo, 271
 KPA, 696
 Kramer R., 440, 468
 Kraut R. E., 151
 krotki, 311
 krotność skojarzenia, 89
 Kruchten P., 709
 krystaliczną czystość, 737
 kryteria, 552
 kryteria kosztowe, 291
 kryteria pewności działania, 291
 kryteria pielęgnowalności, 292
 kryteria użytkownika, 292
 kryteria wydajności, 291
 Kunz W., 562, 592
 kursy projektowania, 26
 kursy technologiczne, 26
 kursy wprowadzające, 26
 kwalifikowanie, 92
 kwantyfikatory OCL, 415
 kwestionariusze, 140, 141

L

Larman C., 258, 766, 771
 Leveson N. G., 348
 liczebność zespołów, 662
 light weight, 737
 light-headed, 737
 linia bazowa, 189, 597, 604
 LinkedList, 74
 Liskov B., 440
 listy kontroli dostępu, 314
 listy uprawnień, 314
 lizak, 271, 321
 local king client, 717
 Lockwood L. A. D., 206
 logika aplikacji, 285
 lokalne atrybuty, 412

lokalny klient samodzielny, 717
 lollipop, 271
 Lotus Notes, 140

Ł

łatwość utrzymania, 162
 łączy, 88
 łącznik, 121, 123
 łącznik architektoniczny, 331, 432, 477

M

macierz kontroli dostępu, 314
 macierz kwalifikacji, 650
 Mack R. L., 542
 MacLean A., 592
 magazynowanie danych, 285
 Malveau R. C., 771
 mapa artefaktów, 728
 mapy stanów, 98
 Martin J., 394
 master directory, 603
 maszyna stanu, 98
 Matyas S., 630
 Mayhew D. J., 543
 McCabe T., 666
 mean time between failures, 162
 mechanizm odwołania zwrotnego, 285
 mechanizmy komunikacyjne, 138
 asynchroniczne, 138
 synchroniczne, 138, 140
 menedżer konfiguracji, 124, 240, 331, 390, 432, 629
 Menedżer ma zawsze rację, 591
 menedżer projektu, 637, 722
 metamodel dziedziczenia, 362
 metoda Fagana, 507
 metoda Joint Application Design, 185
 metodologia Royce'a, 48, 725
 artefakty, 727
 cechy, 725
 dojrzałość procesów, 729
 doświadczenie dziedziczne, 729
 elastyczność procesów, 729
 elementy, 726
 kontrola, 730
 mapa artefaktów, 728
 metryki jakościowe, 730
 metryki menedżerskie, 730
 modelowanie, 727
 planowanie, 726
 powtarzalność, 727
 procesy cyklu życiowego, 728
 ryzyko architektoniczne, 729
 skala projektu, 728
 spójność udziałowców, 729

- metodologia rugby, 737
 - iteracyjne planowanie, 738
 - kontrola, 741
 - modelowanie, 739
 - planowanie, 738
 - powtarzalność, 739
 - procesy cyklu życiowego, 740
 - metodologie, 48, 713, 717, 719, 724
 - Boocha, 48
 - Catalysis, 49
 - kontrola, 723
 - metodologia jednolitego procesu, 49
 - modelowanie, 721
 - monitorowanie, 723
 - OMT, 48
 - planowanie, 719
 - powtarzalność, 720
 - procesy cyklu życiowego, 723
 - prze definiowanie celów projektu, 724
 - Scrum, 674, 737
 - XP, 731
 - metody, 48, 94, 717
 - metody zahaczane, 382
 - metryki, 666
 - Meyer B., 440
 - MFO, 650
 - miara dobroci, 559
 - miary dojrzałości, 668
 - miary kondycji finansowej projektu, 667
 - miary modularności, 668
 - miary postępu technicznego, 668
 - miary stabilności, 668
 - middleware, 276, 355, 359, 384, 571, 744
 - mierniki ilościowe, 666
 - minimalizacja ryzyka, 672
 - Minsky Marvin, 394
 - misja STS-51L, 638
 - mistrz, 674, 675
 - model, 43, 46, 69, 70, 281, 721
 - archiwizowanie, 722
 - komunikowanie, 721
 - projektowanie, 721
 - model analityczny, 94, 213, 214, 455, 705
 - model analityczny systemu planowania podróży, 288
 - model bazarowy, 619
 - model cyklu życiowego oprogramowania, 685, 689, 698
 - model dojrzałości organizacyjnej, 695
 - model dynamiczny, 64, 213, 214
 - model dziedziny aplikacyjnej, 77
 - model funkcjonalny, 64, 213
 - model FURPS, 162
 - model FURPS+, 162
 - model implementacji, 705
 - model kaskadowy, 42, 699, 708
 - model maszyny stanów UML, 98
 - model menedżerski, 640
 - model obiektowy, 64
 - model OMT, 748
 - model OSI-ISO, 276, 277
 - model projektu obiektów, 64
 - model projektu systemu, 327, 705
 - model przypadków użycia, 705
 - model racjonalizacji, 585
 - model sekwencyjny, 687
 - model spiralny, 701
 - model systemu, 69
 - model testowania, 498, 705
 - model ukierunkowany na aktywności, 685
 - model ukierunkowany na encje, 685, 706
 - zagadnieniowy model cyklu życiowego, 707
 - model zadań, 640, 646, 649
 - model „zębaty”, 709
 - modelowanie, 38, 39, 69, 70, 721
 - język UML, 63
 - modelowanie cyklu życiowego, 690
 - modelowanie interakcji między obiektami, 228, 252
 - modelowanie relacji dziedziczenia między obiektami, 234
 - modelowanie systemów nierealistycznych, 39
 - modelowanie zachowania poszczególnych obiektów uzależnionego od ich stanu, 233
 - modelowanie zorientowane obiektowo, 76
 - Model-View-Controller, 258, 281
 - model-widok-kontroler, 258, 281
 - Moduła 2, 269
 - modyfikacja planów projektu, 644
 - modyfikowalność, 290
 - Moran T. P., 592
 - Most, 368, 774
 - delegowanie, 370
 - dziedziczenie, 370
 - Mowbray T. J., 771
 - MSO, 649, 650
 - MTBF, 162, 668
 - MUE, 607
 - Must finish on, 650
 - Must start on, 650
 - MVC, 258, 281
 - My UML Editor, 607
 - Myers G. J., 512, 542
 - MyTrip, 307, 308, 313, 323, 325
- ## N
- nadklasy, 73
 - nadużycie interfejsu, 36
 - namiatka testowa, 499, 505
 - narzędzia, 717
 - narzędzia komunikacji grupowej, 144
 - narzędzia wspomagające zarządzanie konfiguracją, 610

nauka, 38
 nauki o sztuczności, 39
 nauki przyrodnicze, 38
 nauki społeczne, 38
 nawigacja, 88
 nawigacja polinezyjska, 684
 nazwy
 przypadki użycia, 80
 scenariusze, 85
 negocjowanie specyfikacji z klientem, 185
 Netmeeting, 144
 network diagram, 649
 Neumann P., 59
 NFR, 562, 565, 592
 cele realizacyjne, 566
 dekompozycja, 565
 niby-klient, 718
 Niech czas zdecyduje, 591
 niejednoznaczności, 212
 Nielden J., 542
 niepowodzenia w inżynierii oprogramowania, 36
 niezawodność, 162, 290, 494
 niezawodność oprogramowania, 494, 495
 niezawodny system, 325
 niezmienniki, 406
 NIH, 387
 no client, 718
 Nonaka I., 677, 737
 Norman D. A., 206, 740
 Not Invented Here, 387
 notacja, 48, 71
 notacja Backusa-Naura, 609
 notacja Boocha, 64
 notacja kółko-gniazdo, 270, 271
 notacja kropkowa, 408
 notacja OMT, 64
 notacja OOSE, 64
 notacja UML, 63
 notatki, 105
 nowe technologie, 367
 nowe widoki, 368
 nowi dostawcy, 367
 numery wersji, 606

