Projektowanie obiektowe. Role, odpowiedzialność i współpraca

Autorzy: Rebecca Wirfs-Brock, Alan McKean
Tłumaczenie: Przemysław Kowalczyk
ISBN: 83-246-0046-9
Tytuł oryginału: Object Design: Roles, Responsibilities, and Collaborations
Format: B5, stron: 352

Projektowanie i programowanie obiektowe to dziś standard w produkcji oprogramowania. Język UML, powszechnie stosowane narzędzie opisu projektów i architektury oprogramowania, systematyzuje i upraszcza proces projektowania. Projektowanie systemów w oparciu o przypadki użycia oraz role, odpowiedzialność i współpracę obiektów, pozwala na skoncentrowanie się na tym, jak powinien działać system, bez zbyt wczesnego zagłębiania się w szczegóły implementacyjne. Dopiero po opracowaniu prawidłowego projektu można zacząć zastanawiać się, jak zaimplementować projekt przy użyciu klas, interfejsów i hierarchii dziedziczenia.

Książka „Projektowanie obiektowe. Role, odpowiedzialność i współpraca" przedstawia metodykę projektowania obiektowego noszącą nazwę „Projektowania Sterowanego Odpowiedzialnością". Przedstawia praktyczne zasady projektowania obiektów będących integralnymi elementami systemu, w którym każdy obiekt ma specyficzną rolę i zakres odpowiedzialności. Autorzy prezentują najnowsze praktyki i techniki „Projektowania Sterowanego Odpowiedzialnością", a także przedstawiają sposoby ich stosowania w rozwoju nowoczesnych aplikacji obiektowych. Książka przedstawia strategie znajdowania kandydatów na obiekty i zawiera praktyczne przykłady oraz porady, dzięki którym bez problemu wykorzystasz opisywane w niej metody.

- Stereotypy ról obiektów
- Analiza opisu systemu
- Model biznesowy systemu
- Wyszukiwanie kandydatów na obiekty
- Przydzielanie odpowiedzialności obiektom
- Definiowanie współpracy pomiędzy obiektami
- Przekazywanie sterowania w obiektach i systemie

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Projektowanie sterowane odpowiedzialnością

Betty Edwards, autorka książki *Rysowanie a wewnętrzny artysta*, przekonuje, że wielu tak zwanych „kreatywnych" talentów można się nauczyć. Proponuje doskonały eksperyment myślowy:

„Czego trzeba, aby nauczyć dziecko czytać? Co by było, gdybyśmy uważali, że tylko niektórzy, szczęśliwie obdarowani przez naturę, posiadają «kreatywną» zdolność czytania? Gdyby nauczyciele byli przekonani, że najlepszym sposobem nauki jest zarzucić dziecko dużą ilością materiałów, poczekać i przekonać się, czy posiadło wrodzony talent do czytania? Gdyby strach przed zduszeniem kreatywnego procesu czytania powstrzymywał wszelkie próby pomocy nowym czytelnikom? Gdyby dziecko zapytało, na czym polega czytanie, a nauczyciel odpowiedział: «Spróbuj dowolnej metody, która ci przyjdzie do głowy. Ciesz się tym, badaj różne możliwości, czytanie to taka frajda!»? Być może okazałoby się, że jedno lub dwoje dzieci w każdej klasie posiada taki rzadki talent i samo potrafi nauczyć się czytać. Ale, oczywiście, taka metoda nauki jest absurdalna! Czytania można się nauczyć. Podobnie jak rysowania".

Książka Betty Edwards zawiera argumenty przeciwko poglądowi, że umiejętność rysowania wymaga rzadkiego, „artystycznego" talentu oraz że sformalizowana nauka podstawowych technik rysowania hamuje kreatywność. Owych podstawowych technik, podobnie jak technik czytania, można się nauczyć. Nic dziwnego, że wielu z nas nie umie rysować! Nauka rysowania polega po prostu na poznaniu podstawowych umiejętności percepcyjnych — nabyciu odpowiedniego sposobu widzenia przedmiotów, który jest wymagany do zrobienia porządnego rysunku.

Niniejszy rozdział przedstawia podstawowe czynności związane z projektowaniem sterowanym odpowiedzialnością, a także prezentuje przykłady pracy projektowej. Ponieważ projektowanie obiektowe jest procesem wymagającym wysokiej kreatywności, projektanci powinni wiedzieć, kiedy należy zastosować odpowiednie narzędzia pomagające przedstawić problem konceptualnie lub wymyślić rozwiązanie.

Projektowanie obiektowe nie wymaga żadnego rzadkiego, ani specjalnego talentu „projektowego". Chociaż jest czynnością wymagającą dużej kreatywności, jego podstaw można się łatwo nauczyć. Każdy może stać się adeptem projektowania obiektowego. Wystarczy trochę praktyki i doświadczenia, aby nabyć zdolności widzenia natury problemu projektowego i nauczyć się podstawowych strategii tworzenia akceptowalnych rozwiązań.

W tym rozdziale przedstawimy podstawowe kroki rozwoju aplikacji obiektowych przy użyciu metodologii zwanej projektowaniem sterowanym odpowiedzialnością. Najpierw opisujemy akcje i czynności, za które nasze oprogramowanie powinno być „odpowiedzialne". Odpowiedzialność określamy pojęciami, które rozumieją zarówno twórcy, jak i użytkownicy aplikacji. Potem przenosimy naszą uwagę na zaprojektowanie obiektów programistycznych, które implementują wymaganą odpowiedzialność.

Proces widzenia, opisywania i projektowania

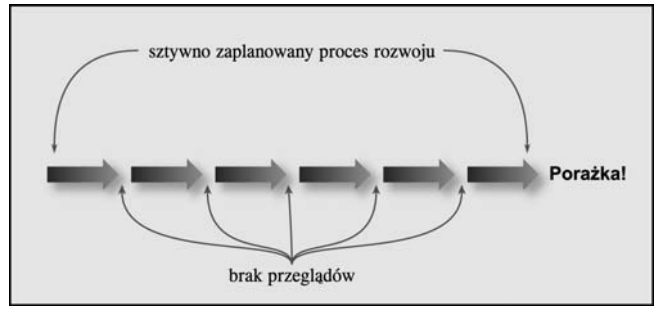
Posiadanie talentu projektowania obiektowego oznacza, że dzięki doświadczeniu, wyrobionej intuicji albo wrodzonej zdolności łatwo potrafimy „wpaść" na rozwiązania, które wymagają od innych wiele wysiłku lub nauki. Szybko udaje nam się dojść do samego sedna problemu i znaleźć sposoby zaprojektowania akceptowalnego rozwiązania.

Na początek chcielibyśmy jasno powiedzieć jedną rzecz: chociaż nasza książka przedstawia czynności związane z tworzeniem projektów obiektowych w sposób liniowy, w praktyce projektowanie bardzo rzadko odbywa się w określonej z góry kolejności. Procesy związane z tworzeniem projektu są bardzo płynne i zależne od pojawiających się w ich trakcie problemów i pytań; niezależnie od tego, że końcowe rezultaty projektowania są bardzo sztywno osadzone w kodzie wynikowym. Nasze możliwości opisania tych burzliwych często czynności są ograniczone przez właściwości słowa drukowanego.

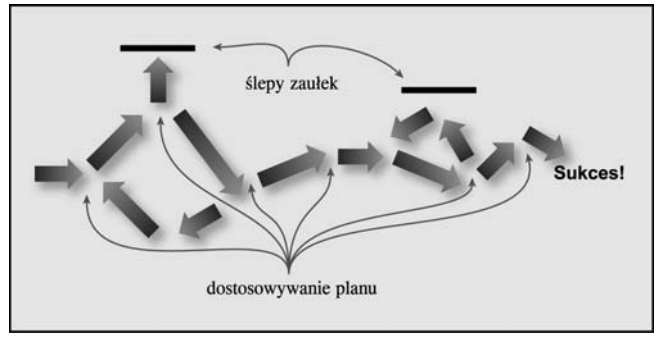
Projektowanie sterowane odpowiedzialnością jest metodologią nieformalną. Składa się z wielu technik, które wspomagają nasz sposób myślenia o tym, w jaki sposób rozdzielić odpowiedzialność aplikacji pomiędzy obiekty i jak sterować ich zachowaniem. Naszym podstawowym narzędziem jest zdolność abstrahowania — tworzenia obiektów, które reprezentują istotę działającej aplikacji.

Nazwa naszej metody projektowej podkreśla znaczenie wątku, który przewija się w każdej czynności: naszego skupienia na odpowiedzialności oprogramowania. Opisuje ona, co nasze obiekty muszą robić, aby spełnić cel swojego istnienia. Nasza praca rozpoczyna się od zebrania wymagań, potem zajmujemy się przybliżonym naszkicowaniem pomysłów, które następnie uszczegóławiamy, opisujemy i przekształcamy w modele programistyczne. Co zaskakujące, na początku projektowania nie skupiamy się na obiektach. Zamiast nich najważniejsze jest ujęcie w opisie przyszłego systemu różnorodnych punktów widzenia zleceniodawców i użytkowników. W naszych rozwiązaniach musimy wziąć pod uwagę wiele różnych perspektyw. Projektowanie sterowane odpowiedzialnością jest procesem wyjaśniania. Przechodzimy od początkowych wymagań do wstępnych opisów i modeli; od ogólnych opisów do bardziej szczegółowych modeli obiektów; od kandydatów na obiekty do precyzyjnych modeli ich odpowiedzialności i wzorców współpracy.

Podczas pracy nigdy nie posuwamy się prostą ścieżką, którą przedstawia rysunek 2.1. Zamiast tego, jak widać na rysunku 2.2, nasza podróż przez projektowanie wypełniona jest zakrętami, nawrotami i wypadami w bok. Kiedy próbujemy stworzyć rozwiązanie projektowe, przechodzimy często pomiędzy różnymi czynnościami, odkrywając różne aspekty problemu. Podchodzimy do niego oportunistycznie — próbujemy wykorzystać nadarzające się okazje. Używamy różnorodnych narzędzi, które pozwalają nam uzyskać



Rysunek 2.1. *Sztywno i dokładnie zaplanowana praca często kończy się porażką*

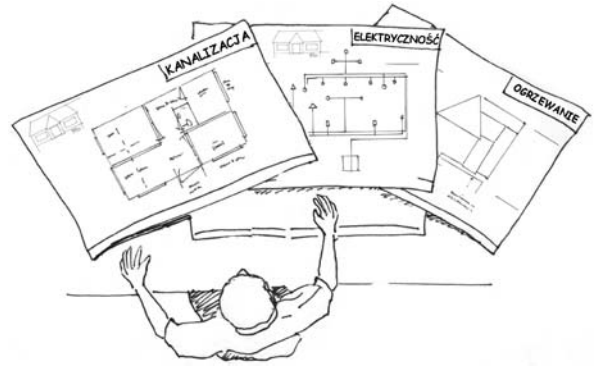


Rysunek 2.2. *Ścieżka projektowania sterowanego odpowiedzialnością jest bardzo elastyczna*

odpowiednią perspektywę, odkryć informacje oraz ukształtować rozwiązania. Nasza praca jest elastyczna i przybiera różne formy.

Kolejność naszych czynności oraz przedmiot, na którym się skupiamy, będą się z konieczności wciąż zmieniać (zobacz rysunek 2.3). Planowanie, dodawanie nowych możliwości, określanie celów, charakteryzowanie aplikacji przez prototypy, tworzenie modelu obiektowego, identyfikowanie trudnych problemów — to tylko niektóre z naszych zadań. Różnią się one swoimi celami, rygorem, przedmiotem, naciskiem, kontekstem oraz narzędziami, które do nich stosujemy.

Nasza prezentacja czynności projektowych jest liniowa, ponieważ ograniczają nas ramy drukowanych, numerowanych stron. Czytając niniejszą książkę, należy zadawać sobie pytania: „Gdzie mogę wykorzystać tę technikę w pracy nad moim projektem? Jakie narzędzie myślowe byłoby w obecnej chwili najbardziej przydatne?". Trzeba wykorzystywać okazje!

