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

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
- Odwzorowywanie modelu na kod
- Testowanie
- Zarządzanie zmianami i konfiguracją
- Cykl życia oprogramowania
- Metodologie

Dobry projekt systemu to podstawa sukcesu!

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

Spis treści

12

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: wspomaganiu 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" przylgnął 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 dygresja[2].

> 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 niezmienne. 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 arytme-

[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 … pięciolatków. I nie można przy tym zrzucać całej winy na pokolenie programistów lat 90.: zdając sobie sprawę ze zbliżającego się *fin de siècle*, skrępowani 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 uwędzenie 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 fatygi ze strony programistów

niż komponenty w miarę dojrzałe. 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 zagadnieńz 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 przemyślanych, 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:

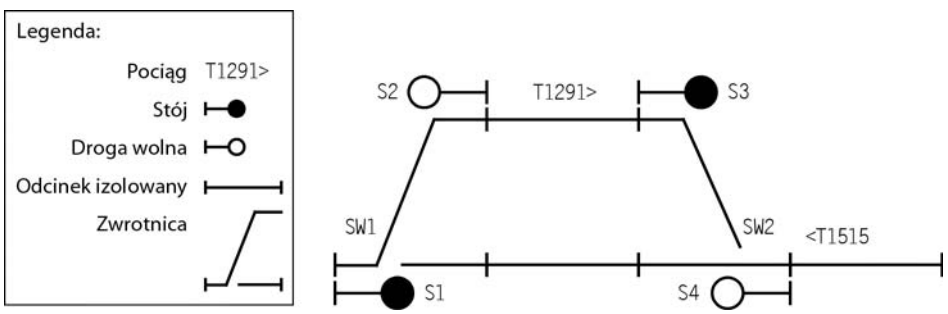
- zagadnienia reprezentujące problemy projektowe i pytania związane z projektem (patrz sekcja 12.3.2),
- ewentualności rozwiązań tych problemów (patrz sekcja 12.3.3),
- argumenty „za" poszczególnymi rozwiązaniami i „przeciw" nim (patrz sekcja 12.3.4),
- decyzje wyboru konkretnych rozwiązań (patrz sekcja 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 przyjrzyjmy 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 urządzeniem 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

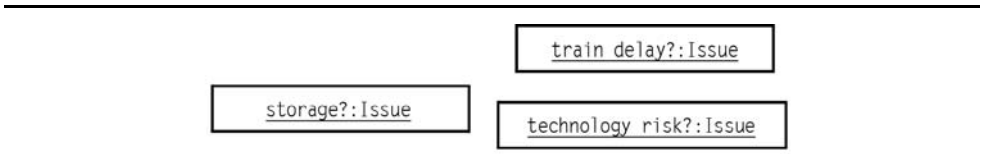
W latach 60. ubiegłego wieku pulpity sterownicze systemu CTC miały postać dedykowanych urządzeń wyposażonych w żarówki wyświetlające stan poszczególnych odcinków izolowanych oraz przyciski służące do przestawiania zwrotnic i ustawiania sygnalizacji. W latach 70. dedykowane pulpity zastąpione zostały 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 zagadnieniezostało 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

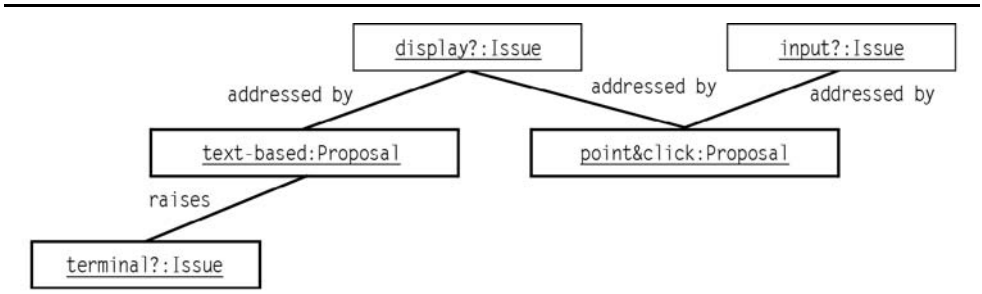
Propozycja jest jedną z odpowiedzi na zagadnienie. *Dyspozytor nie musi być informowany o opóźnieniach pociągów* to propozycja związana z zagadnieniem *Jak szybko powinien być dyspozytor informowany o opóźnieniu pociągu?* Propozycja niekoniecznie musi być dobra czy poprawna w kontekście zagadnienia, 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 zazębiać, tak jak na przykład dla zagadnienia *Jak przechowywać trwałe 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 odśmiecanie* 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 odnośnymi 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 „wskaż 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.