O

obiekt uczestniczący, 67
 obiektowa baza danych, 311
 obiektowy model projektu systemu, 64
 obiekt, 73, 74, 86
 operacje, 94
 przejście między stanami, 98
 stan, 98
 obiekty aplikacyjne, 359
 obiekty brzegowe, 215, 220, 342
 obiekty encji, 215, 218, 342
 obiekty modelu analitycznego, 179
 obiekty projektu, 125
 obiekty proxy, 308
 obiekty realizacyjne, 359
 obiekty sterujące, 215, 342
 obiekty trwałe, 125, 310
 obiekty uczestniczące, 181
 Object Constraint Language, 107, 359, 407
 Object Design Document, 402, 425
 Object Modeling Technique, 48, 64
 Object-Oriented Software Engineering, 64
 obsada stopniowana, 664
 Observer, 778
 Obserwator, 283, 390, 778
 zastosowanie, 393
 obsługa sytuacji wyjątkowych, 324
 OCL, 107, 359, 407
 exists(), 415
 forAll(), 415
 kolekcje, 411, 413
 kontrakty, 421
 kwantyfikatory, 415
 niezmienniki, 421
 odwołania do atrybutów, 414
 ograniczenia, 408
 operacje kolekcji, 414
 warunki końcowe, 409
 warunki wstępne, 408
 ODD, 402, 425, 534, 748
 oddzielenie encji od widoków, 778
 Odell J. J., 394
 odporność na awarie, 290
 odwołania skrośne, 429
 odwołania zwrotne, 285
 odwzorowanie kontraktów w wyjątki, 465, 482
 odwzorowanie modeli klas w schematy
 przechowywania danych, 448
 odwzorowanie modelu na kod, 445
 aktywności, 454
 ARENA, 478
 cele, 447
 dokumentowanie transformacji, 475
 inżynieria odwracająca, 449, 452
 inżynieria postępująca, 448, 452
 konceptje, 448
 odwzorowywanie kontraktów w wyjątki, 465, 482
 odwzorowywanie modelu obiektowego w schemat
 bazy danych, 469, 484
 odwzorowywanie skojarzeń na kolekcje, 458, 480
 optymalizacja modelu, 447, 455
 przydzielanie odpowiedzialności, 477
 refaktoryzacja kodu, 448, 450
 transformacja modelu, 448, 449, 453
 zarządzanie transformacjami, 475
 zasady transformacji, 453

- odzworowanie modelu obiektowego w schematy bazy danych, 469, 484
 - odzworowanie klas i atrybutów, 470
 - odzworowanie pionowe, 473
 - odzworowanie poziome, 475
 - odzworowanie relacji dziedziczenia, 473
 - odzworowanie skojarzeń, 472
 - skryte powiązania, 472
 - tabela pośrednicząca, 472
 - tabela skojarzeniowa, 472
 - odzworowanie oprogramowania w konkretny sprzęt, 303
 - odzworowanie pionowe, 473
 - odzworowanie podsystemów w procesory
 - i komponenty, 306, 337
 - odzworowanie poziome, 473, 475
 - odzworowanie przypadków użycia w obiekty, 224
 - odzworowanie skojarzeń na kolekcje, 458, 480
 - ogół-szczegóły, 217
 - ograniczenia, 106, 107, 163
 - ograniczenia ASAP, 649
 - ograniczenia czasowe, 649, 650
 - ograniczenia MFO, 649
 - ograniczoność zasobów, 22
 - okno projektowe, 334
 - okresowe zebrania statusowe, 665
 - OMT, 48, 64, 748
 - analiza, 48
 - projekt obiektów, 48
 - projektowanie systemu, 48
 - OMTool, 748
 - OOPSLA, 258
 - OOSE, 64, 166
 - operacje, 74, 94
 - opóźnianie kosztownych obliczeń, 457
 - oprogramowanie, 35
 - oprogramowanie pośrednie, 355, 359, 384
 - optymalizacja modelu, 357
 - optymalizacja modelu obiektowego, 353, 455
 - optymalizacja ścieżek dostępu, 455
 - OR decomposition, 566
 - organizacja diagramów, 104
 - organizacja głównego programisty, 679
 - organizacja łącznikowa, 122
 - organizacja projektu, 113, 119
 - organizacja przeglądów, 149
 - organizacja zespołowa, 119
 - organization chart, 640
 - organizowanie projektu, 659, 675
 - formowanie zespołów, 663
 - identyfikacja umiejętności, 660
 - podpisanie umowy projektu, 664
 - przydzielanie ról technicznych, 661
 - przypisywanie ról menedżerskich, 661
 - ustanowienie infrastruktury komunikacyjnej, 660
 - wybór liczności zespołów, 662
 - wyrównywanie braku kwalifikacji, 662
 - zebranie otwierające, 664
 - Orr O., 736
 - OSI-ISO, 276
 - otoczką dla starszego kodu, 773
 - Overgaard G., 107
 - OWL, 130, 669
- ## P
- P2P, 284, 285
 - package, 405
 - Paech B., 586, 592
 - pakiet, 126
 - pakietowa operacja, 405
 - pakietowy atrybut, 405
 - pakiety, 104
 - grupowanie klas, 105
 - grupowanie przypadków użycia, 104
 - pakiety pracy, 640, 646
 - Palmer S., 725
 - Parnas D., 296, 507
 - participants, 640
 - Partsch H., 485
 - partycje, 275
 - partycje aktywności, 103
 - partycjonowanie, 268, 278
 - partycjonowanie aktywności, 103
 - Paulish D. J., 677
 - Paulk M. C., 695, 723
 - peer-to-peer, 284
 - PEM, 130
 - percepcja dwustabilna, 212
 - Perforce, 610, 611
 - Personal Environment Module, 130
 - PERT, 126, 532
 - Petroski H., 739, 766
 - pewność działania, 162
 - piłka, 271
 - Plan testów, 125, 497, 533, 534
 - Plan Zarządzania Konfiguracją Oprogramowania, 627, 628
 - aktywności, 628
 - harmonogram, 628
 - konserwacja planu, 628
 - wstęp, 628
 - zarządzanie, 628
 - zasoby, 628
 - Plan Zarządzania Projektem, 651, 654, 690
 - Planning Poker, 677, 738
 - planowanie, 719
 - planowanie projektu, 654, 674
 - deklaracja problemu, 654, 655
 - Plan Zarządzania Projektem, 654
 - podział pracy, 657
 - projekt wysokopoziomowy, 656