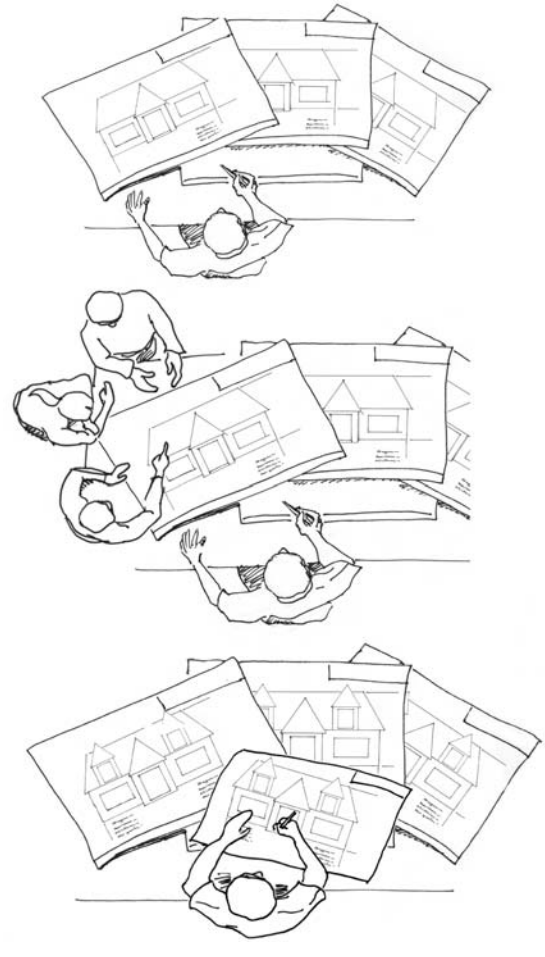


Rysunek 2.3. *Wciąż przenosimy naszą uwagę z jednego obszaru projektu na inny, przekształcając nasze wizje i odkrywając nowe szczegóły*

Nawet przy stosunkowo prostych projektach programistycznych nie możemy przewidzieć wszystkich problemów, pojawiających się w trakcie. Przy takiej złożoności projektów nie wszystkie nasze decyzje będą optymalne. Nasze postępy nie będą jednostajne. W czasie pracy odkrywamy nowe informacje i ograniczenia. Musimy poświecić trochę czasu, aby cofnąć się i „wygładzić" narastające niespójności.

Marvin Minsky powiada, że ludzka inteligencja sprowadza się do zdolności negocjowania rozwiązań i rozstrzygania konfliktów pomiędzy sprzecznymi celami. Jeżeli jedna część naszego umysłu proponuje rozwiązanie, a inna twierdzi, że nie jest ono akceptowalne, zwykle potrafimy znaleźć inny sposób. Jeżeli nie możemy dostrzec rozwiązania, po prostu szukamy lepszej perspektywy.

Aby skompensować niedostatki naszej zdolności przewidywania problemów i zagrożeń, planujemy przerwy na ponowne przeglądanie, poprawianie i dostosowywanie naszego projektu do zmieniającego się zbioru warunków i ograniczeń. Pozwala nam to użyć naszej rosnącej wiedzy i pogłębiającego się zrozumienia problemu do poprawienia efektów wcześniejszych stadiów rozwoju projektu. Jak wynika z rysunku 2.4, nasz proces jest iteracyjny i przyrostowy. Podczas rozwoju projektu przenosimy nacisk ze zbierania wymagań i tworzenia specyfikacji na analizę, projektowanie, testowanie i kodowanie. Zawsze możemy jednak powrócić do poprzednich czynności, aby odkryć nowe aspekty napotkanego problemu.



Rysunek 2.4. *Odkrywanie nowych aspektów obejmuje powstanie pomysłu, przedstawienie go zleceniodawcy w celu uzyskania sprzężenia zwrotnego oraz wprowadzanie zmian i nowych informacji do poprawionego modelu*

Jako projektanci mamy naturalne przekonanie, że obiekty stanowią centrum programistycznego wszechświata. Jednakże *nasza* „obiektowość" nie może przesłaniać nam faktu, że w wymyślaniu, projektowaniu i konstruowaniu udanej aplikacji biorą udział również inni uczestnicy i perspektywy. Podobnie jak na przykład przedstawienie w teatrze, produkcja oprogramowania wymaga znacznie więcej, niż widać w końcowym efekcie. I chociaż w naszej pracy obiekty mogą odgrywać główną rolę, ważne jest również, aby brać pod uwagę wpływ innych perspektyw i czynności na nasz projekt.

Uruchamianie produkcji — definicja i planowanie

Zastosujemy tradycyjne podejście do opisywania procesu projektowania obiektowego. Zacznijmy od początku. Zanim skoczymy na głęboką wodę, należy najpierw zdefiniować cele projektu, skonstruować plan ich osiągnięcia oraz zapewnić dla nich akceptację zleceniodawcy.

W długich lub złożonych projektach musimy przedyskutować i udokumentować wymagania użytkowników oraz zademonstrować, jak nasz system będzie służył tym, którzy za niego zapłacą. Musimy pamiętać, że nasz sukces lub porażka najmocniej odbiją się na naszych zleceniodawcach. Nawet przy niewielkich projektach trochę planowania nigdy nie zaszkodzi. Dzięki niemu uzyskujemy zwarte przedstawienie projektu, które składa się z określenia zamiarów, ogólnego opisu rozwiązania oraz definicji celów i korzyści.

Planowanie projektu przygotowuje scenę dla rozwoju naszych pomysłów. Jest scenariuszem naszego działania. Musimy pamiętać, że naszym głównym celem jest zadowolenie użytkowników i zleceniodawców. Plan projektu zawiera opis następujących zagadnień:

- Jak będziemy tworzyć oprogramowanie?
- Wartości, które są ważne dla projektu i ludzi weń zaangażowanych.
- Ludzie, ich role, procesy i oczekiwane wyniki.
- Oczekiwana postać końcowa produktu.

Planowanie i definiowanie projektu są fundamentalnymi czynnościami, ale nie są przedmiotem tej książki. Mając plan czynności, możemy zająć się strukturami i procesami. Naszym celem jest zrozumienie, co powinno robić nasze oprogramowanie oraz jak będzie wspomagać swoich użytkowników.

„Jest to w dużym stopniu kwestia artystyczna. Twórca projektu działa podobnie jak starożytny bard, którego epickie poematy nie były zapisywane, lecz recytowane z pamięci. Musi wybierać struktury, które łatwo zapamiętać, dzięki czemu publiczność nie zgubi głównego wątku opowieści".

Michael Jackson

Przygotowanie sceny — wstępny opis

Początkowo staramy się zawęzić pole działania i uszczegółowić nasze opisy. Zaczynamy od niedokładnych szkiców, oszukujemy w obszarach wymagających szczegółów, których nie potrafimy jeszcze określić. Powtarzamy cykle odkrywania, refleksji i opisywania. Krok po kroku dodajemy szczegóły, wyjaśniamy nieścisłości i rozstrzygamy sprzeczności między wymaganiami. Początkowo nasze opisy nie mówią nic o obiektach. Perspektywę obiektową dodajemy dopiero po ogólnym opisaniu całego systemu. Pojęcia obiektowe

będą stanowić jądro modelu wewnętrznych składników naszego systemu. Naszą receptę na analizę systemu stanowi tabela 2.1.

Tabela 2.1. *Analiza zawiera definicję systemu, jego opis oraz czynności związane z analizą obiektową*

Analiza sterowana odpowiedzialnością		
Faza	**Czynność**	**Rezultaty**
Definicja systemu	Tworzenie wysokopoziomowej architektury systemu	Diagram granic systemu
		Diagramy wysokiego poziomu architektury technicznej
		Opisy i diagramy pojęć w systemie
	Identyfikacja początkowych pojęć w systemie	Słownik pojęć
	Identyfikacja odpowiedzialności systemu	Perspektywa i funkcje systemu
		Charakterystyki użytkowania
		Ogólne ograniczenia, założenia i zależności
Szczegółowy opis	Specyfikacja środowiska rozwojowego	Dokumentacja istniejących schematów rozwojowych, programów zewnętrznych, API oraz narzędzi komputerowych
	Tworzenie tekstowych opisów sposobów, w jakie użytkownicy chcieliby wykonywać swoje zadania	Lista aktorów — różnych typów użytkowników oraz systemów zewnętrznych, które mają kontakt z naszym systemem
		Opisy przypadków użycia — niesformalizowane, tekstowe opisy zadań użytkowników
		Scenariusze i konwersacje — bardziej szczegółowe i formalne opisy konkretnych przykładów użycia
	Analiza specjalnych wymagań, które mają wpływ na projekt	Strategie zwiększania wydajności, korzystania z danych wcześniejszych systemów, planowanie obsługi rozproszonych danych i przetwarzania, odporności na błędy, niezawodności
	Dokumentacja dynamiki systemu	Diagramy aktywności pokazujące ograniczenia pomiędzy przypadkami użycia
	Ukazanie ekranów aplikacji i interakcji z perspektywy użytkownika	Specyfikacje ekranów
		Model nawigacji
Analiza obiektowa	Identyfikowanie obiektów dziedzinowych oraz ich intuicyjnych zakresów odpowiedzialności	Karty CRC, opisujące role i odpowiedzialność obiektów
		Wstępny model obiektowy
	Dokumentowanie dodatkowych pojęć i terminów	Słowniki definiujące koncepcje, opisy zachowania i reguły biznesowe

Oczywiście, w każdym projekcie rezultaty mogą być trochę inne. W zależności od specyfiki systemu niektóre dokumenty mogą nie być zbyt przydatne. Na przykład jeżeli tworzymy aplikację, która nie obsługuje pracy interaktywnej z użytkownikami, możemy pominąć projekty ekranów. Aby zaprojektować odpowiedzialność, skupiamy się na tych opisach, które dają nam wartościowe perspektywy. Niektóre wymagania odkrywamy już podczas dyskusji ze zleceniodawcami. Odpowiadają one z grubsza wymaganiom

użytkowników, ale obejmują również pewną liczbę wymagań klienckich oraz administracyjnych:

- Użyteczność
- Wydajność
- Konfigurowalność
- Autoryzacja użytkowników
- Współbieżność
- Skalowalność
- Bezpieczeństwo
- Niezawodność

Czasem tego typu wymagania wychodzą na jaw dopiero podczas rozwoju lub nawet wstępnego użytkowania systemu przez programistów, testerów i użytkowników wersji beta. Wiele wymagań i aspektów często nakłada się na siebie, ale różni zleceniodawcy przedstawiają je w różnej formie. Bezpieczeństwo może stanowić najważniejszy aspekt aplikacji dla użytkowników, którzy nie chcą, by „dane o kartach kredytowych wyciekały do Internetu", ale nie są to wymagania równie szczegółowe, jak te, które przedstawia ekspertowi od bezpieczeństwa internetowego administrator serwisu WWW.

Oprócz tych oczywistych wymagań, które mają mierzalny i bezpośredni wpływ na projekt, dodatkowe wymagania elastyczności, łatwości konserwowania, rozszerzania czy ponownego używania komponentów mogą ograniczać akceptowalne rozwiązania, chociaż nie widać ich zwykle z perspektywy interakcji użytkownika z aplikacją. W wielu przypadkach jednak to właśnie te cechy, jeżeli będziemy je ignorować, mogą prowadzić do niepowodzenia projektu. Jako projektanci powinniśmy ze zbioru wymagań stworzyć projekt, który jest z nimi zgodny. Jednak musimy być przygotowani na to, że niezależnie od naszych starań, i tak nie uda nam się od razu zidentyfikować wszystkich wymagań.

Przystępujemy do produkcji — projekt

Projektując, konstruujemy model pracy naszego systemu. Proces projektowania obiektowego możemy rozbić na dwie główne fazy: tworzenie początkowego projektu (praca badawcza opisana w tabeli 2.2), a następnie konstruowanie bardziej szczegółowych rozwiązań (uściślanie pokazane w tabeli 2.3).