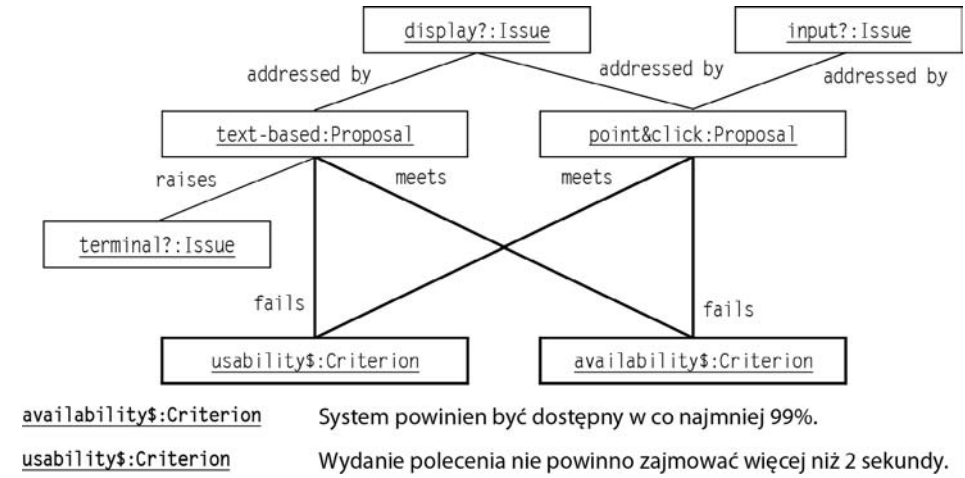
12.3.4. Wartościowanie elementów przestrzeni rozwiązań: kryteria i argumenty

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 kryteriów 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 niefunkcyjne wymagania jak najdłuższej pracy bez awarii i użyteczności (`usablility`) rozumianej w tym przypadku jako minimalizacja czasu wydawania poprawnych poleceń (patrz rysunek 12.5). Kryteria te wyprowadzone zostały wprost z niefunkcyjnych 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 `assesment`, niespełnienie — przez etykietę `fails`

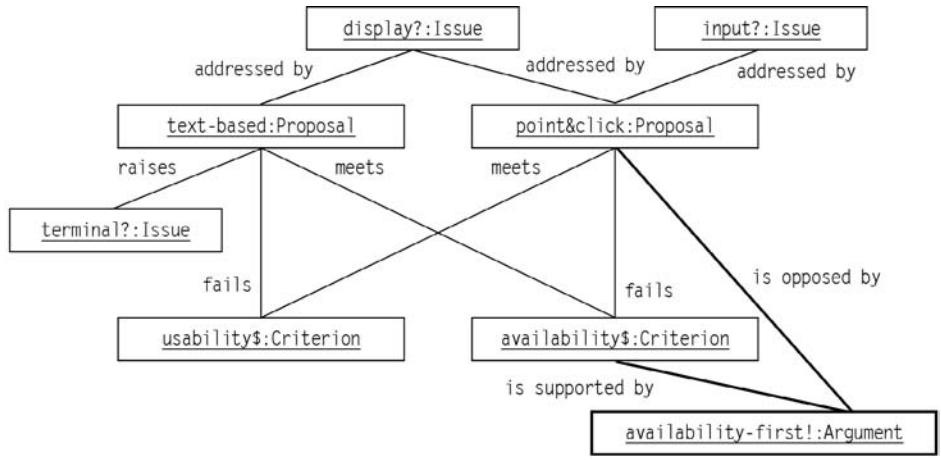
Stając wobec wyboru między dostępnością a użytecznością systemu CTC, zauważamy, że korzyści z większej użyteczności systemu mogą być okupione uszczerbkiem na jego dostępności, co może się przekładać na poważne awarie niwelujące korzyści wynikające z wygodniejszej i sprawniejszej obsługi systemu — jest 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 „wskaż 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)