- planowanie projektu
 - SPMP, 654
 - tworzenie początkowego harmonogramu, 659
 - wysokopoziomowy projekt systemu, 654
- planowanie testów, 497, 532
- PlanTrip, 288
- platforma sprzętowa, 306
- pliki, 311
- pluskwy, 494
- płaska obsada, 664
- początkowa identyfikacja obiektów modelu
 - analitycznego, 179
- początkowy harmonogram, 659
- pojęcie rugby, 737
- pojęcie zorientowane obiektowo, 40
- podklasy, 73
- podmiana implementacji, 774
- podpisanie umowy projektu, 664
- podsystemy, 70, 267, 268, 294
- podtypowanie, 363
- pojedynczy projekt, 623
- poker planistyczny, 738
- Polecenie, 377, 390, 775
 - delegowanie, 378
 - dziedziczenie, 378
 - zastosowanie, 392
- polimorfizm, 517
- pomocnicze przepływy pracy, 704
- Popper K., 41, 59, 495
- poprawki, 499, 505
- poprawność modelu projektu systemu, 327
- poprawność specyfikacji, 164
- Portny S. E., 677
- post mortem, 644, 673
- post:, 409
- postrealizacja, 693
- potencjalnie dostarczalne przyrosty produktu, 674
- potentially deliverable product increment, 674
- Poter A. A., 496
- potoki, 286
- powtarzalna architektura, 720
- powtarzane trawersowanie skojarzeń, 455
- poziomy dojrzałości organizacji cyklu życiowego, 695
- pozyskiwanie wiedzy, 38, 41, 42
- praca, 640
- praca grupowa, 577
- prawa dostępu, 313
- pre:, 408
- Premierlani W., 311, 348, 449, 472
- prerealizacja, 691
- primary key, 470
- priorytety funkcja systemu, 243
- priorytetyzacja czynników ryzyka, 671
- private, 405
- Problem Statement, 642, 655
- problemy z oprogramowaniem, 37
- problemy z użytecznością, 158
- proces, 688
- proces falsyfikacji, 78
- proces jednolity, 703
- proces przetwarzania wymagań, 689
- proces rewizyjny, 244
- proces sterowany racjonalizacją, 38
- procesor, 306
- procesy cyklu życiowego, 723
- procesy integralne, 694
- procesy międzyrealizacyjne, 694
- product backlog, 674
- product owner, 675
- produkt, 43, 46, 115, 124, 640, 646
- produkt docelowy, 46, 115
- produkt finalny, 640, 648
- produkt wewnętrzny, 46, 125, 648
- profile, 539
- programista, 477, 629
- programowanie bezosobowe, 679, 732
- programowanie ekstremalne, 485, 520, 633, 725, 731
 - aktywności, 734
 - Częste integrowanie, 735
 - emisje systemu, 733
 - gra planistyczna, 733
 - idealny tydzień, 733
 - kontrola, 735
 - modelowanie, 734
 - Najpierw testy, 734
 - partner, 732
 - planowanie, 732
 - powtarzalność, 734
 - procesy cyklu życiowego, 734
 - Programowanie parami, 735
 - prowadzący, 732
 - Refaktoryzacja przed rozszerzeniem, 735
 - system samoorganizujący się, 735
 - szybkość projektu, 733
 - testy, 734
 - Testy dla każdej nowej usterki, 735
 - zasady, 732
- programowanie sterowane problemami, 42
- programowanie sterowane ryzykiem, 42
- programowanie wielowątkowości, 321
- Project Agreement, 642, 664
- project functions, 646
- project kick off meeting, 674
- project velocity, 733
- projekt, 43, 56, 113, 115
 - członkowie, 116
 - faza definiowania projektu, 116
 - faza startowa, 117
 - faza terminalna, 117
 - faza ustalona, 117
 - harmonogram, 115, 126
 - interakcje między uczestnikami projektu, 120

- komunikacja, 117
- komunikacja planowa, 128
- komunikacja pozaplanowa, 135
- koncepte komunikacyjne, 128
- koncepte organizacyjne, 119
- organizacja hierarchiczna, 120
- organizacja projektów, 119
- organizacja zespołowa, 119
- produkt, 115, 124
- realizacja, 116
- role, 122
- struktura łącznikowa, 121, 122
- uczestnicy, 116
- zadanie, 116, 124
- zespoły, 119
- projekt mieszkania, 264
- projekt obiektów, 48, 49
- projekt systemu, 51
- projekt systemu informatycznego, 37
- projekt wysokopoziomowy, 656
- projekt XP, 743
- projektant obiektów, 123, 432
- projektowanie, 49, 721
- projektowanie globalnego przepływu sterowania, 319, 341
- projektowanie niezawodnych systemów, 325
- projektowanie obiektów, 53, 353, 355
 - aktywności, 356
 - ARENA, 390
 - biblioteki klas, 383
 - delegowanie, 363
 - dokumentowanie wykorzystania gotowych rozwiązań, 388
 - dziedziczenie implementacyjne, 360
 - dziedziczenie specyfikacyjne, 360
 - elementy, 353
 - gotowe komponenty, 356, 367
 - hermetyzacja hierarchii, 378
 - hermetyzacja kontekstu, 373
 - hermetyzacja niekompatybilnych komponentów, 371
 - hermetyzacja platformy, 376
 - hermetyzacja przechowywania danych, 368
 - hermetyzacja przepływu sterowania, 377
 - heurystyki wyboru wzorców projektowych, 379
 - identyfikacja frameworków aplikacyjnych, 381
 - komponenty, 384
 - obiekty aplikacyjne, 359
 - obiekty realizacyjne, 359
 - optymalizacja modelu, 357
 - przydzielanie odpowiedzialności, 389
 - przystosowywanie frameworków aplikacyjnych, 381
 - restrukturyzacja modelu, 357
 - specyfikowanie interfejsów, 357, 399
 - wielokrotne wykorzystanie, 359
 - wybór wzorców projektowych, 367
 - wzorce projektowe, 356, 364
 - zarządzanie projektowaniem obiektów, 425
 - zarządzanie wykorzystywaniem gotowych rozwiązań, 386
 - zasada zastępowania Liskov, 364
- projektowanie sterowane cechami, 725
- projektowanie sterowane odpowiedzialnością, 166
- projektowanie systemu, 48, 49, 51, 263, 266, 305, 552
 - aktywności, 288
 - architektura programowa, 267
 - architektura warstwowa, 268
 - ARENA, 334
 - cele projektowe, 267
 - definiowanie celów projektowych, 290
 - dokumentowanie, 328
 - doskonalenie dekompozycji stosownie do celów projektowych, 263
 - graniczne przypadki użycia, 267
 - hermetyzacja podsystemów, 295
 - identyfikacja celów projektowych, 290
 - identyfikacja podsystemów, 294
 - interfejsy podsystemów, 270
 - interfejsy programisty, 270
 - iteracje, 333
 - klasy, 268
 - kompromisy projektowe, 293
 - komunikacja, 331
 - koncepte, 267
 - kryteria kosztowe, 291
 - kryteria pewności działania, 291
 - kryteria pielęgnowalności, 292
 - kryteria użytkownika, 292
 - kryteria wydajności, 291
 - odwzorowanie podsystemów w procesory i komponenty, 337
 - partycje, 275
 - partycjonowanie, 268
 - podsystemy, 267, 268
 - projektowanie wstępnej dekompozycji, 263
 - rozpoznawanie celów projektowych, 263
 - spoistość, 267, 271, 272
 - sprzężenie, 267, 271
 - style architektoniczne, 279
 - usługi, 267, 270
 - warstwy, 275
 - zarządzanie projektowaniem systemu, 328
- projektowanie za pomocą kontraktów, 440
- promocja, 602, 605, 615
- promotion, 602
- propozycja, 557
- protected, 405
- protokolant, 142, 587
- protokół z zebrania, 143, 144
- prototyp interfejsu użytkownika, 509
- prototyp pionowy, 334, 509