Tabela 2.2. *Projektowanie badawcze koncentruje się na stworzeniu początkowego modelu obiektowego systemu*

Projektowanie badawcze	
Czynność	**Rezultaty**
Połączenie obiektów dziedzinowych ze sterującymi	Zbiór kart CRC modelujących obiekty, role, odpowiedzialność i współpracę
Przypisanie zakresów odpowiedzialności do obiektów	Diagramy sekwencji lub współpracy
	Opisy odpowiedzialności i współpracy podsystemów
Rozwój początkowego modelu współpracy	Początkowe definicje klas
	Działające prototypy

Tabela 2.3. *Uściślanie projektu obejmuje czynności, dzięki którym projekt staje się bardziej przewidywalny, spójny, elastyczny i zrozumiały*

Uściślanie projektu	
Czynność	**Rezultaty**
Uzasadnianie kompromisów	Dokumentacja decyzji projektowych
Rozłożenie sterowania aplikacją	Zidentyfikowane style sterowania
	Łatwe do zrozumienia wzorce podejmowania decyzji i delegowania zadań w modelu obiektowym
Określenie statycznej i dynamicznej relacji widzialności pomiędzy obiektami	Poprawione definicje i diagramy klas
Poprawianie modelu w celu ułatwienia konserwacji, zwiększenia elastyczności i spójności	Stworzenie nowych abstrakcji obiektowych
	Poprawione role obiektów, obejmujące mieszanki stereotypów
	Uproszczone, spójne interfejsy i wzorce współpracy
	Specyfikacje klas implementujących poszczególne role
	Zastosowanie wzorców projektowych
Jasne udokumentowanie projektu	Diagramy UML opisujące pakiety, komponenty, podsystemy, klasy, sekwencje interakcji, współpracę, interfejsy
	Kod
Formalizacja projektu	Kontrakty pomiędzy komponentami systemu oraz kluczowymi klasami

Ilość czasu, który poświęcamy na badanie alternatyw oraz poprawianie projektu, a także objętość dokumentacji, którą możemy stworzyć, zależy w dużym stopniu od problemu. Radzimy skupić się na tych czynnościach, które wnoszą wymierny wkład do projektu. Nie musimy koniecznie wykonać każdej z wymienianych tutaj czynności. Także produkowanie stosów dokumentów projektowych nie jest gwarancją sukcesu. Przedstawione tutaj czynności oraz uzyskiwane rezultaty należy traktować jako ogólny przewodnik i dostosować je do własnych potrzeb.

W pewnym momencie dochodzimy do punktu, w którym gotowy jest początkowy projekt badawczy i chcielibyśmy zakończyć projektowanie, a rozpocząć programowanie. Może się tak zdarzyć nawet po stosunkowo krótkim czasie, szczególnie gdy projekt jest prosty i wiemy, jak go wykonać. Czasem chcemy dowieść poprawności jakiejś jego części przez implementację prototypu, zanim zainwestujemy czas i energię w projektowanie pozostałych podsystemów, które zależą od tego, czy nasze rozwiązanie się sprawdzi. Może być i odwrotnie, zanim zaczniemy implementację, zechcemy wprowadzić poprawki do naszego projektu. Niezależnie od tego, czy poświęcimy więcej czasu na doskonalenie projektu, czy też będziemy go poprawiać równolegle z implementowaniem, nasze początkowe pomysły projektowe z pewnością się zmienią. Większość aplikacji jest zbyt złożona, aby za pierwszym razem stworzyć „prawidłowy" projekt. Tak więc tworzenie działającego rozwiązania oznacza częste powracanie do początkowych założeń, aby upewnić się, że system kształtuje się zgodnie z oczekiwaniami zleceniodawców. Może też oznaczać, że potrzebujemy dodatkowego czasu na zaprojektowanie elastycznego rozwiązania albo wprowadzenie obsługi warunków wyjątkowych.

Czynności projektowe — od początkowych badań po szczegółowe uściślenia — stanowią główny temat tej książki. Ale zanim zagłębimy się w projektowanie, wyjaśnijmy, co powinniśmy „wyraźnie widzieć", aby stworzyć odpowiedni projekt.

„Widzenie" z wielu perspektyw

„Fakty są dla naukowców jak powietrze. Bez niego nie potrafią latać".
Iwan Pawłow

Każdy zleceniodawca i użytkownik w naszym procesie projektowym ma różne potrzeby i priorytety. Każda osoba będzie patrzeć na nasze postępy i powstającą aplikację z własnej, unikalnej perspektywy. Ponieważ większość zleceniodawców nie mówi naszym ojczystym, „obiektowym" językiem, przed projektantami stają dwa wyzwania:

- Jak poprawnie zinterpretować wymagania zleceniodawców i ich priorytety?
- Jak przedstawiać nasze projekty w sposób zrozumiały dla osób bez wykształcenia informatycznego?

Każdy uczestnik procesu produkcji oprogramowania ma inne kryteria oceniania naszych osiągnięć. Różne punkty widzenia wpływają na zróżnicowanie priorytetów i aspektów, które są istotne dla różnych osób.

Na przykład przyszli użytkownicy chcą zobaczyć, czy aplikacja ułatwi im wykonywanie swoich zadań. Chcą, by sterowanie i przetwarzanie aplikacji było spójne i „naturalne". Analityk biznesowy będzie wolał upewnić się, że zespół rozwojowy prawidłowo rozumie i potrafi zaimplementować wszystkie reguły i procesy logiki biznesowej. Tester chce sprawdzić, czy aplikacja spełnia wszystkie wymagania oraz, między innymi, zakładane cele wydajnościowe. Niektórzy zleceniodawcy interesują się czasem szczegółami i jakością naszego projektu, ale większość zwykle tego nie robi. Wszyscy będą żądali zapewnień, że nasz projekt odpowiada ich potrzebom i priorytetom. Zróbmy zatem szybki przegląd całego procesu, aby zobaczyć, jak stworzyć projekt odpowiadający specyficznym potrzebom naszych zleceniodawców.

Pisanie scenariusza — analiza opisów

„Nie ma sensu mówić o czymś precyzyjnie, jeżeli nawet nie wiemy, o czym mówimy".
John von Neumann

W początkowej fazie procesu naszym podstawowym celem jest zrozumieć najważniejsze wymagania i dać temu wyraz. Przekształcamy mgliste, twórcze pomysły w specyfikacje tego, co mamy zbudować. Błędy w specyfikacji produktu są najbardziej kosztowne, bo rzutują na wszystkie późniejsze czynności. Jest więc bardzo ważne, aby komunikować charakterystyki naszego oprogramowania przy użyciu prostego, jednoznacznego języka tym osobom, które będą go używać oraz konserwować. Aby zrozumieć, jak nasz system dopasowuje się do bezpośredniego środowiska, w którym będzie uruchamiany, a także jak komunikuje się z nieco szerszym sąsiedztwem zewnętrznych urządzeń, baz danych czy innych programów, używamy różnych perspektyw, które ilustruje rysunek 2.5.

Rysunek 2.5. *Zleceniodawcy widzą i opisują system ze swoich unikalnych punktów widzenia*

„Opisy są widocznym medium przekazywania myśli".
Michael Jackson

Jakim językiem powinniśmy opisywać nasz system? Nie istnieje jeden język, wspólny dla użytkowników, klientów, analityków danych, projektantów, programistów i menedżerów, którym moglibyśmy odpowiednio opisać nasze oprogramowanie. Zbieramy cały wachlarz opisów, używając różnych języków i notacji. Naszym celem jest wyjaśnienie tego, co niejasne oraz zebranie i opisanie jednym głosem tego, za co ma być odpowiedzialne nasze oprogramowanie. Gromadzimy różne opisy i wprowadzamy uwidocznione w nich aspekty do naszych specyfikacji. Staramy się zrozumieć, gdzie nasz system powinien się „kończyć", gdzie „zaczyna" się jego środowisko zewnętrzne oraz jakie funkcje powinien wykonywać. Kiedy już nakreślimy te granice, skupiamy się na wewnętrznych szczegółach naszego oprogramowania oraz sposobach jego komunikacji ze środowiskiem. Rozwijamy spójny, wspólny słownik pojęć i używamy go konsekwentnie do opisywania zleceniodawcom przedmiotów, które mają związek z naszym systemem, procesów, które wspiera oraz zakresów odpowiedzialności, które implementuje.

Opisy użytkowania

Ponieważ wiele wymagań do aplikacji pochodzi od jej przyszłych użytkowników, muszą oni prawidłowo je rozumieć. Z punktu widzenia użytkownika istnieje wokół naszego systemu granica, która oddziela nasze oprogramowanie od świata zewnętrznego. Użytkownicy postrzegają system głównie w aspekcie wspierania przezeń ich pracy. Takie zadaniowe spojrzenie może być opisane przez zbiór przypadków użycia. Są one częścią modelu w języku UML. Rozbijamy specyfikację wielkiej aplikacji na przypadki użycia, które w zwarty sposób opisują oddzielne „kawałki" funkcjonalności z zewnętrznej perspektywy.

Przypadki użycia oraz zorientowanie projektu na potrzeby użytkownika są ważne, ale nie obejmują całego systemu. Model jest zbiorem powiązanych opisów. Istnieją różne typy modeli — użytkowania, danych, obiektów, stanów, procesów i wiele innych.

Aktorzy i ich spojrzenie na nasz system

Język UML definiuje aktora jako *kogoś* lub *coś*, co jest na zewnątrz systemu i wchodzi w interakcje z nim. Aktorów możemy podzielić na trzy najważniejsze grupy:

- użytkowników,
- administratorów,
- programy i urządzenia zewnętrzne.

Wszyscy aktorzy ma dwie wspólne cechy:

- są na zewnątrz aplikacji,
- mają inicjatywę, powodują zdarzenia lub wchodzą w interakcje z naszym systemem.

Obiekty najlepiej opisują pojęcia i przedmioty, ich charakterystyki, odpowiedzialności i interakcje.

Organizując opisy użytkowania wokół aktorów, sprawiamy, że zakresy odpowiedzialności naszego oprogramowania zorientowane są na punkty widzenia każdego aktora. W późniejszych krokach będziemy rozwijać projekt na podstawie tych opisów oraz mając na uwadze pożądane cechy systemu. Ale w obecnym stadium rozwoju potrzebujemy innych, bardziej abstrakcyjnych opisów, aby stworzyć pojedynczy, spójny model obiektowy. Potrzebujemy opisów bogatych w szczegóły, intencje, implikacje i cele. Model obiektowy jedynie proponuje rozwiązanie problemu. Przemilcza ono codzienne potrzeby, intencje i priorytety zleceniodawców i użytkowników.

Jeżeli oczekujemy od naszego oprogramowania jakiejś cechy, nasze potrzeby muszą być udokumentowane w którymś z opisów. Pożądane cechy nie pojawiają się „same z siebie".

Bogate i szczegółowe opisy, których potrzebujemy, powinny przedstawiać funkcjonalność, punkty zmienności i konfiguracji oraz podstawy architektury systemu. Identyfikujemy grupy ludzi oraz programy i urządzenia zewnętrzne, z którymi komunikuje się nasza aplikacja i opisujemy sposoby tych interakcji. Notujemy, gdzie są obszary wymagające elastyczności oraz na jaką zmienność warunków powinna być przygotowana aplikacja. Najlepiej, jak potrafimy, staramy się, by nasza dokumentacja była zrozumiała dla tych, którzy potrzebują tej wiedzy. Jeżeli na tym etapie budujemy jakieś modele obiektowe albo prototypy kodu, to tylko by pogłębić własne zrozumienie niektórych wymagań. Stworzone teraz prototypy są zwykle później porzucane.

Przypadki użycia

Przypadek użycia to „powiązana zachowaniem sekwencja transakcji w dialogu z systemem".
Ivar Jacobson

Przypadki użycia, opisane w 1992 roku przez Ivara Jacobsona, są częścią języka UML. Wiele osób używa ich jako narzędzia opisu, jak będzie użytkowany przyszły system. Innym „do szczęścia" wystarcza hierarchiczna lista funkcji systemu, proste historie użytkowników lub długie dokumenty specyfikacyjne. Przypadki użycia są szczególnie cenne, ponieważ pozwalają uchwycić działanie aplikacji z zewnętrznej perspektywy użytkownika. Używamy trzech form opisów przypadków użycia: proste teksty, zwane *narracjami* (ang. *narrative*), *scenariusze* (ang. *scenario*) składające się z numerowanych kroków oraz *konwersacje* (ang. *conversation*) eksponujące dialog między użytkownikiem a systemem. Każda forma opisu przypadków użycia kładzie nacisk na inny ich aspekt.