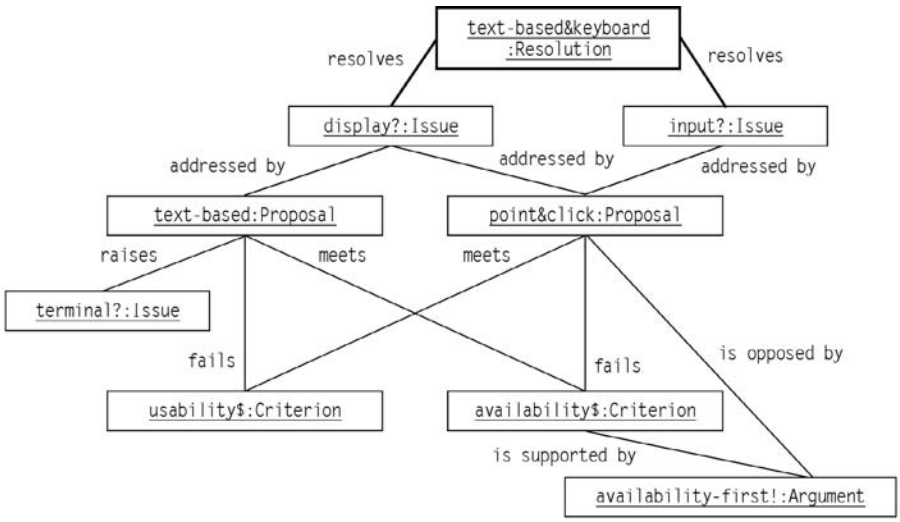
rozwiązaniem, atrybut `status` ma wartość `active`; w przeciwnym razie ma on wartość `obsolete`. Z zamkniętym zagadnieniem skojarzone jest dokładnie jedno rozstrzygnięcie o statusie `active` i dowolna liczba (lub zero) rozstrzygnięć o statusie `obsolete`.

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].

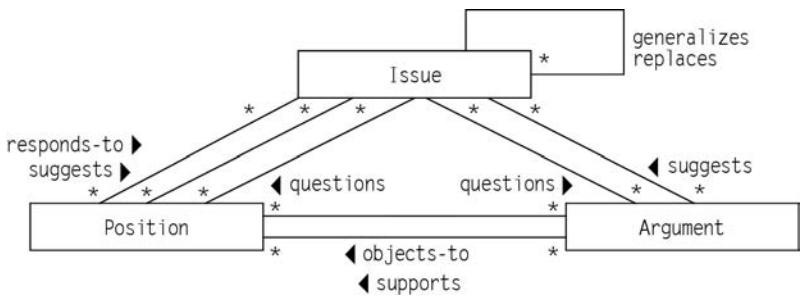
Rysunek 12.8. Przykład implementacji rozstrzygnięcia. Element działania wyróżniony jest pogrubioną ramką

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 klas 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.