prototyp poziomy, 334, 509
 prototyp z krainy Oz, 509
 prototypowanie, 77, 78
 prototypy, 666
 proxy, 308
 Proxy, 308, 457, 779
 proxy client, 718
 prywatna operacja, 405
 prywatny atrybut, 405
 przebiegi, 674
 przechowywanie danych, 309, 311, 339
 przeczucowanie błędów, 512
 predefiniowanie celów projektu, 724
 przeglądy, 495
 przeglądy klienckie, 117, 131, 149
 przeglądy modelu analitycznego, 235
 przeglądy partnerskie, 117, 132
 przeglądy post mortem, 134
 przeglądy projektu, 117, 132, 666
 przeglądy przebiegów, 677
 przeglądy statusowe, 133
 przejście między stanami, 98
 przejście wewnętrzne, 100
 przenośność, 162
 przepływ sterowania, 304, 319, 341
 przepływ zdarzeń, 80, 86
 przepływy pracy, 704
 przepustowość, 162
 przestrzeń robocza, 606
 przewidywalne zdarzenia komunikacyjne, 128
 przydział programistów do projektu, 664
 przydzielanie obiektów i podsystemów do węzłów, 307
 przydzielanie odpowiedzialności, 239, 330, 389, 431
 przydzielanie ról technicznych, 661
 przypadki testowe, 499, 503, 540
 przypadki użycia, 51, 79, 81, 171
 aktorzy, 80
 doskonalenie, 173
 identyfikacja, 171
 nazwa, 80
 przepływ zdarzeń, 80
 przypadki użycia dla sytuacji wyjątkowych, 346
 warunki końcowe, 80
 warunki wstępne, 80
 wymagania jakościowe, 80
 zależności komunikacyjne, 82
 przypisywanie ról menedżerskich, 661
 przyrosty, 608
 przystosowywanie frameworków aplikacyjnych, 381
 pseudo-client, 718
 pseudowymagania, 163
 public, 405
 publiczna klasa, 405
 publiczny atrybut, 405
 Pull Up Constructor Body, 450

Pull Up Field, 450
 Pull Up Method, 450
 Purvis M., 591

Q

QOC, 562, 564, 590, 592
 Questions, Options, and Criteria, 562, 564
 QuestMap, 592

R

racjonalizacja, 42, 55, 549, 551, 579
 aktywności, 567
 argumentacja, 552
 argumenty, 559, 570
 aspekty kierownicze, 585
 bieżące kolekcjonowanie racjonalizacji, 553
 CTC, 568
 Decision Representation Language, 564
 decyzje, 552
 definiowanie problemów, 556
 dokumentowanie, 585
 DRL, 564
 eksploracja przestrzeni rozwiązań, 557
 elementy działania, 561
 ewentualności, 551
 heurystyki komunikowania, 588
 IBIS, 563
 implementacja rozstrzygnięć, 561
 Issue-Based Information System, 563
 kolapsacja przestrzeni rozwiązań, 560
 kolekcjonowanie asynchroniczne, 577
 kolekcjonowanie racjonalizacji, 553, 562, 569
 konceptje, 554
 kryteria, 552, 559, 570
 modele racjonalizacji, 585
 modelowanie zagadnień, 589
 negocjowanie zagadnień, 589
 NFR, 565
 propozycja, 557, 570
 protokolant, 587
 przypisywanie odpowiedzialności, 587
 QOC, 564
 Questions, Options, and Criteria, 564
 redaktor racjonalizacji, 586, 587
 rekonstruowanie racjonalizacji, 553, 582
 rozstrzygnięcie, 560
 strategie rozwiązywania konfliktów, 590
 wartościowanie elementów przestrzeni rozwiązań, 559
 weryfikator racjonalizacji, 587
 zagadnienie, 551, 556
 zintegrowanie racjonalizacji z modelami systemu, 553

racjonalizowanie zmian, 506
RAD, 188, 206, 238, 534, 748
Raport zdarzeń testowych, 533, 535
raportowanie, 120
raportowanie statusu, 601
Rational Unified Process, 709
rationale, 549
Ray W. H., 740
RCS, 610, 633
rdzenne przypadki użycia, 703
realistyczna specyfikacja, 165
realistyczność modelu, 236, 327
realizacja celów projektowych, 301, 306
 aktywności, 306
realizacja projektu, 116
realizacja skojarzeń, 448
redaktor dokumentacji, 123, 239, 331
redaktor racjonalizacji, 586, 587
redukcja krotności, 92
redukcja obiektów do atrybutów, 456
redukcja złożoności modelu, 82, 84
redundancja, 302
refaktoryzacja kodu, 448, 450, 485
 wchłonięcie klasy, 457
 wciągnięcie metod, 450, 451
 wciągnięcie pola, 450
 wciągnięcie treści konstruktora, 450, 451
refleksje post mortem, 673
Rehabilitation Act, 163
reengineering, 166
reinżynieria, 166
rekonstruowanie racjonalizacji, 553, 582
rekurencyjna reprezentacja hierarchii, 776
relacja dziedziczenia, 84
relacja komunikacji, 82
relacja między aktorami a przypadkami użycia, 176
relacja ogół-szczegół, 217
relacja rozszerzania, 83, 180
relacja rozszerzania między przypadkami użycia, 176
relacja zawierania, 82, 180
relacja zawierania między przypadkami użycia, 177
relacje, 469
relacyjne bazy danych, 311, 469
release, 602
release candidate, 525
reliability, 494
Remote Procedure Call, 283
ReportEmergency, 81, 219
repository, 603
repozytorium, 279, 603, 606
REQuest, 586, 592
Requirements Analysis Document, 188, 238
RequisitePro, 187
resolution, 560
responsibility-driven design, 166
restrukturyzacja modelu, 353, 357

reverse engineering, 427, 452, 706
review, 495
Revision Control System, 610, 633
Ritchie D. M., 287
Rittel H., 562, 592
Robson D. J., 516
Rochkind M. J., 632
role, 44, 45, 89, 640, 646
role menedżerskie, 661
role międzyfunkcyjne, 123
role programistyczne, 123
role techniczne, 661
role w realizacji projektu, 122
role zarządcze, 122
round-trip engineering, 706
Rowen R. B., 709
Royce W., 532, 677, 687, 699, 725
rozbudowywanie infrastruktury komunikacyjnej, 118
rozmowy telefoniczne, 140
rozpoczynanie pracy systemu, 345
rozpoznawanie kwalifikacji, 643
rozproszenie geograficzne, 718
rozstrzygnięcie, 560
rozszerzalność, 363
rozszerzenia diagramów, 106
rozwiązywanie problemów, 21, 38, 40, 117, 137, 138
rozwidlenia, 102
różnice między zawieraniem a rozszerzaniem, 178
RPC, 283
Rubin E., 212
Rubin J., 509, 542
rugby, 737
Rumbaugh J., 107, 108, 162, 215, 455, 485
rundy, 701
ryzyko, 669

S

samoloty pasażerskie, 598
sandwich testing, 521, 522
SatWatch, 161, 163, 187
sawtooth model, 709
scalanie gałęzi, 620
SCCS, 632
scenariusze, 85, 130, 171
 instancje aktorów uczestniczących, 86
 nazwy, 85
 przepływ zdarzeń, 86
 scenariusz bieżący, 169
 scenariusz ewaluacyjny, 170
 scenariusz treningowy, 170
 scenariusz wizjonerski, 169
scenopis rysunkowy, 509
schedule, 646
Scheduler, 540