Przypadki użycia mogą różnić się poziomem szczegółowości, w zależności od tego, dla kogo są przeznaczone. Możemy rozpocząć od bardzo ogólnego opisu, a następnie wzbogać go w szczegóły i opisywać sekwencje czynności i interakcji pomiędzy użytkownikiem a systemem. Forma, którą wybierzemy, zależy od tego, co próbujemy przekazać.

Możemy też połączyć więcej niż jedną formę w opisie przypadku użycia w zależności od potrzeb naszych odbiorców. Zwykle zaczynamy od opisu narracyjnego, który prezentuje ogólny przegląd przypadku użycia. Następnie, w razie potrzeby, możemy go wzbogacić w dowolną liczbę scenariuszy i konwersacji, które uszczegółowią ogólny obraz.

Przykład edytora tekstu

Rozważmy przypadki użycia, które potrzebne nam były do napisania niniejszego rozdziału. Pisanie książek nie jest główną funkcją edytora tekstu, którego używamy. Jest to ogólne narzędzie do przygotowywania dokumentów. Korzystając z niego, musimy dostosowywać nasze czynności do zadań, które wspomaga: wprowadzania i poprawiania tekstu. Brakuje mu innych funkcji, które przydatne byłyby przy pisaniu książki: prowadzenie badań, burze mózgów, tworzenie streszczeń i schematów. Zadania, które można znaleźć wśród funkcjonalności edytora tekstu, to otwieranie dokumentu, pisanie i redagowanie tekstu.

Naszym celem jest przedstawienie zadań użytkowników jak najbardziej opisowo. Funkcje systemu, które w ogólnym opisie mogą się wydawać bardzo proste, stają się często długą serią decyzji i czynności, podejmowanych przez użytkownika.

Pisanie to czynność dość swobodna. Łączymy i mieszamy zadania pisarskie w nieprzewidywalnej kolejności. Ponieważ edytor tekstu ma wspomagać szeroką gamę stylów pisania, jego funkcje najlepiej opisać za pomocą niewielkich przypadków użycia, które mogą być wykonywane w dowolnej kolejności. Ale zadania przydatne podczas pisania książki są większe; składają się z wielu podzadań. Formatowanie strony jest serią zmian marginesów, wcięć, nagłówków, stopek i tak dalej. Reorganizowanie sekwencji akapitów jest serią operacji „wytnij" i „wklej". Nadajemy nazwy przypadkom użycia i opisujemy je z punktu widzenia użytkownika — na przykład „Redaguj tekst", „Zapisz dokument do pliku" czy „Wyszukaj w pomocy kontekstowej". Zauważmy, że w naszych przykładach nazwy przyjmują zwykle formę „Wykonaj czynność na obiekcie". Oto narracyjne przedstawienie przypadku użycia, który dotyczy zapisywania dokumentu:

> Dokumenty można zapisywać w różnych formatach plików. Kiedy zapisujemy nowy dokument, używany jest domyślny format pliku, chyba że użytkownik wybierze inny. Po zakończeniu operacji „Zapisz dokument" plik reprezentuje dokument dokładnie w takiej formie, w jakiej był przedstawiony użytkownikowi w momencie wykonania operacji.

Punkt widzenia systemu ważny jest zwykle tylko dla twórców aplikacji. Użytkownikom mówi niewiele. Pisząc przypadki użycia, powinniśmy trzymać się perspektywy użytkownika.

Inną alternatywą jest nazywanie i opisywanie przypadków z punktu widzenia naszego edytora tekstu. Operacja „Otwórz dokument" mogłaby zostać opisana jako „Otwórz plik i wczytaj go do bufora tekstowego". Nie zalecamy jednak przyjmowania perspektywy systemowej. Nie jest ona naturalna dla użytkowników i powoduje dziwne wrażenie, jakby to system patrzył na użytkownika z wnętrza komputera i opisywał, co zamierza zrobić.

Przypadki użycia dla naszego edytora tekstu opisują raczej niewielkie fragmenty funkcjonalności. Praktyczna zasada jest taka, że przypadki użycia powinny obejmować takie zakresy, jakie najbardziej odpowiadają użytkownikowi. Poziom szczegółowości również może się zmieniać. Użytkownicy mogą zadowolić się kilkoma ogólnymi zdaniami albo wymagać opisania najdrobniejszych szczegółów, zależy to również od tego, jaka jest ich znajomość opisywanego procesu i jak bardzo jest on złożony. Mimo tych wszystkich różnic w poziomie abstrakcji i szczegółowości, narracyjne przypadki użycia mają jedną wspólną cechę: opisują ogólne możliwości aplikacji w jednym lub dwóch akapitach, posługując się przy tym zwykłym językiem.

Scenariusze

Przedstawione wcześniej narracyjne przypadki użycia opisują ogólne możliwości aplikacji, natomiast scenariusze prezentują konkretne ścieżki, które prowadzą użytkownika do wykonania zadania. Pojedynczy przypadek użycia może zostać wykonany na bardzo wiele sposobów. Poniższy scenariusz, „Zapisz dokument do pliku HTML" pokazuje, czym różni się ta operacja od swojego „rodzica", narracyjnego przypadku użycia „Zapisz dokument do pliku":

Scenariusz: Zapisz dokument do pliku HTML

1. Użytkownik wydaje polecenie zapisu do pliku.
2. Program wyświetla okno dialogowe „Zapis pliku", gdzie użytkownik może oglądać i modyfikować katalog, nazwę pliku i typ dokumentu.
3. Jeżeli plik jest zapisywany po raz pierwszy i użytkownik nie nadał mu nazwy, jest ona konstruowana na podstawie pierwszego wiersza tekstu w dokumencie oraz domyślnego rozszerzenia pliku.
4. Użytkownik wybiera typ dokumentu HTML w opcjach okna dialogowego, co powoduje zmianę rozszerzenia pliku na *„.htm"* w razie potrzeby.
5. Użytkownik może zmienić nazwę pliku i katalog.
6. Użytkownik wydaje programowi polecenie zakończenia zapisu do pliku.
7. Program wyświetla ostrzeżenie, że zapis do pliku w formacie HTML może spowodować utratę części informacji o formatowaniu tekstu. Użytkownik może wybrać porzucenie lub kontynuowanie operacji zapisu.
8. Użytkownik wybiera kontynuację zapisywania dokumentu w formacie HTML.
9. Program zapisuje dokument i wyświetla tekst przeformatowany na nowo. Niektóre informacje o formatowaniu tekstu, na przykład wyliczenia, wcięcia, użyte czcionki, mogą zostać zmienione w stosunku do stanu przed zapisem.

Jeżeli potrzebujemy bardziej konkretnego i szczegółowego wyjaśnienia, jak powinna być wykonywana dana funkcja, tworzymy scenariusze opisujące czynności i informacje dotyczące specyficznych sytuacji. Jeszcze większy poziom szczegółowości oraz nacisk na interakcje między użytkownikiem a systemem pozwalają ująć nam konwersacje.

Konwersacje

Konwersacje przedstawiają interakcje pomiędzy użytkownikiem a systemem w formie *dialogu*. Ich celem jest określenie zakresu odpowiedzialności obu stron: użytkownika, który inicjuje dialog, oraz programu, który monitoruje i odpowiada na czynności wykonywane przez użytkownika. Bardzo szczegółowa forma konwersacji pozwala nam jasno przedstawić odpowiedzi aplikacji na czynności użytkownika.

Każda konwersacja składa się z dwóch oddzielnych części: kolumny zawierającej czynności użytkownika i wprowadzane przez niego dane oraz kolumny prezentującej odpowiedzi aplikacji. Ta kolumna jest pierwszym przybliżeniem listy czynności i odpowiedzi aplikacji. Projektanci będą używać jej zawartości podczas projektowania systemu i przydzielania odpowiedzialności i zadań dla populacji obiektów programistycznych.

Konwersacje i scenariusze rozwijane są zwykle wokół głównego przebiegu akcji, często wybierając jedną ścieżkę spośród wielu alternatywnych.

Konwersacje zawierają kolejne rundy interakcji pomiędzy użytkownikiem a systemem. Każda runda to para obejmująca czynność użytkownika i sekwencję odpowiedzi aplikacji. Rundy mogą być bardziej interaktywne albo też przypominać przetwarzanie wsadowe, w zależności od aplikacji. Na przykład w naszym edytorze tekstu rundami interaktywnymi mogłyby być przechwytywanie i sprawdzanie każdego naciśniętego klawisza, automatyczne poprawianie częstych błędów literowych czy sygnalizowanie użytkownikowi słów, których aplikacja nie znalazła w słowniku. Przeciwieństwem takiego trybu jest na przykład wypełnianie formularzy na stronach WWW. Tam zwykle użytkownik wypełnia większość pól, a dopiero później dane wysyłane są w całości do serwera, który przetwarza je wsadowo.

Tabela 2.4 przedstawia konwersację dotycząca zadania „Zapisz dokument do pliku”.

Konwersacje pozwalają nam zawrzeć więcej szczegółów, których nie ma w narracjach ani scenariuszach. Na przykład, pokazujemy, że nasza aplikacja cały czas informuje użytkownika, jakie pliki o wybranym rozszerzeniu znajdują się w każdym przeglądanym katalogu. Ma to na celu pomóc użytkownikowi w wybraniu unikalnej nazwy pliku.

Wzbogacanie w szczegóły

Projektanci, podobnie jak użytkownicy, muszą dokładnie rozumieć, jak oprogramowanie odpowiada na zdarzenia zewnętrzne. Opisy zawarte w konwersacjach i scenariuszach kształtują naszą pracę projektową. Określone w nich zakresy odpowiedzialności systemu przypisujemy grupom obiektów, które będą współpracować, aby je wykonać.

Konwersacje mogą być bardzo suche, ale mogą też naśladować rozmowę dwóch starych przyjaciół.

Opisy zawarte w konwersacjach i scenariuszach muszą jednak zwykle i tak zostać wzbogacone o dalsze szczegóły, zanim programiści będą mogli przystąpić do budowania działającego systemu, a testerzy do przygotowywania przypadków testowych. Jakie mamy konwencje obsługi błędów? Jakie wartości domyślne powinniśmy przyjąć? Dotychczasowe opisy można wzbogacić między innymi o:

Tabela 2.4. *Konwersacja dotycząca zapisywania pliku opisuje czynności użytkownika i odpowiadające im obowiązki systemu*

Czynności użytkownika	Obowiązki systemu
Wybiera polecenie zapisu pliku	Wyświetla nazwę pliku, który ma być zapisany oraz znajdujące się w aktualnym katalogu podkatalogi i pliki mające takie samo rozszerzenie, jak zapisywany dokument.
	Jeżeli zapis wykonywany jest po raz pierwszy, konstruuje nazwę pliku z rozszerzeniem odpowiadającym domyślnemu formatowi dokumentu.
Zmienia katalog (opcjonalnie).	Wyświetla w oknie dialogowym zawartość nowego katalogu.
Zmienia nazwę pliku (opcjonalnie).	Zmienia zapamiętaną nazwę pliku i wyświetla ją ponownie.
Zmienia format dokumentu (opcjonalnie).	Zmienia zapamiętany format dokumentu.
	Dostosowuje rozszerzenie do konwencji nowego formatu dokumentu i ponownie je wyświetla.
	Ponownie wyświetla zawartość katalogu, pokazując tylko te pliki, które mają wybrane rozszerzenie.
Wybiera przycisk *OK*, aby sfinalizować zapis	Jeżeli wybrany format dokumentu powoduje utratę informacji, wyświetla ostrzeżenie.
	Zapisuje dokument do pliku.
	Ponownie wyświetla zawartość dokumentu, jeżeli format został zmieniony.

- informacje wprowadzane przez użytkowników oraz wartości domyślne, jeżeli niektóre informacje mogą zostać pominięte;
- ograniczenia, jakie muszą być spełnione, zanim zostaną wykonane krytyczne czynności systemu;
- punkty podejmowania decyzji przez użytkownika, które mogą powodować podjęcie innej ścieżki lub scenariusza;
- szczegóły dotyczące kluczowych algorytmów;
- limity czasowe i zawartość każdego znaczącego sprzężenia zwrotnego;
- odnośniki do odpowiednich specyfikacji.