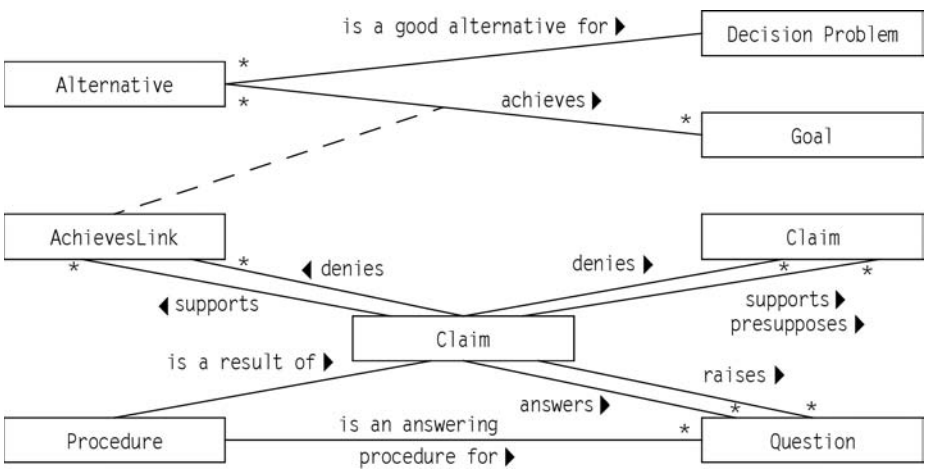


Rysunek 12.9. Model systemu IBIS

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.

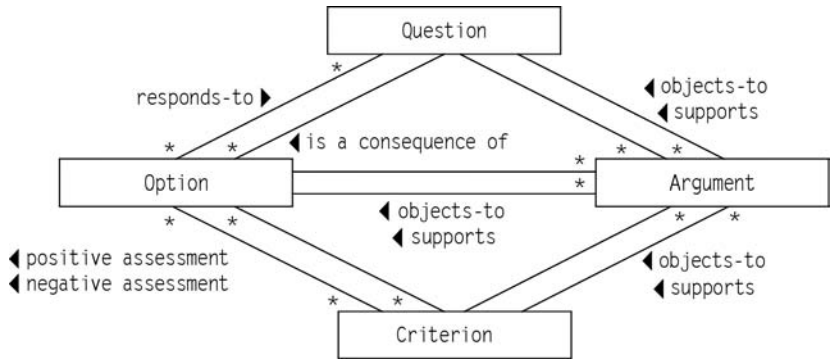


Rysunek 12.10. Model systemu DRL

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. Koncepcyjna 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 zajmiemy się w sekcji 12.4).

NFR

Framework **NFR**[3], 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.*

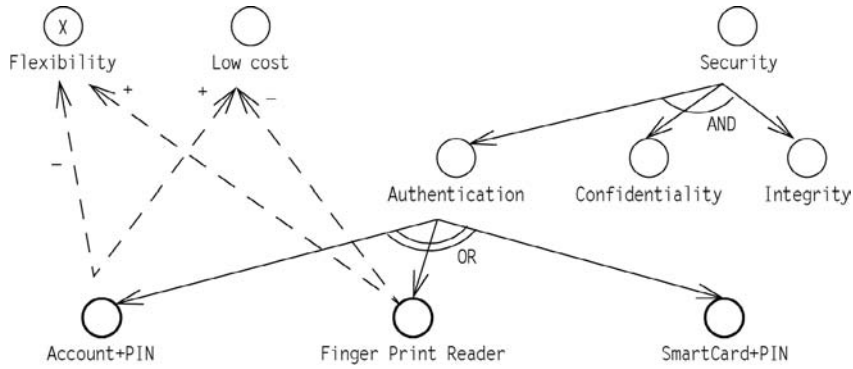
- *Dekompozycja koniunkcyjna* (*AND decomposition*) — by spełniony był określony cel, muszą być spełnione *wszystkie* podcele stanowiące wynik jego dekompozycji.
- *Dekompozycja dysjunkcyjna* (*OR decomposition*) — by spełniony był określony cel, wystarczy spełnienie *co najmniej jednego* spośród podcelów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