- schemat bazy danych, 311
- schemat identyfikowania wersji, 606
- schemat organizacyjny, 640
- schematy, 469
- Schlichter J., 151
- Schwaber K., 674, 677, 725
- sciences of the artificial, 39
- SCMP, 627
- Scrum, 674, 677, 725, 737, 739
 - dni robocze, 675
 - kontrola projektu, 675
 - kończenie projektu, 677
 - mistrz, 674, 675
 - organizowanie projektu, 675
 - planowanie projektu, 674
 - potencjalnie dostarczalne przyrosty produktu, 674
 - przebiegi, 674
 - przeglądy przebiegów, 677
 - role, 675
 - schemat ogólny, 678
 - właściciel produktu, 675
 - wykresy wygaszania, 675
 - zebranie planistyczne, 674
 - zebranie startowe, 674
 - zebranie statusowe, 675
 - zespół, 675
- Scrum master, 675
- Scrum of Scrums, 766
- Scrum team, 675
- SDD, 328, 534, 748
- sekcja 508 ustawy Rehabilitation Act, 163
- sekwencje abstrakcyjne, 96
- sekwencyjne modele ukierunkowane
 - na aktywności, 699
 - model kaskadowy, 699
 - V-Model, 700
- select(), 414
- self, 408
- sequence, 413
- Service Oriented Architecture, 348
- serwer, 283
- serwer prezentacji, 286
- sesja JAD, 185
- sesje treningowe, 118
- set, 413
- sformułowanie problemu, 129
- shadow testing, 531
- Shaw M., 296
- Siegel S. G., 602
- Siewiorek D. P., 348, 543
- Simon H., 39
- size, 414
- skalowalność, 335
- skill matrix, 650
- skojarzenia jeden do jednego, 459
- skojarzenia jeden na wiele, 461
- skojarzenia kwalifikowane, 92, 463
- skojarzenia niekwalifikowane, 92
- skojarzenia wiele na wiele, 461
- skojarzenia wielokrotne, 455
- skojarzenie, 88, 229, 458
 - atrybuty, 229
- skryte powiązania, 472
- slack time, 649
- Smalltalk-80, 258
- smoke test, 632
- SNET, 650
- SNLT, 650
- SOA, 348
- socket, 270
- Software Configuration Management Plan, 627
- software library, 603
- software life cycle models, 685
- software life cycles, 637
- Software Project Management Plan, 642
- software reliability, 494
- solidność, 162
- sort, 287
- Source Code Control System, 632
- specjalista od zastosowań, 124
- specjalista z zakresu dziedziny aplikacyjnej, 124
- specjalista z zakresu dziedziny realizacyjnej, 124
- specjalizacja, 216, 217
- Specyfikacja przypadków testowych, 533, 534, 535
- specyfikacja usług, 533
- specyfikacja wymagań, 159, 164
 - identyfikowalność, 165
 - jednoznaczność, 164
 - kompletność, 164
 - poprawność, 164
 - realizm, 165
 - spójność, 164
 - weryfikowalność, 165
- specyfikowanie interfejsów, 357, 399, 401, 403
 - aktywności, 402, 416
 - ARENA, 433
 - brakujące atrybuty i operacje, 402
 - definiowanie sygnatur, 402
 - definiowanie widzialności, 402
 - dziedziczenie kontraktów, 424
 - ekstender klasy, 403
 - identyfikacja brakujących atrybutów, 417
 - identyfikacja brakujących operacji, 417, 434
 - implementator klasy, 403
 - język OCL, 407
 - kolekcje OCL, 411
 - kontrakty, 406
 - kwantyfikatory OCL, 415
 - specyfikowanie kontraktów, 402, 435, 438
 - specyfikowanie niezmienników, 421
 - specyfikowanie sygnatur, 418
 - specyfikowanie typów, 418

- specyfikowanie warunków końcowych, 419
- specyfikowanie warunków wstępnych, 419
- specyfikowanie widzialności, 418
- sygnatury, 403, 404
- typy, 403
- użytkownik klasy, 403
- widzialność, 403, 405
- zarządzanie projektowaniem obiektów, 425
- specyfikowanie niezmienników, 421
- SPMP, 642, 652, 654, 655, 690
- spoistość, 267, 271, 272
- spotkania osobiste, 140, 141
- spójność modelu, 235, 327
- spójność specyfikacji, 164
- sprint backlog, 674
- sprzęt, 43
- sprzężenie, 267, 271
- SQL, 471
- staged build, 631
- stan obiektu, 98
- standard cykli życiowych, 688
- stany realizacji projektu, 116
- Start no earlier than, 650
- Start no later than, 650
- statecharts, 98
- statyczna polityka dostępu, 316
- stereotypy, 106, 216
 - «boundary», 106
 - «control», 106, 216
 - «entity», 106
 - «extent», 106
 - «include», 106
 - «invariant», 408
 - «postcondition», 408
 - «precondition», 408
 - «sut», 540
 - «testCase», 540
 - «testComponent», 540
 - «testContext», 540
 - «testObjective», 540
- sterowanie proceduralne, 319
- sterowanie wielowątkowe, 320
- sterowanie zdarzeniowe, 319
- sterownik testowy, 499, 505
- storyboard, 509
- Strategia, 373, 518, 780
 - delegowanie, 375
 - dziedziczenie, 375
- strategia przechowywania danych, 311
- strategia przechowywania obiektów trwałych, 340
- strategia zrównoleglonych projektów, 624
- strategie integrowania pionowego, 525
- strategie integrowania poziomego, 521
- strategie minimalizowania ryzyka, 672
- strategie rozwiązywania konfliktów, 590
- Strategy, 780
- Straus D., 151
- Streeter L. A., 151
- struktura łącznikowa, 121, 122
- struktura podziału pracy, 640, 646, 648, 657
- struktura raportowania organizacji, 120
- strukturalizowane wywiady, 140
- STS-51L, 638
- style architektoniczne, 279
- subklasy, 73, 93, 360
- subtelna kontrola, 741
- subtle control, 741
- Suchman L. A., 719, 720
- suchy przebieg, 150
- superklasy, 73, 93, 360
- supporting workflows, 704
- Sutherland J., 677
- Swarz R. S., 348, 543
- swimlanes, 103
- Swing, 258, 276, 378
- Switzer J., 151
- SYBIL, 564
- sygnały dymne, 138
- sygnatury, 404
- synchroniczna komunikacja grupowa, 140
- synchroniczny mechanizm komunikacyjny, 138, 140
- syndrom „to nie nasz wynalazek”, 387
- syndrom Y2K, 551
- system, 43, 46, 69
- system ARENA, 57, 190, 245
- system centralnego sterowania ruchem, 555
- system chaordyczny, 766
- system CVS, 608
- System Design Document, 328
- system informatyczny, 35, 37
- system komputerowy promu kosmicznego, 302
- system plików, 91
- system samoorganizujący się, 735
- system sztuczny, 39
- system tablicowy, 280
- system under test, 540
- system wysokiej jakości, 382
- sytuacje wyjątkowe, 346
- szablon dokumentu RAD, 206
- szkolenia, 662
- szybkość projektu, 733
- szyfrowanie, 318
- szyfrowanie informacji, 313

Ś

- ścieżka krytyczna, 649
- ścieżka krytyczna PERT, 127
- ściśle dziedziczenie, 364
- śledzenie problemów, 506
- średni czas pracy między awariami, 162, 668

środowisko projektu, 716, 717, 744
 czas trwania projektu, 719
 częstotliwość zmian, 719
 doświadczenie uczestników, 717
 klimat technologiczny, 718
 kontakt z użytkownikami, 718
 rozproszenie geograficzne, 718
 typ klienta, 717
 środowisko testowe, 539

T

T.H.E., 296
 tabela pośrednicząca, 472
 tabela skojarzeniowa, 472
 tabele, 469
 Takeuchi H., 677, 737
 taksonomiczna identyfikacja ryzyka, 669
 task, 640, 646
 task model, 649
 Taxonomy-Based Risk Identification, 669
 Taylor F., 719, 740
 TCS, 535
 teoria umysłu pojemnika, 41
 Test Case Specification, 535
 test driver, 499
 test harness, 539
 Test Incident Report, 535
 Test Plan, 497
 test stub, 499
 TestCase, 503, 538
 tester, 123, 124
 testing, 494
 testowanie, 54, 491, 494, 496, 553
 aktywności, 497, 506
 automatyzacja, 538
 awaria, 499, 500
 błędny stan, 499, 500
 dokumentowanie, 532
 inspekcja komponentu, 506, 507
 JUnit, 538
 koncepty, 498
 namiatka testowa, 499, 505
 Plan testów, 533, 534, 536
 planowanie testów, 497, 532
 poprawki, 499, 505
 profile, 539
 Przydzielanie odpowiedzialności, 536
 przypadki testowe, 503
 Raport podsumowujący, 533
 Raport sumaryczny, 535
 Raport zdarzeń testowych, 533, 535
 Specyfikacja przypadków testowych, 533, 534, 535
 sterownik testowy, 499, 505
 środowisko testowe, 539
 TCS, 535