Zamiast przeładowywać tymi szczegółami główny tekst przypadku użycia w formie narracyjnej, scenariusza czy konwersacji, dołączamy do niego listę dodatkowych faktów, ograniczeń, informacji czy nawet obaw. Wzbogacając nasze opisy o powyższe szczegóły, wiążemy przypadki użycia z ograniczeniami projektowymi, pomysłami, a także z innymi specyfikacjami i wymaganiami.

Dokumenty projektowe są łatwiejsze do zrozumienia, jeżeli pisane są z zachowaniem odpowiedniego poziomu szczegółowości (lub ogólności). Dodatkowe szczegóły możemy podać na przykład w załączniku, poza głównym tekstem przypadku użycia.

Alternatywy, adnotacje i inne specyfikacje

Zaletami konwersacji i scenariuszy jest ich prostota i zwięzłość. Czasami jednak chcemy ująć w projekcie bardzo szczegółowe informacje o tym, jak nasze oprogramowanie

wykonuje swoje obowiązki. Na przykład zachowanie aplikacji może zależeć od informacji podanych przez użytkownika lub innych, specyficznych warunków. Aby uchronić nasze konwersacje i scenariusze od przeładowania drobiazgami, umieszczamy je poza głównym tekstem przypadku użycia.

Czynności wyjątkowe

Możemy również uzupełnić nasz przypadek użycia o opis sytuacji anormalnych, stanowiących odchylenie od zwyczajnego trybu wykonywania zadania, tworząc rozdział „Wyjątki":

Wyjątki

Próba nadpisania istniejącego pliku — poinformować użytkownika i poprosić o potwierdzenie chęci nadpisania poprzedniej zawartości pliku.

Wyjątki opisują zarówno nietypowe sytuacje, jak i sposoby ich rozwiązywania. Rozwiązanie możemy opisać jednym lub dwoma zdaniami, jeżeli jest proste. W bardziej złożonych przypadkach możemy odnieść się do innej konwersacji lub scenariusza. W sekcji „Wyjątki" prezentujemy, jak nasze oprogramowanie powinno sobie radzić z oczekiwanymi problemami. W niektórych sytuacjach aplikacja może podjąć odpowiednią reakcję i powrócić do normalnego trybu pracy. Wtedy użytkownik kontynuuje swoją pracę, ale być może na zmienionej ścieżce lub według innego scenariusza. Innym razem jedynym możliwym wyjściem jest przerwanie zadania użytkownika lub nawet pracy aplikacji.

Strategie biznesowe i aplikacji

Odpowiedzi naszego systemu zależą często od jasno określonych reguł biznesowych i strategii (ang. *policy*) aplikacji. Zachowanie programu musi być w zgodzie z takimi strategiami, jak na przykład „powinna istnieć możliwość zapisywania dokumentów w różnych formatach". Stosowne strategie powinniśmy jasno sformułować i zawrzeć je w odpowiednim dokumencie:

Strategie

Nie pozwalaj użytkownikowi zapisać dokumentu do pliku, który jest otwarty przez innego użytkownika.

Jeżeli dokument zapisywany jest po raz pierwszy, skonstruuj i zaproponuj użytkownikowi nazwę pliku na podstawie zawartości pierwszego wiersza tekstu w dokumencie.

Każdy pomysł to nadarzająca się okazja. Nie strać jej!

Nasze rosnące zrozumienie powstającej aplikacji często podsuwa nam nowe pomysły na temat tego, jak można zaprojektować nasz system. Naszą zasadą przewodnią jest „Nie trać okazji!". Zamiast sztywno dzielić nasze czynności i dokumenty na „analizę" i „projekt", zbieramy i dokumentujemy wszelkie informacje, kiedy tylko je napotkamy.

Notatki projektowe

Aby zawrzeć w przypadku użycia wszelkie dodatkowe warunki lub konwencje, które mogą okazać się cenne dla projektanta, dodajemy rozdział „Notatki projektowe”:

Notatki projektowe

Format dokumentu wskazywany jest przez rozszerzenie nazwy pliku. Niektóre formaty mają wspólne rozszerzenie, ale informację o rzeczywistym formacie przechowujemy w deskryptorze formatu pliku. Typowe rozszerzenia:

- *.doc* — pliki w standardowym formacie wszystkich wersji,
- *.rtf* — format pełno tekstowy (ang. *rich text*),
- *.txt* — format tekstu z podziałem na wiersze lub bez,
- *.html* — format hipertekstowego języka znaczników (ang. *Hypertext Markup Language*).

Inne specyfikacje

Układy ekranów, specyfikacje okien, wyciągi z obowiązujących przepisów, wymagania wydajnościowe czy odnośniki do standardów i strategii dostarczają nam szerszy kontekst systemu. Wiążąc je z naszą dokumentacją przypadków użycia, uzyskujemy jeszcze bardziej dogłębny pogląd na oczekiwane zachowanie naszego systemu. Gromadzenie tego typu informacji, choć jest bardzo wartościowe dla projektantów, daje dodatkowo zleceniodawcom okazję do przekonania się, że ich wymagania i problemy są brane pod uwagę.

Słowniki

Pisząc przypadki użycia i inne dokumenty, staramy się konsekwentnie używać spójnego słownictwa. Zbierając specyficzne dla projektu definicje, często używane określenia, zwroty czy nawet żargon w postaci słownika, możemy ułatwić sobie tworzenie jeszcze bardziej jasnych i spójnych specyfikacji:

Dokument — zawiera tekst, podzielony na akapity oraz inne obiekty bitmapowe i graficzne. Tworzony jest za pomocą narzędzia do redagowania tekstu. Jego zawartość można modyfikować przy użyciu poleceń dostępnych w edytorze tekstu.

Obiekt graficzny — może być wyświetlany w dokumencie. Może zostać stworzony w edytorze tekstu lub zaimportowany z innej aplikacji i wstawiony do dokumentu. Jeżeli pozwalają na to właściwości obiektu graficznego, użytkownik może zmieniać jego rozmiary, skalować go itp.

Jak dotychczas, nasze dokumenty nie miały wiele wspólnego z obiektowością. Dopiero kiedy zbierzemy opisy systemu z różnorakich punktów widzenia, możemy pokusić się o próbę reprezentacji ich w ujednoliconej formie — modelu kandydatów na obiekty.

Obiekty konceptualne

Jednym z naszych celów jest zapewnienie łatwej przekładalności naszego projektu na obiektowy język programowania. Pierwszym krokiem w kierunku projektu obiektowego będzie opisanie kluczowych pojęć — zbioru kandydatów na obiekty. Zaczynamy już zagłębiać się w „myślenie obiektowe". Nasi zleceniodawcy rozumieją zwykle jeszcze obiekty na wysokim poziomie ogólności, ponieważ odzwierciedlają one bezpośrednio podstawowe pojęcia dziedzinowe. Ale gdy zaczniemy zagłębiać się w bardziej szczegółowe czynności projektowe, nasze obiekty nabiorą więcej cech związanych z komputerami, stając się coraz bardziej obce dla laików.

Koncentracja na jądrze

Naszym celem jest zbudowanie dobrze zaprojektowanej aplikacji, która działa zgodnie ze specyfikacjami i może dostosować się do niewielkich zmian. Potrzebuje zatem solidnego jądra (ang. *core*). „Jądro" może oznaczać wiele rzeczy:

Co będziemy uważać za „jądro", zależy od tego, na co kładziony jest największy nacisk w naszej aplikacji oraz jak bardzo zależy nam na jej sukcesie.

- kluczowe obiekty, pojęcia i procesy dziedzinowe,
- obiekty implementujące skomplikowane algorytmy,
- infrastrukturę techniczną,
- obiekty obsługujące zadania aplikacji,
- własne obiekty interfejsu użytkownika.

Kandydaci na obiekty mogą dotrwać do włączenia do modelu obiektowego aplikacji bez zmian, ze zmianami lub też mogą zostać odrzuceni gdzieś po drodze.

W naszej aplikacji edytora tekstu obiekty, które reprezentują części dokumentu — czyli klasy takie, jak *Dokument*, *Strona*, *Akapit* czy *Korektor Pisowni* — stanowią jej jądro. Pojawiły się podczas początkowego formowania pojęć.

Dokument

Dokument zawiera tekst oraz inne obiekty wizualne, które reprezentują treść przygotowaną w innych aplikacjach. Dokumenty składają się z sekwencji elementów, na przykład akapitów, obiektów graficznych, tabel. Użytkownik może te elementy formatować i rozmieszczać wizualnie na stronach.

Strona

Strona reprezentuje to, co jest widoczne na wydrukowanej stronie dokumentu. Składa się z akapitów i innych elementów dokumentu, a także opcjonalnych nagłówków i stopek, które zawierają tekst umieszczany odpowiednio na górze i dole strony.

Akapit

Akapit jest elementem dokumentu, który składa się z tekstu lub innych obiektów graficznych. Nowy akapit tworzy się, gdy użytkownik podczas redagowania tekstu naciska klawisz *Enter*. Każdy akapit ma przypisany odpowiedni styl, który steruje wyświetlaniem jego zawartości i określa na przykład odstępy między wierszami tekstu w akapicie.

Korektor pisowni

Korektor pisowni sprawdza, czy słowa w całym dokumencie albo zaznaczonym jego fragmencie znajdują się w słowniku ortograficznym, który dołączony jest do aplikacji edytora tekstu albo zostały dodane przez użytkownika do jego własnego słownika. Informuje użytkownika o każdym błędnie wpisanym słowie i daje możliwość poprawienia, zignorowania albo dodania słowa do słownika użytkownika.

Jeżeli te obiekty przetrwają okres kandydowania i wejdą w poczet nowo stworzonych obiektów projektowych, będzie to oznaczać, że reprezentują odpowiedzialność aplikacji w taki sposób, który odpowiada naszym celom projektowym.

Obsadzanie ról — projektowanie badawcze

Jeżeli sednem analizy jest zachowanie aplikacji, to sedno projektowania stanowią obiekty, dzięki którym możemy osiągnąć pożądane zachowanie. Projektując, wytyczamy i brukujemy drogi, którymi nasza aplikacja będzie później postępować. Podobnie jak dobry urbanista, projektant bierze pod uwagę, w jakich kierunkach będzie się rozrastać jego oprogramowanie, jak będzie się zmieniać, gdzie są najbardziej prawdopodobne punkty nagromadzenia zmian.

W rozdziale 3., pod tytułem „Szukanie obiektów", przedstawiamy strategie identyfikowania i charakteryzowania obiektów projektowych.

Istnieje znacząca różnica pomiędzy obiektami konceptualnymi a projektowymi. Chociaż oba rodzaje opisują przedmioty, dokumenty wysokiego poziomu ignorują szczegóły, często bardzo istotne. Nie jest to jednak pominięcie przypadkowe. Obiekty konceptualne i zakresy odpowiedzialności systemu określają ramy pracy, która pozostała do wykonania. Projektując, tworzymy model obiektów, które współpracują, aby osiągnąć określone we wcześniejszej fazie cele.

„Większość naszych modeli myślowych jest intuicyjna i w dużym stopniu podświadoma, jednak istotą zarówno nauki, jak i ekonomii jest tworzenie sprecyzowanych, szczegółowo określonych modeli, które możemy dzielić i omawiać z innymi".
Trygve Reenskaug

Projektanci sprawdzają, czy obiekty konceptualne wnoszą coś wartościowego do systemu. Są one tylko kandydatami i mogą zostać odrzucone, jeżeli okaże się, że ich wartość lub znaczenie są znikome. Może się jednak okazać, że zostaną włączone do projektu, a nawet staną się jego ważnymi elementami. W naszej przykładowej aplikacji edytora tekstu, *Dokument* stanowi wartościowy obiekt konceptualny. Przypisujemy mu odpowiedzialność za przechowywanie tekstu i jego struktury w postaci kolekcji *Akapitów*, które rozmieszczone są na *Stronach*. Podobnie, *Akapit* zawiera *Tekst*, który składa się ze *Słów*.