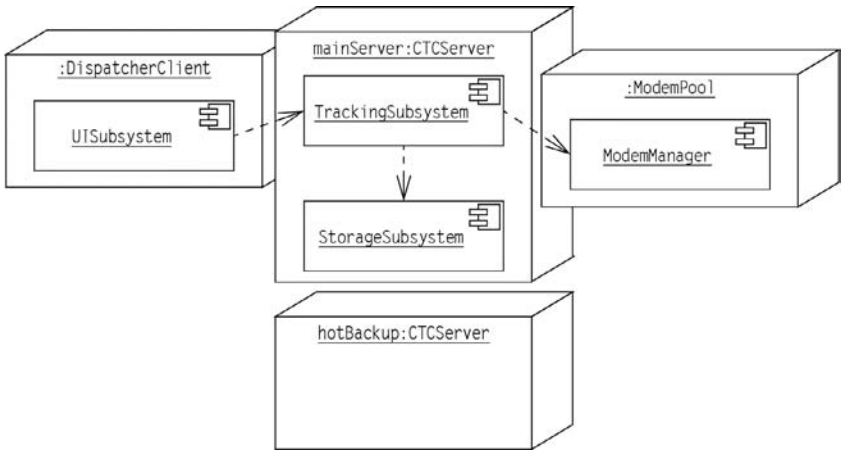
Z tych właśnie względów w tym rozdziale skoncentrujemy się właśnie na etapie projektowania systemu; opisywane techniki kolekcjonowania racjonalizacji wyglądają podobnie na wszystkich innych etapach, począwszy od zbierania wymagań, aż po testy pilotażowe. Poszczególne aktywności opisywać będziemy w kontekście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

Zebrania umożliwiają 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
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 [czas: 15 minut]

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

4. Dyskusja [czas: 35 minut]

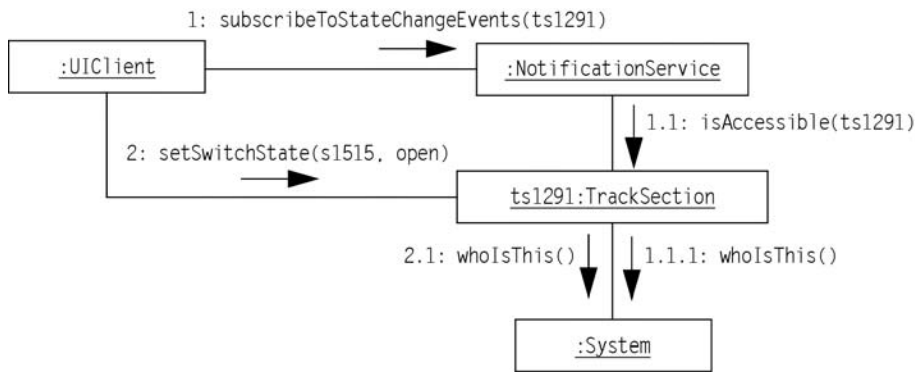
`I[1]`: Czy dyspozytor ma prawo oglądać sekcje dyspozycyjne kontrolowane przez innych dyspozytorów?
`I[2]`: Czy dyspozytor ma prawo wysyłania poleceń do urządzeń znajdujących się w sekcjach dyspozycyjnych kontrolowanych przez innych dyspozytorów?
`I[3]`: W jaki sposób kontrola dostępu powinna być zintegrowana 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.