TestCase, 503, 538
 testowanie akceptacyjne, 497, 526, 644
 testowanie bazujące na modelach, 539
 testowanie funkcjonalności, 497, 526
 testowanie graniczne, 511
 testowanie hurtowe, 521
 testowanie instalacji, 497, 526
 testowanie ręczne, 538
 testowanie kanapkowe, 521, 522
 testowanie modułów, 510
 testowanie pilotażowe, 526, 530
 testowanie polimorfizmu, 517
 testowanie regresyjne, 506, 537
 testowanie rywala, 530
 testowanie sterowane stanami, 516
 testowanie strukturalne, 497
 testowanie systemu, 497, 506, 526
 testowanie ścieżek, 512
 testowanie użyteczności, 497, 506, 508
 testowanie w cieniu, 530
 testowanie wahadłowców, 492
 testowanie wstępujące, 521, 522
 testowanie wydajności, 497, 526, 528
 testowanie wymaganiowe, 526
 testowanie zstępujące, 521, 522, 523
 testowany komponent, 498
 testy białoskrzynkowe, 504
 testy czarnoskrzynkowe, 504
 TIR, 535
 U2TP, 539
 uprząż testowa, 539
 usterki, 499, 500
 zarządzanie testowaniem, 531
 testowanie integracyjne, 497, 506, 519, 520
 strategie integrowania pionowego, 525
 strategie integrowania poziomego, 521
 testowanie jednostkowe, 497, 506, 510
 graf przepływów, 513
 pokrycie, 510
 reprezentatywność, 511
 rozłączność, 511
 testowanie graniczne, 511
 testowanie polimorfizmu, 517
 testowanie ścieżek, 512
 testowe klasy równoważności, 510
 testowany komponent, 498
 testowany system, 540
 testowe klasy równoważności, 510
 testy akceptacyjne, 530
 testy bezpieczeństwa, 529
 testy białoskrzynkowe, 504
 testy czarnoskrzynkowe, 504
 testy dubletowe, 520
 testy dwójkowe, 520
 testy instalacyjne, 531
 testy kwadrupletowe, 520

testy na dym, 632
testy objętościowe, 529
testy odtwarzania, 530
testy pilotażowe, 530
testy polowe, 530
testy produktu, 509
testy prototypu, 509
testy przeciążeniowe, 529
testy regresyjne, 537
testy scenariusza, 508
testy tripletowe, 520
testy uwarunkowań czasowych, 529
testy wzorcowe, 530
TEX, 607
tępe zliczanie, 684
ThingLab, 440
Thompson K., 287
Tichy W., 633, 771
TicketDistributor, 44
 aktywności, 47
 dekompozycja systemu, 53
 model dynamiczny systemu, 51, 52
 model obiektowy systemu, 52
 produkty, 46
 przypadki użycia, 51
 PurchaseOneWayTicket, 51
 role, 45
 zadania, 47
 zasoby, 47
TIR, 535
tolerowanie usterek, 495
Tonies C. C., 687
top-down testing, 521, 522
tory pływakie, 103
Total Quality Management, 740
TP, 497
TQM, 740
transformacja modelu, 447, 448, 449
 dokumentowanie, 475
 zarządzanie transformacjami, 475
 zasady, 453
trwałe dane, 303, 309
Turner C. D., 516
tworzenie oprogramowania, 54
tworzenie systemu, 692
typ klienta, 717
typed languages, 72
typy atrybutów, 404
typy danych, 72

U

U2TP, 539, 540
uczestnicy, 43, 44, 116, 640
udział w zebraniach zespołu, 147

UML, 21, 43, 63, 64, 65, 78, 607
 diagramy aktywności, 68, 79, 101
 diagramy interakcji, 64, 67, 79, 95
 diagramy klas, 64, 65, 78, 86
 diagramy przypadków użycia, 64, 65, 78, 79
 diagramy sekwencji, 67
 diagramy stanów, 67, 79, 98
 diagramy wdrażania, 304
 dziedziczenie, 93
 klasy, 75
 komponenty fizyczne, 269
 komponenty logiczne, 269
 krotność, 90
 maszyna stanu, 98
 model dynamiczny, 64
 model funkcjonalny, 64
 model obiektowy, 64
 modelowanie, 63
 notacja, 71
 notatki, 105
 obiekty, 74
 ograniczenia, 106
 organizacja diagramów, 104
 pakiety, 104
 rozszerzenia diagramów, 106
 skojarzenia, 88
 stereotypy, 106
 tory pływakie, 103
 U2TP, 539
 widzialność elementu, 406
UML 2 Testing Profile, 539
umowa projektu, 642, 664
Unified Modeling Language, 48
Unified Process, 162, 532, 703, 725
Unified Software Development Process, 49, 166, 703
unikanie usterek, 495
union(), 414
unit testing, 510
Universal Modeling Language, 21, 43, 63
UNIX, 287
uprząż testowa, 539
URPS, 163
uruchamianie na sucho, 150
uruchamianie projektu, 643
usługi, 267, 270, 321, 343
usterki, 494, 499, 500
usterki maszyny wirtualnej, 502
usterki wykryte podczas inspekcji kodu, 125
usterki wykryte podczas testowania, 125
utrzymanie systemu, 54
uwierzytelnianie, 313, 317
uzgodnienie modelu analitycznego z klientem, 243
uzgodnienie umowy projektu, 642
użyteczność, 158, 162, 568
użytkownik, 124, 239
użytkownik klasy, 403

V

vaporware, 37
 variants, 602
 verdict, 540
 version, 602
 Vlissides J., 283
 V-model, 687, 699, 700
 VMS, 632
 Vodde B., 766

W

waga lekka, 737
 walkthrough, 495
 warianty, 602, 604, 623
 Warmer J., 407
 warstwy, 275
 wartość wypracowana, 667
 warunki graniczne, 304, 323, 345
 warunki końcowe, 80, 406, 419
 warunki wstępne, 80, 406, 419
 waterfall model, 699
 WBS, 646, 648, 657
 wchłonięcie klasy, 457
 wciągnięcie metod, 450, 451
 wciągnięcie pola, 450
 wciągnięcie treści konstruktora, 450, 451
 Weber C. V., 723
 WebifyWatch, 167
 WebObjects, 384
 Weinand A., 382
 Weinberg G. M., 732
 wersje, 602, 604
 weryfikacja modelu analitycznego, 254
 weryfikacja projektu systemu, 326
 weryfikator, 240, 331
 weryfikator racjonalizacji, 587
 weryfikowalna specyfikacja, 165
 weryfikowalne cechy specyfikacji, 164
 wędrówka po kodzie, 132, 495
 węzły kontrolne, 101
 whitebox tests, 504
 wideokonferencje, 140
 widok, 69, 281, 368
 widzialność, 405
 wiele na wiele, 90
 wielokrotne wykorzystywanie frameworków, 721
 wielokrotne wykorzystywanie rozwiązań
 wzorcowych, 353, 359
 ARENA, 390
 biblioteki klas, 383
 delegowanie, 363
 dokumentowanie wykorzystania gotowych
 rozwiązań, 388