Przy dalszym badaniu okazuje się, że obiekt *Akapit* to prawdziwa projektowa kopalnia złota. Możemy wyobrazić sobie *Akapity* złożone z *Tekstów* i rozmaitych innych obiektów, reprezentujących grafikę, rysunki, wykresy czy nawet treść dostarczaną przez inne źródła. *Akapity* rozdzielone są obiektami *Podział Akapitu*. Obiekt *Tekst* zawiera znaki, które składają się na obiekty *Słowo*, a te z kolei — na obiekty *Zdanie*.

Dostrzegamy opisy odpowiedzialności w różnorodnie sformułowanych opisach i przekształcamy je na odpowiednio zdefiniowane obiekty projektowe. Gdy dodajemy do nich nasze własne pomysły, model staje się bardziej kompletny i szczegółowy.

Podczas wprowadzania tekstu obiekt *Analizator Składniowy* formuje *Tekst* ze *Słów*. *Słowa* mają lokalizację w dokumencie — pozycję początku i końca — oraz *Znaki*, stanowiące ich treść. Granice *Słowa* określają odstępy, znaki przestankowe lub elementy nietekstowe. Po utworzeniu każde słowo przekazywane jest do *Korektora Pisowni*, który sprawdza jego poprawność.

Często obiekty konceptualne i wstępni kandydaci na obiekty stanowią wygodne „mięso armatnie" w projekcie — ulegają przeróżnym transformacjom i podziałom w kolejnych fazach projektowania. Rzadziej zdarza się, że kandydaci przechodzą przez analizę i projektowanie bez większych zmian przydzielonych obowiązków. Każdy obiekt, który pozostanie ostatecznie w projekcie, musi mieć jasno określoną rolę i odpowiedni zakres odpowiedzialności. Najczęściej nie są one jeszcze wyrażane dostatecznie precyzyjnie w omówionych dotychczas dokumentach analitycznych.

Doświadczeni projektanci, kiedy słyszą określenie „wymagania", natychmiast myślą o obiektach i ich odpowiedzialności. Zwykle szybko też uzupełniają je o dodatkowe zakresy, które dopełniają zachowanie kandydata. W ten sposób dokonuje się przeskok od surowego pomysłu do dobrze zdefiniowanego obiektu. Doświadczenie pomaga im też wypełniać dziury w powstającym modelu obiektowym za pomocą wymyślanych na bieżąco pomysłów i rozwiązań programistycznych. Czasem takie przeskoki mogą się zbić z tropu kogoś mniej doświadczonego w myśleniu obiektowym.

Na przykład, przyglądając się dokładniej *Korektorowi Pisowni*, możemy dojść do wniosku, że aby mógł wypełniać powierzony mu zakres odpowiedzialności rozpoznawania niepoprawnej pisowni, możemy zaprojektować go w ten sposób, że podstawowe formy słów będzie przechowywał w obiekcie *Słownik Ortograficzny* oraz stosował do nich odpowiednie reguły odmiany przez liczby czy czasy. Jest mało prawdopodobne, że *Korektor Pisowni* przetrwa okres projektowania jako pojedynczy obiekt. Raczej okaże się, że w kolejny fazach przekształci się on w zbiór współpracujących obiektów, być może nawet w podsystem.

Karty CRC

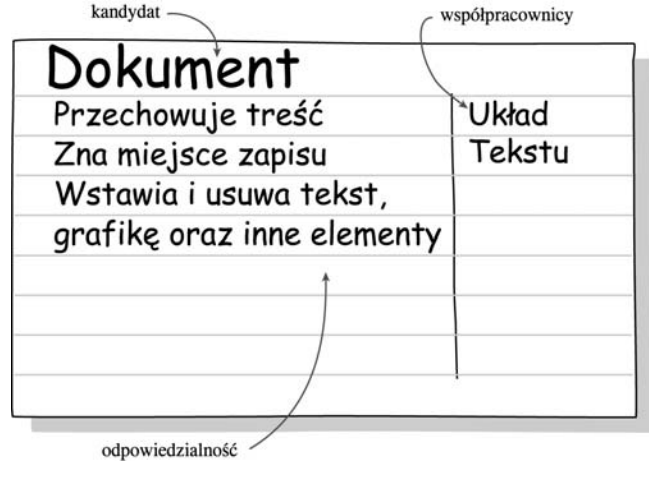
Początkowe pomysły na temat kandydatów, zarówno na obiekty, jak i na role, zapisujemy na kartach CRC. Skrót *CRC* oznacza dla nas *Candidates, Responsibilities, Collaborators*, czyli kandydaci, odpowiedzialność, współpracownicy. Takie karty to wygodne, mało techniczne narzędzie, wspomagające badanie początkowych pomysłów projektowych. Na jednej stronie karty CRC zapisujemy nieformalny opis celu istnienia każdego kandydata oraz stereotypy jego ról (zobacz rysunek 2.6).

Chociaż karty CRC zostały wynalezione po to, by opisywać klasy, ich odpowiedzialność i współpracowników, my zalecamy poszukanie najpierw *kandydatów*. Decyzje, jak przedstawić je za pomocą klas, podejmiemy później — kiedy już przekonamy się, że nasze pomysły są warte zachowania.

Rysunek 2.6. *Charakterystyka kandydata na jednej ze stron karty CRC*

Przechodząc do bardziej szczegółowego opisu, odwracamy kartę CRC na drugą stronę i zapisujemy tam zakresy odpowiedzialności kandydata: co powinien „wiedzieć" i „robić" (zobacz rysunek 2.7). Opis odpowiedzialności mówi nam, jakie informacje powinien przechowywać obiekt oraz jakie zadania powinien umieć wykonać. Współpracownicy to te obiekty, z których zakresów odpowiedzialności korzysta nasz obiekt podczas wypełniania swoich własnych obowiązków.

W rozdziale 4., pod tytułem „Odpowiedzialność", omówimy sposoby identyfikowania i przydzielania odpowiedzialności do właściwych kandydatów.



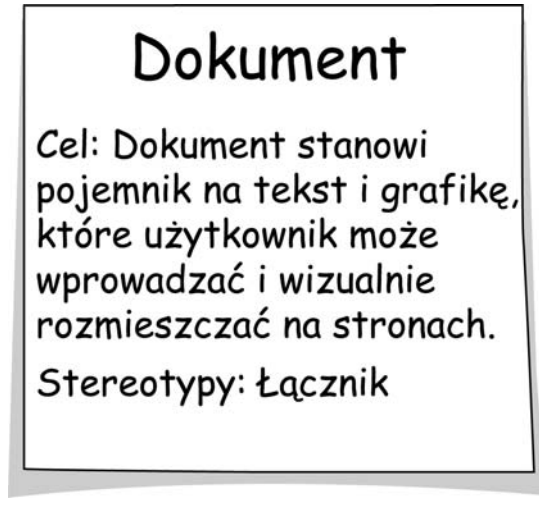
Rysunek 2.7. *Druga strona karty CRC zawiera opis odpowiedzialności i współpracowników*

Karty sprawdzają się jako narzędzie, ponieważ są wygodne, łatwe w użyciu i nie wymagają skomplikowanej techniki. Można je przesuwać, kreślić i przerabiać. Ponieważ nie inwestujemy w ich stworzenie dużo czasu, łatwiej nam zdecydować na wyrzucenie do kosza karty, która okazała się chybionym pomysłem. Służą do zapisywania naszych początkowych pomysłów i nie stanowią trwałego elementu projektu.

Może nas kusić, aby użyć komputera, ale nie dajmy się zwieść iluzji, że nasi kandydaci będą lepiej opisani, ponieważ karta stworzona na komputerze wygląda ładniej. W tej fazie, badanie różnych opcji powinno być zabawą, łatwą i tanią.

Ponieważ karty są niewielkie i istnieją na zewnątrz komputera, możemy je łatwo rozłożyć na stole i objąć wzrokiem wiele z nich na raz (być może nawet wszystkie). Możemy je przesuwać, grupować, zmieniać ich układ, aby lepiej uwidocznić nowe pomysły. Wolnymi miejscami możemy oznaczać te obszary projektu, w których wciąż brakuje nam pomysłów lub rozwiązań.

Nie ma oczywiście sensu upierać się przy użyciu kart. Jeżeli lepiej pracuje nam się na czystych kartkach papieru od drukarki albo małych, żółtych karteczkach, powinniśmy ich używać. Jeśli mamy pod ręką tablicę, możemy wykorzystać ją do naszkicowania ogólnego schematu. Kandydatów możemy skrótowo charakteryzować na samoprzylepnych karteczkach, których układ łatwo zmienić w każdej chwili (zobacz rysunek 2.8).



Rysunek 2.8. *Żółte karteczki są łatwiejsze w użyciu w porównaniu do „kart" CRC*

Oczywiste obiekty konceptualne, które identyfikujemy na kartach CRC na początku projektowania, to tylko jedna część łamigłówki. Prawdziwym wyzwaniem dla naszej kreatywności jest znajdowanie rozwiązań nieintuicyjnych. Stanowią one oznakę elastycznego i dobrze przemyślanego programu. To właśnie ich szukamy podczas projektowania.

Rozwiązania — używanie wzorców

Możemy zyskać całkiem znaczące korzyści, jeżeli wiemy, gdzie szukać gotowych do zaadaptowania rozwiązań. Nasze umiejętności projektowe możemy wspomóc, znajdując dobre wzorce i ucząc się, gdzie i kiedy można je zastosować. Używanie rozwiązań, które dowiodły swojej użyteczności w wielu różnych kontekstach, pomaga nam wypełnić braki w naszym myśleniu. Przystępując do rozwiązywania nowego problemu, mamy już w głowie „wydeptane" ścieżki sprawdzonych rozwiązań.

Dobrzy projektanci ułatwiają sobie pracę, przystosowując do swoich potrzeb wypróbowane wcześniej rozwiązania. Analizują różne projekty i wykorzystują doświadczenia swoje oraz innych.

Powinniśmy teraz dokładnie przemyśleć kluczowy aspekt naszego edytora tekstu: jak obsłużyć wielką ilość operacji, które może wykonać użytkownik. Edytor tekstu jest, w dosłownym znaczeniu, „odpowiedzialny" za obsługiwanie żądań redagowania,

wstawiania, wyszukiwania i formatowania tekstu, zapisywania i otwierania plików, wycinania, kopiowania i wklejania zaznaczonych fragmentów, drukowania, przeglądania, sprawdzania pisowni i gramatyki i wielu innych.

- Jak powinniśmy wykonywać te czynności?
- Każda pozycja w menu reprezentuje żądanie, aby nasz edytor tekstu wykonał jakąś operację. Czy (i jak) można cofnąć jej skutki?
- Wiele czynności dotyczy określonej części dokumentu. W jaki sposób przechowujemy informacje o tym, na jakiej części operujemy?

Problem określenia sposobu sterowania jest głównym problemem w większości aplikacji.

Na przykład, operacja „wytnij" usuwa podświetlony tekst i umieszcza go w specjalnym buforze. Natomiast operacja „pogrub" powoduje użycie czcionki półgrubej w podświetlonym tekście lub, w wypadku braku podświetlenia, słowie zawierającym aktualną pozycję kursora. Zapisywanie dokumentu oznacza otwarcie odpowiedniego pliku do zapisu i umieszczenie w nim właściwych informacji.

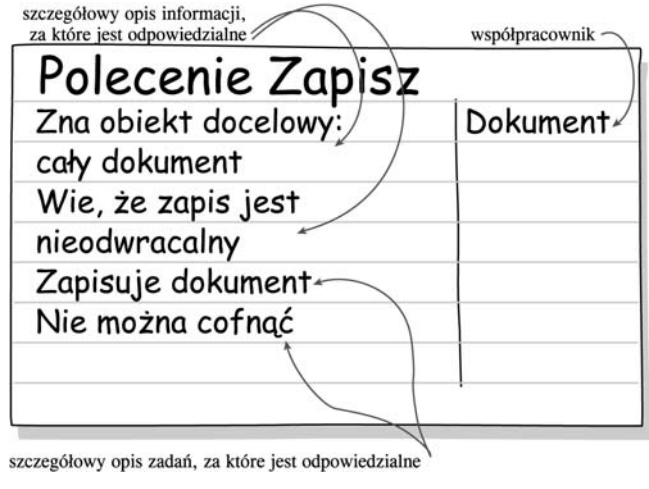
Wzorzec projektowy *Polecenie* (ang. *Command*), opisany w książce *Wzorce projektowe: tworzenie oprogramowania obiektowego wielokrotnego użytku*, pozwala zamienić operację w obiekt. Każda czynność może być reprezentowana przez oddzielny obiekt, odgrywający rolę *Polecenia*. Możemy z nich stworzyć hierarchię dziedziczenia. Wzorzec *Polecenie* w postaci opisanej we wspomnianej książce jest bardzo ogólny i musimy go dostosować do potrzeb naszej aplikacji edytora tekstu. Aby go użyć, musimy zacząć myśleć o wykonywaniu i cofaniu wszystkich operacji naszego edytora, jako o odpowiednich obiektach. Co to właściwie oznacza? Jak możemy dopasować wszystkie operacje do formy opisanej we wzorcu *Polecenie*?