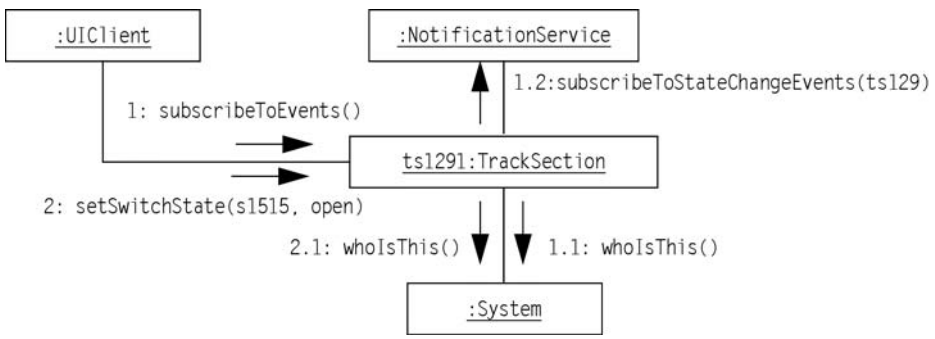


Rysunek 12.14. Propozycja P[1]: Dostęp do klasy TrackSection kontrolowany jest za pomocą listy uprawnień utrzymywanej przez ten komponent. W oparciu o tę listę usługa NotificationService sprawdza, czy żądający informacji dyspozytor ma prawo do jej uzyskiwania

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 żądaną informację

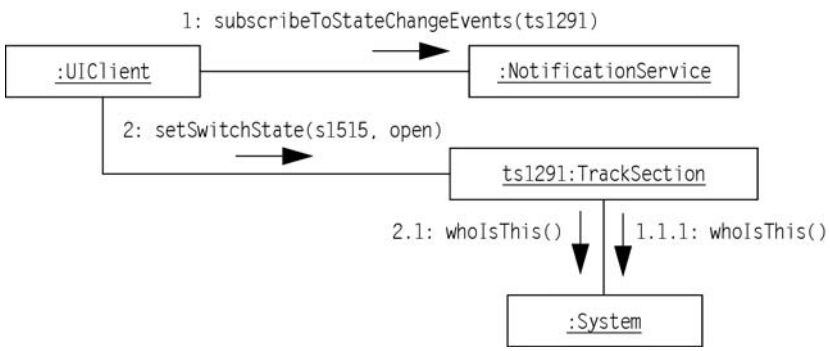
wszystkich „chronionych”[5] operacji i całokształtu kontroli dostępu. Ponadto, gdy zmienią się uprawnienia dostępu, klasa `TrackSection` może we własnym zakresie anulować subskrypcje pozostające w sprzeczności z nowymi uprawnieniami.

[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 `isAccessible()` klasy `TrackSection`. *Usługa dostępna jest bezpośrednio jedynie dla klasy* `TrackSection`, *podsystem* `UIClient` *nie ma do niej dostępu, subskrybuje on żądaną informację przez wywoływanie metody* `subscribeToEvents()` *klasy* `TrackSection`.
`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 listę utrzymywaną przez każdy obiekt tej klasy. *Klient, żądający informacji o stanie sekcji, powinien tę informację subskrybować, wywołując metodę* `subscribeToEvents()` *klasy* `TrackSection`. *Klasa* `TrackSection` *po pozytywnym zweryfikowaniu uprawnień podsystemu żądającego informacji wywołuje operację* `subscribeToTrackSectionEvents()` *usługi* `NotificationService`.

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 wywoływanie 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. ~~*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`~~.

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
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?

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 rzeczonej 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 poświeconego 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 pokaźnym 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 wysyła 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?