dziedziczenie implementacyjne, 360
 dziedziczenie specyfikacyjne, 360
 frameworki aplikacyjne, 381
 obiekty aplikacyjne, 359
 obiekty realizacyjne, 359
 przydzielanie odpowiedzialności, 389
 wzorce projektowe, 364, 383
 zarządzanie wykorzystywaniem gotowych
 rozwiązań, 386
 zasada zastępowania Liskov, 364
 wielowątkowość, 321
 wielozbiory, 411, 413
 Większość ma rację, 591
 Wigand R. T., 679
 WinWin, 590, 592
 Władca Pierścieni, 446
 Właściciel ma ostatnie słowo, 591
 właściciel produktu, 675
 Work Breakdown Structure, 640, 646
 work packages, 640, 646
 work product, 640
 work units, 640
 workflows, 704
 wpadki produkcyjne, 354
 wspieralność, 162
 wspinaczka wysokogórska, 716
 współdzielenie kodu między warianty, 624
 wstępna deklaracja problemu, 190
 wstępne definiowanie architektury systemu, 642
 wstępne zaplanowanie zarządzania projektem, 642
 Wstępny Plan Zarządzania Projektem
 Programistycznym, 642
 WWW, 140
 wybór konfiguracji sprzętowej, 306
 wybór liczebności zespołów, 662
 wybór platformy, 306
 wybór strategii przechowywania danych, 311
 wybór strategii przechowywania obiektów trwałych,
 340
 wydajność, 162
 wykaz zaległości produktu, 674
 wykaz zaległości przebiegu, 674
 wykorzystywanie gotowych rozwiązań, 353, 386
 wykres Gantta, 126
 wykresy wygaszania, 676
 wykrywanie usterek, 495
 wymagania, 157
 wymagania dotyczące interfejsu, 163
 wymagania funkcyjne, 48, 161, 191
 wymagania implementacyjne, 163
 wymagania jakościowe, 80, 163
 wymagania operacyjne, 163
 wymagania pakietowe, 163
 wymagania pozafunkcyjne, 48, 162, 163, 182, 191,
 205
 wymagania prawne, 163

wyraziste kamienie milowe, 665
 wyrównywanie braku kwalifikacji, 662
 wysokopoziomowy projekt systemu, 654
 wystąpienia, 72
 wysyłanie komunikatu, 75, 95
 wywiady strukturalne, 141
 wzorce projektowe, 359, 364, 383, 402, 721, 771
 Abstract Factory, 772
 Adapter, 365, 366, 371, 773
 Bridge, 774
 Command, 775
 Composite, 776
 delegowanie, 364
 dziedziczenie, 364
 Fabryka abstrakcyjna, 376, 390, 772
 Facade, 777
 Fasada, 295, 777
 hermetyzacja algorytmów, 780
 hermetyzacja hierarchii, 378
 hermetyzacja kontekstu, 373
 hermetyzacja kosztownych obiektów, 779
 hermetyzacja platformy, 376, 772
 hermetyzacja podsystemów, 777
 hermetyzacja przechowywania danych, 368
 hermetyzacja przepływu sterowania, 377, 775
 heurystyki pomocne w wyborze wzorców
 projektowych, 781
 heurystyki wyboru wzorców projektowych, 379
 Kompozyt, 378, 776
 Most, 368, 774
 Observer, 778
 Obserwator, 283, 393, 778
 oddzielenie encji od widoków, 778
 otoczka dla starszego kodu, 773
 podmiana implementacji, 774
 Polecenie, 377, 392, 775
 Proxy, 308, 457, 779
 rekurencyjna reprezentacja hierarchii, 776
 Strategia, 373, 518, 780
 Strategy, 780
 wybór wzorców projektowych, 367

X

X11, 276, 278
 XP, 725, 731, 737

Y

Yahoo Groups, 145
 Yourdon E., 107, 296

Z

zachowanie obiektu, 67
 zachowanie systemu w kategoriach aktywności, 69
 zachowanie zewnętrzne, 79
 zadanie, 43, 44, 47, 116, 124, 640, 646
 pakiet, 126
 specyfikacja prac, 126
 zagadnienia, 551, 556, 707
 zagadnienia wynikowe, 557
 zagadnieniowy model cyklu życiowego, 707
 zagnieżdżone maszyny stanów, 100
 zagnieżdżony stan, 100
 założenia kontroli dostępu, 312, 340
 zamknięta architektura warstwowa, 277
 zamknięte zagadnienie, 707
 zaporą sieciową, 315
 zarządzanie analizą wymagań, 237
 zarządzanie cyklem życiowym, 683
 zarządzanie danymi, 303
 zarządzanie emisjami, 616
 zarządzanie gałęziami, 619
 zarządzanie generowaniem binarów, 601
 zarządzanie identyfikowalnością, 187
 zarządzanie jakością oprogramowania, 691
 zarządzanie konfiguracją, 56, 189, 597, 600
 agregat CM, 602, 603
 agregat zarządzania konfiguracją, 602
 aktywności, 597, 600, 601, 611
 aspekty kierownicze, 627
 audyt, 601
 biblioteka, 603, 606
 ClearCase, 610, 611
 CVS, 610, 611
 dokumentowanie, 627
 elementy konfiguracji, 602, 603
 emisja, 602, 605
 ewolucja elementu konfiguracji, 608
 gałęzie, 607
 heurystyki wspomagające zarządzanie
 gałęziami, 621
 identyfikacja agregatów CM, 613
 identyfikacja elementów konfiguracji, 600, 613
 identyfikatory wersji, 606
 integracja ciągła, 630
 katalog główny, 603
 konceptcje, 602
 konfiguracja, 604
 konflikty, 620, 623
 kontrola zmian, 601
 linia bazowa, 604
 narzędzia wspomagające zarządzanie
 konfiguracją, 610
 numery wersji, 606
 Perforce, 610, 611