Doświadczony projektant zwykle prawie natychmiast rozpoznaje potrzebę użycia wzorca projektowego *Polecenie*. Nowicjusz prawdopodobnie prototypowałby kilka różnych sposobów rozwiązania problemu określania różnych typów zachowań „polecenia", zanim odkryłby, że wzorzec dostarcza spójnego i eleganckiego rozwiązania. Odkrywając, że wzorzec projektowy dobrze pasuje do naszego problemu, korzystamy z doświadczeń projektowych wielu innych osób.

Zaczynamy od zadeklarowania, że każdy obiekt grający rolę *Polecenie* jest odpowiedzialny za wykonanie określonej czynności (zobacz rysunek 2.9). Niewątpliwie nasz projekt będzie potrzebował wielu różnych rodzajów *Poleceń* i klas implementujących je, aby modelować każdą z ogromnej liczby operacji, które może wywołać użytkownik naszego edytora tekstu. Każdy rodzaj *Polecenia* jest również odpowiedzialny za cofnięcie wykonanej przez siebie czynności, dzięki czemu uzyskujemy obsługę opcji „cofnij" w całej aplikacji. Zdefiniujemy odpowiedzialność każdego rodzaju polecenia. Implementując nasz projekt, stworzymy bazową klasę *Polecenie*, w której zadeklarujemy, że każdy obiekt tej klasy potrafi „wykonać czynność" oraz „cofnąć" jej efekty. Dodatkowo, każde *Polecenie* przechowuje informację o obiekcie, który stanowi jego cel. W edytorze tekstu obiektem docelowym jest zwykle część tekstu, której dotyczy dane *Polecenie*.

Każdy rodzaj obiektu *Polecenie* obsługuje ten sam zakres odpowiedzialności, przypisany do roli *Polecenie* — ale na bardzo różne sposoby. Na przykład wykonanie *Polecenia Zapisz* oznacza zapisanie zawartości dokumentu do pliku. Zapis nie jest odwracalny i nigdy nie będziemy wykonywać cofania zapisu. Obiektem docelowym w *Poleceniu Zapisz* jest cały dokument. Ponieważ więc w celu wypełnienia odpowiedzialności *Polecenia Zapisz* następuje współpraca pomiędzy obiektami, umieszczamy *Dokument* wśród współpracowników (zobacz rysunek 2.10).

Rysunek 2.9. *Odpowiedzialność obiektu Polecenie jest określona bardzo ogólnie*

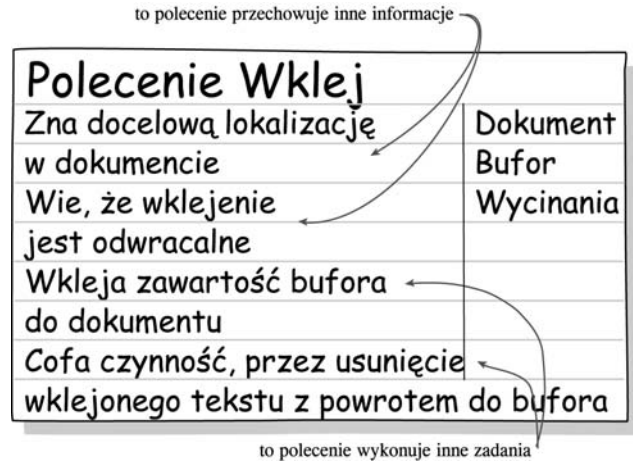


Rysunek 2.10. *Odpowiedzialność obiektu Polecenie Zapisz jest określona bardziej precyzyjnie*

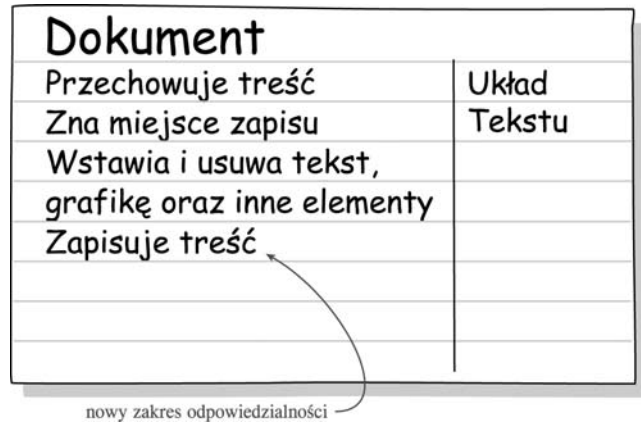
Możemy stworzyć kartę CRC dla każdego rodzaju polecenia, opisując na niej jego zakres odpowiedzialności. Na przykład *Polecenie Wklej* wypełnia swoją rolę, umieszczając tekst w dokumencie w aktualnej pozycji kursora lub usuwając go z powrotem, w razie wycofania (zobacz rysunek 2.11).

Kiedy określając odpowiedzialność każdej klasy, część zadań delegujemy do innych klas, musimy uzupełnić ich karty CRC (na przykład kartę *Dokumentu*, patrz rysunek 2.12).

W tym przypadku udało nam się zastosować wzorzec projektowy *Polecenie*, aby pokazać, jak bardzo możemy sobie ułatwić konkretny problem, korzystając ze sprawdzonych rozwiązań. Jednak czasami problem może się okazać trudniejszy. Często musimy ponownie przemyśleć nasze pomysły projektowe, poprzesuwać karty na stole, precyzyjniej określić role obiektów czy dodać współpracowników. Bywa, że nie możemy posunąć się do przodu, ponieważ nie potrafimy dostrzec rozwiązań. Co gorsza, zdarza się także, że nabywając doświadczenia w projektowaniu, nasze wcześniejsze rozwiązania zaczynamy uważać za brzydkie lub nieefektywne.



Rysunek 2.11. *Polecenie Wklej również wypełnia zakres odpowiedzialności obiektu Polecenie*



Rysunek 2.12. *Uściślając projekt, dodajemy nowe zakresy odpowiedzialności i współpracowników*

We wczesnych fazach nasze pomysły są bardzo płynne. Radykalne zmiany są łatwe do wprowadzenia i często pożądane. Możemy przenosić odpowiedzialność, zmieniać współpracowników, poprawiać role obiektów i wprowadzać nowych „graczy" bez wielkiego wysiłku. Rozważając różne możliwości, nabieramy pewności i przekonania, że nasze rozwiązanie jest dobre.

Poszukiwanie rozwiązania

Jak dokonać wyboru spośród wielu akceptowalnych alternatyw projektowych? Możemy przyjąć następującą prostą strategię:

1. Jeżeli nie mamy wstępnego pomysłu, tworzymy rozwiązanie, które działa w zadowalającym stopniu.
2. Badamy jego ograniczenia i zalety. Każdą ocenę powinniśmy oprzeć na porównaniu do przynajmniej jednego rozwiązania alternatywnego.
3. Wybieramy rozwiązanie, które najwięcej wnosi do spójności projektu.

Naszym podstawowym narzędziem projektowym jest zdolność abstrakcji. Pozwala nam ona korzystać efektywnie z hermetyzacji, dziedziczenia, współpracy i innych technik obiektowych.

4. Staramy się nie przepracować nad rozwiązaniem.
5. Dopasowujemy rozwiązanie do znanych nam wzorców projektowych.
6. Zapożyczamy i adaptujemy sprawdzone pomysły projektowe i archetypy.
7. Kiedy rozwiązanie staje się nieeleganckie, powinniśmy jeszcze raz rozważyć wcześniejsze decyzje i, w razie potrzeby, nie bać się wybrać innej ścieżki.
8. Jeżeli nie mamy czasu, nie czekajmy na olśnienie. Abstrakcji i elegancji nie da się wymusić.

Przeskakiwanie od pomysłów do szczegółów

Jednym ze sposobów pilnowania, czy projektując nie zbaczamy w manowce, jest testowanie naszego projektu za pomocą szczegółów. Rozwiązanie, które brzmi nieźle w ogólności, może zostać rozsadzone przez szczegóły, które się w nim nie mieszczą. W naszych scenariuszach i konwersacjach sformułowaliśmy bardzo istotne opisy działania aplikacji. Możemy sprawdzić poprawność naszego projektu za pomocą dodatkowych warunków, które odkrywamy, zagłębiając się w szczegóły. Modelujemy na wysokim poziomie, a później poświęcamy czas na opracowanie szczegółów. W pracy projektowej przenosimy się między modelowaniem i abstrahowaniem a dopracowywaniem i uściślaniem.

Czasami zagłębienie się w szczegóły pozwala łatwiej dostrzec abstrakcje. Przeglądając wszystkie możliwe polecenia w edytorze tekstu, możemy nabrać lepszego poglądu na to, co mają wspólnego i dostrzec lepszy sposób zunifikowania ich.

Te szczegóły możemy znaleźć w naszych dokumentach analitycznych. Powracając do konwersacji dotyczącej zadania „Zapisz dokument do pliku", zauważamy wiele zakresów odpowiedzialności, które należałoby przypisać obiektom (tabela 2.5). Konwersacje są świetnym źródłem opisów odpowiedzialności obiektów. Wcześniej koncentrowaliśmy się na istocie funkcjonalności aplikacji, więc celowo ignorowaliśmy szczegóły. By jednak nasz projekt był kompletny, musimy uporządkować zakresy odpowiedzialności systemu i wiele innych szczegółów, wymyślając wiele obiektów projektowych i planując współpracę pomiędzy nimi. Na rysunku przedstawiamy w nawiasach wstępne propozycje przydziału zakresów odpowiedzialności systemu do jednego lub wielu potencjalnych kandydatów.

Po wykonaniu początkowej próby określenia zakresów odpowiedzialności na podstawie scenariuszy i konwersacji i przypisania ich do obiektów musimy skonstruować bardziej kompletne rozwiązanie i ocenić jego zalety w stosunku do alternatyw. Będziemy szukać szczegółowych odpowiedzi, jak obiekt wypełnia odpowiedzialność określoną na wysokim poziomie:

- Co obiekt robi? Jaki wnosi wkład do odpowiedzialności określonej na wysokim poziomie?
- W jaki sposób współpracuje z innymi obiektami, które także biorą udział w wypełnianiu tej odpowiedzialności?
- Jakie informacje musi przechowywać?
- Jakie komunikaty wysyła do innych? W jakiej kolejności?
- Jakie są ich argumenty? Co jest zwracane z każdego żądania?