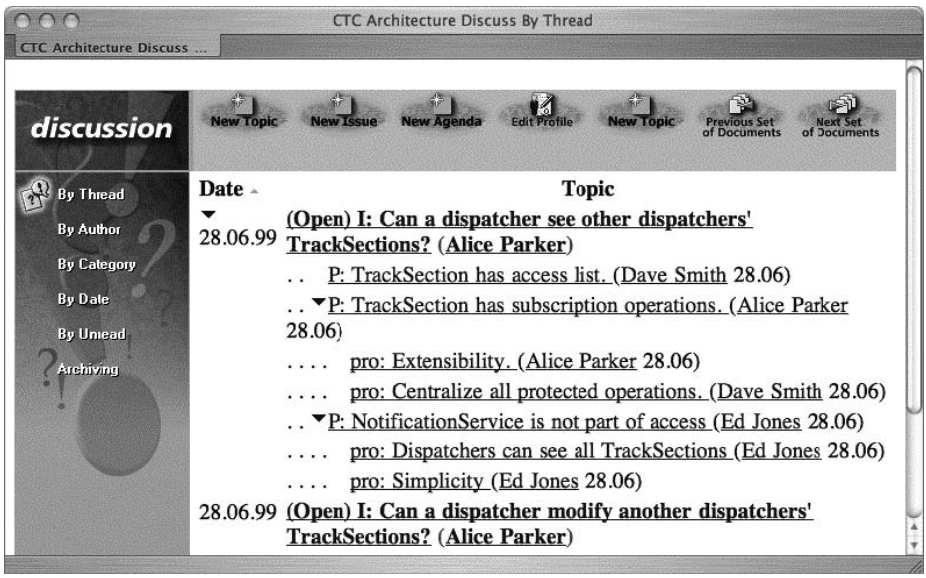
Data: 15 wrz, czw, 13:12:48 -0400

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 integrowanie 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 wielożytkowa baza zagadnień dostępna w sieci WWW (czego przykład widzimy na rysunku 12.17). Ustanawiając odpowiednie procedury wspomnianej integracji, możemy kolekcjonować obszerną informacją 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 pierwowzór 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, 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 rzeczonej 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 hiperłączy, 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 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.

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 identyfikacje dyspozytorów uprawnionych do obserwowania ich stanu („czytelników")[6]. Ze względów ogólności (rozszerzalności) lista zaimplementowana jest w ten sposób, że może zawierać wielu „czytelników" 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, wywoływana 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 „czytelników 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 `TrackSection` i usługą powiadamiania?**	
Kontrola dostępu w systemie CTC realizowana jest na poziomie sekcji dyspozycyjnych (`TrackSection`): dyspozytor (`Dispatcher`) 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 `TrackSection` odpowiedzialna jest za modyfikowanie stanu sekcji oraz subskrypcje związane z powiadamianiem.** Kontrola dostępu zaimplementowana jest w formie listy uprawnień utrzymywanej przez obiekty klasy `TrackSection`. Uprawnienia sprawdzane są dla każdej operacji odczytującej lub modyfikującej stan wspomnianych obiektów. Podsystem zamierzający subskrybować powiadomienia o zdarzeniach wywołuje odpowiednie metody klasy `TrackSection`, które z kolei przekazują wywołanie do usługi powiadamiania (`NotificationService`) 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 `TrackSection` zgrupowane są w tej klasie.
P[2]: **Klasa `TrackSection` odpowiedzialna jest wyłącznie za modyfikowanie stanu sekcji, usługa powiadamiania (`NotificationService`) zarządza tylko powiadamianiem.** Podobnie jak `P[1]`, z tą jednak różnicą, że podsystem żąda subskrypcji bezpośrednio od usługi powiadamiania, która odwołuje się do obiektu `TrackSection` w celu zweryfikowania uprawnień do tej subskrypcji.	Na rzecz propozycji: • Interfejs niezależny od kontroli dostępu: interfejsy klas `NotificationService` i `TrackSection` są takie same jak w przypadku nieobecności kontroli dostępu. • Przeciwko propozycji: • Cykliczna zależność między klasami `NotificationService` i `TrackSection`: obiekt `TrackSection` wywołuje usługę powiadamiającą w celu zasygnalizowania zdarzenia, zaś usługa powiadamiająca odwołuje się do obiektu `TrackSection` 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).