- zarządzanie konfiguracją
 - Plan Zarządzania Konfiguracją Oprogramowania, 627, 628
 - planowanie aktywności, 629
 - promocja, 602, 605
 - przestrzeń robocza, 606
 - przypisywanie odpowiedzialności, 628
 - raportowanie statusu, 601
 - RCS, 610
 - repozytorium, 603, 606
 - role, 629
 - scalanie gałęzi, 620
 - schematy identyfikowania wersji, 606
 - SCMP, 627
 - testowanie, 630
 - warianty, 602, 604
 - wersje, 602, 604
 - współdzielenie kodu między warianty, 624
 - zarządzanie emisjami, 616
 - zarządzanie gałęziami, 619
 - zarządzanie generowaniem binariów, 601
 - zarządzanie procesem, 601
 - zarządzanie promocjami, 615, 630
 - zarządzanie propozycjami zmian, 626
 - zarządzanie wariantami, 623
 - zarządzanie zmianami, 626
 - zbiory zmian, 608
 - zmiany, 608
 - żądanie zmiany, 602, 604
- zarządzanie procesem, 601
- zarządzanie projektem, 54, 56, 552, 637, 639, 690
 - aktywności, 640, 641, 645, 646
 - aktywności klasycznego zarządzania projektem, 653
 - elementy działania, 646
 - faza definicyjna, 642
 - faza koncepcyjna, 642
 - faza końcowa, 644
 - faza startowa, 643
 - faza ustalona, 643
 - funkcje projektowe, 646
 - harmonogram, 640, 646
 - jednostki pracy, 640
 - koncepcje, 646
 - kontrola projektu, 641, 665
 - kończenie projektu, 641, 671
 - macierz kwalifikacji, 650
 - model zadań, 640, 646, 649
 - modele menedżerskie, 640
 - organizowanie projektu, 641, 659
 - pakiety pracy, 640, 646
 - Plan Zarządzania Projektem, 651
 - planowanie projektu, 641, 654
 - praca, 640
 - produkt, 640, 646
 - produkt finalny, 640, 648
 - produkt wewnętrzny, 648
 - role, 646
 - SPMP, 642, 652
 - struktura podziału pracy, 640, 646, 648
 - uzgodnienie umowy projektu, 642
 - WBS, 648
 - Wstępny Plan Zarządzania Projektem Programistycznym, 642
 - wynik, 640
 - zadanie, 640, 646
 - zasoby, 640
- zarządzanie projektowaniem obiektów, 425
 - dokumentowanie projektowania obiektów, 425
 - przydzielanie odpowiedzialności, 431
 - wykorzystywanie kontraktów w analizie wymagań, 432
- zarządzanie projektowaniem systemu, 328
- zarządzanie promocjami, 615
- zarządzanie propozycjami zmian, 626
- zarządzanie racjonalizacją, 55, 549
 - aspekty kierownicze, 585
 - racjonalizacja, 551
- zarządzanie ryzykiem, 644, 669
- zarządzanie scentralizowane architektonicznie, 655
- zarządzanie testowaniem, 531
- zarządzanie transformacjami, 475
- zarządzanie tworzeniem oprogramowania, 54
 - cykl życiowy oprogramowania, 56
 - kommunikacja, 55
 - zarządzanie konfiguracją oprogramowania, 56
 - zarządzanie projektem, 56
 - zarządzanie racjonalizacją, 55
- zarządzanie wariantami, 623
- zarządzanie wykorzystywaniem gotowych rozwiązań, 386
- zarządzanie zbieraniem wymagań, 183
- zarządzanie zmianami, 626
- zasada zastępowania Liskov, 364, 394
- zasoby, 43, 47, 640
- zastoje, 284
- zbieranie wymagań, 50, 65, 157, 159, 552
 - aktywności, 160, 166
 - analiza wymagań, 159
 - dokument analizy wymagań, 188
 - dokumentowanie, 188
 - doskonalenie przypadków użycia, 160, 173
 - identyfikacja aktorów, 160, 167
 - identyfikacja przypadków użycia, 160, 171
 - identyfikacja relacji między aktorami a przypadkami użycia, 176
 - identyfikacja relacji między przypadkami użycia, 160
 - identyfikacja scenariuszy, 160, 169
 - identyfikacja wymagań pozafunkcyjnych, 160, 182
 - inżynieria interfejsu, 165, 166
 - inżynieria pierwotna, 165

- inżynieria wtórna, 165
 - JAD, 185
 - jednoznaczność, 164
 - kompletność, 164
 - kommunikacja, 158
 - konceptje, 161
 - negocjowanie specyfikacji z klientem, 185
 - początkowa identyfikacja obiektów modelu
 analitycznego, 179
 - poprawność, 164
 - RAD, 188
 - specyfikacja wymagań, 159
 - spójność, 164
 - system ARENA, 190
 - weryfikowalne cechy specyfikacji, 164
 - wymagania funkcyjne, 161
 - wymagania pozafunkcyjne, 162
 - zarządzanie identyfikowalnością, 187
 - zarządzanie zbieraniem wymagań, 183
 - zbiory, 411, 413
 - zbiory zmian, 608
 - zdalne wywoływania procedur, 283
 - zdarzenia, 75, 76
 - zdobycie K2, 714
 - zebrania, 147, 665
 - zebrania planistyczne, 674
 - zebrania początkujące, 118
 - zebrania statusowe, 117, 118, 665, 675
 - zebranie otwierające, 664
 - zebranie startowe, 674
 - zespoły, 119, 663
 - zespoły czteroosobowe, 663
 - zespoły międzyfunkcyjne, 121, 146
 - zespoły podsystemów, 146
 - zespoły trzyosobowe, 663
 - zespół (Scrum), 675
 - zintegrowane środowisko programistyczne, 280
 - zintegrowanie racjonalizacji z modelami systemu, 553
 - złącza, 270
 - złączenia, 102
 - złudzenie optyczne, 212
 - zmiany, 37, 608
 - zmiany w implementacji, 367
 - zmiany w środowisku operacyjnym, 324
 - zmiany w zakresie dziedziny aplikacyjnej, 368
 - zmodyfikowane testowanie kanapkowe, 523
 - znacznikowane komentarze Javadoc, 429
 - zorientowane obiektowo tworzenie
 oprogramowania, 41
 - zorientowanie obiektowe, 40
 - zrównoległone projekty, 623
 - Z-Schematy, 72
 - zwinna realizacja projektu, 673
 - planowanie projektu, 674
 - zwinne metodologie, 725, 737
- ## Ż
- źle zlokalizowane atrybuty, 456
- ## Ż
- żądanie wyjaśnień, 117, 136
 - żądanie zmian, 117, 137, 138, 602, 604

Projektowanie systemów informatycznych to zadanie bardzo skomplikowane. Ogromna liczba zależności, reguł i wyjątków od nich sprawia, że nie jest możliwe podejście do tego zadania ot tak, z marszu. Zbieranie i analiza wymagań, przygotowanie diagramów klas, aktywności, stanów czy interakcji to tylko część etapów, z którymi musi poradzić sobie projektant. Jeżeli nałożyć na to wszystko wzorce projektowe, stajemy przed prawie nierozwiązywalnym zadaniem. Na szczęście — prawie!

Dzięki tej książce dowiesz się, jak sprostać temu karkołomnemu zadaniu! W trakcie lektury poznasz język UML, który wprowadził porządek w tym skomplikowanym procesie, oraz podstawowe koncepcje inżynierii oprogramowania. Nauczysz się zarządzać procesem tworzenia oprogramowania, zbierać oraz analizować wymagania, identyfikować podsystemy, specyfikować interfejsy oraz testować. Znajdziesz tu wszystko na temat zarządzania zmianami. W trakcie lektury sprawdzisz, jak wygląda cykl życia oprogramowania oraz jak zarządzać konfiguracją. Dodatkowo poznasz metodologię działań, które doprowadzą Cię do wyznaczonego celu. Książka ta stanowi obowiązkową pozycję dla każdego projektanta oraz analityka. Jednak programiści również znajdą tu wiele cennych wskazówek!

Niepowodzenia w inżynierii oprogramowania
Podstawowe koncepcje inżynierii oprogramowania
Modelowanie przy użyciu języka UML
Organizacja projektu
Narzędzie do komunikacji grupowej
Proces zbierania wymagań
Identyfikacja aktorów, scenariuszy oraz przypadków użycia
Określanie obiektów modelu analitycznego
Analiza wymagań
Dekompozycja systemu na podsystemy
Identyfikacja celów projektowych
Projektowanie obiektów
Wzorce projektowe
Specyfikowanie interfejsów
Odwzorowywanie modelu na kod
Testowanie
Zarządzanie zmianami i konfiguracją
Cykl życia oprogramowania
Metodologie

**Dobry projekt systemu
to podstawa sukcesu!**

nr katalogowy: 5912

 **Księgarnia Internetowa**
<http://helion.pl>

 **Zamówienia telefoniczne:**
0 801 339900
0 601 339900



Helion

Sprawdź najnowsze promocje:
• <http://helion.promocje>
Kupuj najchejniej czytając:
• <http://helion.poblastaffery>
Zamów informacje o nowościach:
• <http://helion.gdnowosci>

Helion Sp. z o.o.
ul. Rakowiecka 1c, 84-100 Gliwice
tel.: 32 230 98 40
e-mail: helion@helion.pl
<http://helion.pl>

helion.pl
katalog
Internetowa

Cena 129,00 zł

ISBN 978-83-246-2872-8



Informatyka w najlepszym wydaniu