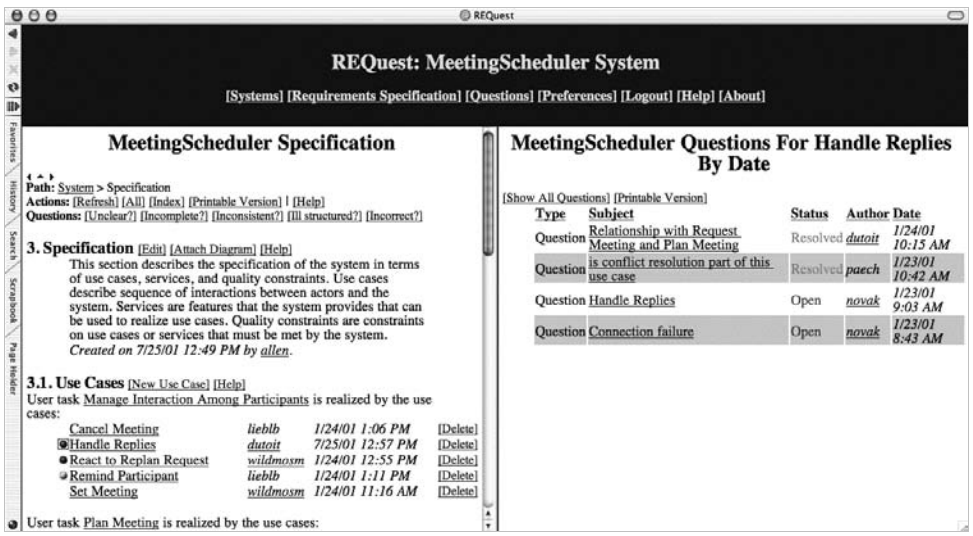
Jak poprzednio, koncentrujemy się na racjonalizacji związanej z etapem projektowania systemu, opisywane techniki przydatne są jednak równie dobrze na wszystkich pozostałych etapach.

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 rychło 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.



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 koleją rzeczy 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 zainteresowanych 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 krótka, sugestywna fraza określająca jego kategorię (na przykład „zagadnienie_dostęp/powiadamianie").
- *Scentralizuj zagadnienia*. Mimo iż zagadnienia mogą być w 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 dyskutowanie 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 priorytecie może równać się ignorowaniu interesów jednej lub kilku negocjujących stron.

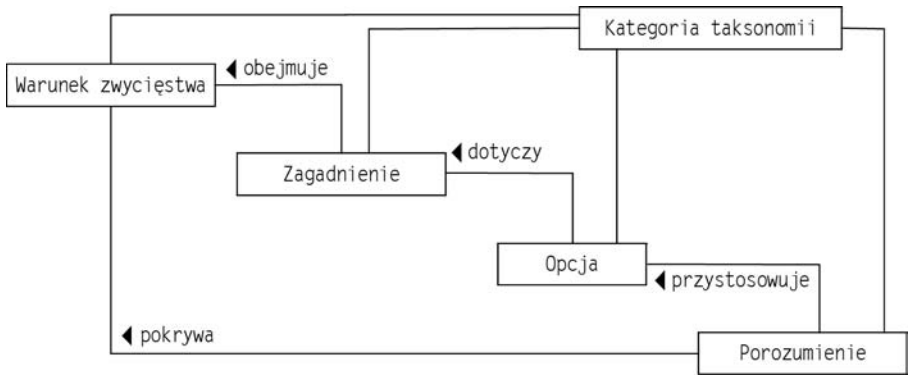
Postrzeganie realizacji projektu w kategoriach negocjacji uwydatnia jego aspekt społeczny. Programiści są przecież ludźmi, a zatem — poza oczywistym nastawieniem technicznym — prezentują określony stosunek emocjonalny do poszczególnych ewentualności rozwiązania określonego problemu. Przekłada się to 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 rozstrzyganiem 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. Rozmaite narzędzia wspomagające współpracę grupową umożliwiają przypisanie wag istotności poszczególnym argumentom w modelu zagadnień i wyłonienie „zwycięskiej" propozycji na podstawie formuły arytmetycznej, o czy 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 dyskutowanego 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 dokonać 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 czy 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 beneficjantami 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 uczyniliśmy 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 Plannung, 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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Ź

Ż

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.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

helion.pl
księgarnia internetowa

Cena 129,00 zł

ISBN 978-83-246-2872-8

9788324628728

Informatyka w najlepszym wydaniu