Tabela 2.5. *Odpowiedzialność systemu jest przypisana do obiektów*

Czynności użytkownika	Obowiązki systemu
Wybiera polecenie zapisu pliku	Wyświetla nazwę pliku, który ma być zapisany oraz znajdujące się w aktualnym katalogu podkatalogi i pliki mające takie samo rozszerzenie, jak zapisywany dokument. (Przydzielamy koordynację wysokopoziomową *Kontrolerowi Okna Dialogowego Zapisu*, którym kieruje *Polecenie Zapisz*.)
	Jeżeli zapis wykonywany jest po raz pierwszy, konstruuje nazwę pliku z rozszerzeniem odpowiadającym domyślnemu formatowi dokumentu. (Przypisanie nowego dostawcy usług?)
Zmienia katalog (opcjonalnie)	Wyświetla zawartość nowego katalogu w oknie dialogowym. (*Kontroler Okna Dialogowego Zapisu* współpracuje z *Katalogiem*.)
Zmienia nazwę pliku (opcjonalnie)	Zmienia zapamiętaną nazwę pliku i wyświetla ją ponownie. (*Kontroler Okna Dialogowego Zapisu* współpracuje z *Dokumentem*, a ten z *Plikiem*.)
Zmienia format dokumentu (opcjonalnie)	Zmienia zapamiętany format dokumentu. (*Dokument*)
	Dostosowuje rozszerzenie do konwencji nowego formatu dokumentu i ponownie je wyświetla. (*Kontroler Okna Dialogowego Zapisu* współpracuje z jakimś obiektem, który przechowuje rozszerzenia przypisane do typów dokumentów — prawdopodobnie *Menedżer Plików*?)
	Ponownie wyświetla zawartość katalogu, pokazując tylko te pliki, które mają wybrane rozszerzenie. (*Kontroler Okna Dialogowego Zapisu*)
Wybiera przycisk *OK*, aby sfinalizować zapis	Jeżeli wybrany format dokumentu powoduje utratę informacji, wyświetla ostrzeżenie. (*Kontroler Okna Dialogowego Zapisu*)
	Zapisuje dokument do pliku. (*Polecenie Zapisz* współpracuje z *Dokumentem*.)
	Ponownie wyświetla zawartość dokumentu, jeżeli format został zmieniony. (*Polecenie Zapisz* współpracuje z *Kontrolerem Przetwarzania Tekstu*.)

Zaprojektujemy interakcje między obiektami i dalej będziemy rozdzielać ich odpowiedzialność. Stworzymy dodatkową dokumentacje projektową i rysunki. Sporządzimy diagramy opisujące typowe modele współpracy oraz pokazujące klasy, które implementują nasz projekt. Wreszcie projekt znajdzie swój ostateczny wyraz w kodzie.

W rozdziale 7., zatytułowanym „Opisywanie współpracy”, prezentujemy różne możliwości dokumentowania kluczowych interakcji i współpracy między obiektami, zarówno przy użyciu technik nieformalnych, jak i diagramów UML.

Przed premierą — dopracowywanie projektu

Jako projektanci odgrywamy znaczącą rolę w realizacji gładko przeprowadzonej produkcji oprogramowania. Projekt badawczy to tylko początek. Kiedy już „przemielimy” nasze początkowe pomysły i zorientujemy się, w którą stronę należy pójść, musimy systematycznie i dogłębnie dopracować każdy aspekt naszego projektu. Powinniśmy zadać wiele pytań, a na ich podstawie podjąć decyzje i wprowadzić poprawki, które będą miały duże konsekwencje. Oto pytania, na które powinniśmy sobie odpowiedzieć:

- Jakie style współpracy będą dominować?
- Jak nasz projekt potrafi zaadaptować się do zmieniających się potrzeb użytkownika?
- Gdzie powinna być wbudowana możliwość przyszłościowych rozszerzeń i modyfikacji?
- Co zrobić, aby nasz projekt był bardziej spójny, przewidywalny i łatwiejszy do zrozumienia?
- Jak bardzo odporne na zakłócenia musi być nasze oprogramowanie?

Pozostało jeszcze wiele pracy! Strategie pracy nad tymi pytaniami stanowią serce tej książki. Omówmy skrótowo choćby niektóre z nich.

Określanie stylów współpracy i sterowania

Bardzo ważną decyzją jest jak najlepsze rozmieszczenie sterowania i odpowiedzialności za podejmowanie decyzji pomiędzy współpracowników. Pytania o sterowanie pojawiają się w wielu obszarach:

Rozdział 5., „Współpraca", pokazuje, jak stworzyć model współpracy między obiektami. W rozdziale 6., „Styl sterowania", opisujemy strategie rozwijania spójnych wzorców współpracy oraz „centrów sterowania" w aplikacji.

- Jak sterujemy i koordynujemy przypadki użycia i zdarzenia spowodowane przez użytkownika?
- Jakie są architektoniczne ograniczenia współpracy i stylów sterowania?
- Gdzie umieszczamy odpowiedzialność za podejmowanie decyzji?
- Jak powinna być rozwiązana obsługa wyjątków i przywracanie stabilności po błędach?

Odpowiedzi na powyższe pytania mają ogromny wpływ na to, jak odpowiedzialność jest rozłożona pomiędzy częściami modelu. Naszym celem jest zaprojektowanie spójnych, przewidywalnych wzorców interakcji. Obiekty typu *Polecenie* są miejscem sterowania dla czynności użytkownika. Podjęcie takiego wyboru projektowego skutkuje pojawieniem się jasnego wzorca do obsługi operacji spowodowanych przez użytkownika. Wzorzec projektowy *Polecenie* daje nam abstrakcję tego, jak obsługiwać te czynności oraz jak dodawać nowe rodzaje kontrolerów takich czynności. Dodawanie nowych operacji powinno być całkiem proste i ograniczać się do tworzenia nowych podklas *Polecenie*.

Ale istnieje jeszcze wiele innych miejsc, w których będziemy musieli zastanowić się, jaki styl sterowania przyjąć. Na przykład w korektorze pisowni musimy podjąć decyzje, jak reprezentować reguły fleksyjne oraz jak wykrywać i raportować błędy ortograficzne. Sposób sterowania zapisem i odczytem dokumentów wymaga dogłębnego przemyślenia. Tworzenie stylu współpracy lub sterowania wymaga zadecydowania, jak rozdzielić sterowanie pomiędzy współpracowników oraz jakie wzorce współpracy powielać. Nasze wybory dotyczące przydziału odpowiedzialności za sterowanie i wykonywanie poleceń mogą prowadzić do rozwiązań bardziej scentralizowanych lub bardziej rozproszonych.

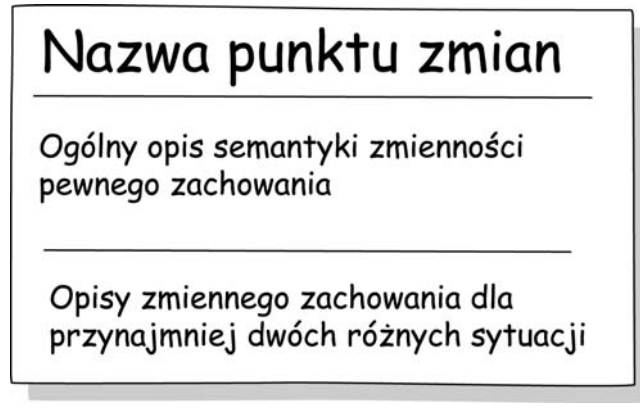
Przewidywanie zmienności użytkowników

Typowy edytor tekstu obsługuje wiele różnych stylów pracy użytkowników, ich preferencji i trybów interakcji. Bardzo wiele opcji jest pod bezpośrednią kontrolą użytkownika, od sposobu wyświetlania całego dokumentu do szczegółów w rodzaju częstości automatycznego zapisu czy ignorowania niektórych błędów literowych. W tego typu aplikacji najlepszy sposób obsługi tej ogromnej liczby zmiennych stanowi podstawowe zagadnienie projektowe. Nasz edytor tekstu podejmuje w czasie działania bardzo wiele decyzji „taktycznych" na podstawie aktualnych ustawień i preferencji.

W rozdziale 9., pod tytułem „Elastyczność", dyskutujemy o projektowaniu aplikacji, które potrafią się dostosować do użytkownika i przewidują pewien zaplanowany stopień zmienności. Wzorce oraz implementowanie zaprojektowanych „punktów zaczepienia" stanowią klucz do zwiększania elastyczności.

Projektowanie a elastyczność i rozszerzalność

Elastyczność nie jest samoistną cechą żadnego projektu. Aby ją uzyskać, wzbogacamy zachowanie naszych obiektów o dodatkowe operacje, umożliwiające rozszerzanie i rekonfigurację. Zaczynamy od opisania obszarów, w których najbardziej zależy nam na elastyczności. Krótko charakteryzujemy, jak może się zmieniać zachowanie, kiedy powinno to następować oraz pokazujemy na przykładach istotę tych zmian. Zwarty opis zmienności umieszczamy na *karcie punktu zmian* (ang. *hot spot card*, zobacz rysunek 2.13). Podobnie jak w przypadku kart CRC, ilość miejsca na takiej karcie jest ograniczona. Pozwala to nam skupić się na samej istocie.



Rysunek 2.13. *Karta punktu zmian opisuje i demonstruje zmienność*

Mając charakterystykę tego, kiedy i jak nasze oprogramowanie powinno zachowywać się elastycznie, możemy użyć jednej techniki projektowej lub kombinacji kilku: abstrakcji, klasyfikacji, złożenia, dziedziczenia czy parametryzacji.

Nasz edytor tekstu musi obsługiwać wiele punktów zmian związanych z użytkownikami. Dodatkowo, oczekujemy obsługi nowych cech i możliwości — nowych rodzajów grafiki, nowych formatów dokumentów, nowych i bardziej złożonych sposobów sprawdzania pisowni czy gramatyki, nowych szablonów. Nasze oprogramowanie musi być elastyczne od samego początku.

Istnieje wiele technik, których możemy użyć, by nasz projekt mógł obsługiwać zaplanowaną zmienność. Od najprostszych, jak sprawdzanie wartości odpowiednich parametrów, do najbardziej wyrafinowanych. Zachowanie obiektu możemy konfigurować, przekazując do metod odpowiednie parametry. Możemy na przykład zaprojektować obiekt, który zapamiętuje informacje raz uzyskane i później z nich korzysta. Obiekt może wypełniać swój zakres na różne sposoby, jeżeli część odpowiedzialności oddeleguje do innych dostawców usług, których możemy odpowiednio konfigurować. Na przykład idealnie byłoby, gdyby nowy format dokumentu można było dodać do aplikacji, podłączając jako „wtyczkę" nowego dostawcę usług. Wtyczka potrafiłaby przy zapisie wygenerować na podstawie dokumentu odpowiedni plik oraz odtworzyć dokument z pliku przy odczycie.

Nie ma jednej zawsze najlepszej strategii obsługi zmienności. Zwykle zalecamy proste rozwiązania, które pozwalają łączyć się z innymi w celu budowania większych systemów. Złożoność możemy dodać zawsze, jeżeli potrzebujemy bardziej uniwersalnego rozwiązania. Przeprojektowywanie jest jedną z istotnych części konserwowania dłużej działających systemów.

Projektowanie a niezawodność

Duża część złożoności projektów programistycznych wynika z sytuacji, które, choć przewidywane, nie są „normalne". Podczas pracy z edytorem tekstu użytkownik ma bardzo dużo okazji, by nie dostarczyć wszystkich wymaganych informacji lub wykonać operację w sposób nieprzewidziany przez autorów.

Rozdział 8., „Niezawodna współpraca", pokazuje strategie zwiększania zdolności aplikacji do obsługi sytuacji wyjątkowych oraz przywracania stabilności działania programu po błędach.

Co się stanie, jeżeli spróbujemy nadpisać dokumentem istniejący plik? Jak nasza aplikacja powinna reagować na polecenie zapisania dokumentu w formacie, które spowoduje utratę części informacji? Akurat te pytania są dość łatwe, ale jest wiele innych. Jak zareagować na sytuację, w której aplikacji brakuje pamięci do działania lub miejsca na dysku do zapisu? A jeżeli podczas odczytu dokumentu z pliku okaże się, że części danych aplikacja nie potrafi zinterpretować? To są już trudniejsze problemy. Nasi użytkownicy oczekują, że aplikacja poprawnie obsłuży te sytuacje, jeżeli to możliwe lub pokaże odpowiednią informację w przeciwnym wypadku.

Nasze obiekty muszą być zaprojektowane w taki sposób, by odpowiedzialnie i konsekwentnie (a także najlepiej, jak potrafią) obsługiwały sytuacje wyjątkowe. Projektowanie spójnych strategii obsługi wyjątków oraz systematyczne umieszczanie jej w kontrolerach, dostawcach usług i innych „odpowiedzialnych" obiektach powoduje, że nasze oprogramowanie radzi sobie z takimi sytuacjami bardziej przewidywalnie.

Tworzenie przewidywalnych, spójnych i zrozumiałych projektów

Istotą dobrego projektu jest przewidywalność i spójność. Złożoność naszej aplikacji opanowujemy przez tworzenie spójnych rozwiązań. Nie chcemy, aby nasz projekt przerażał innych. Dlatego też, jeżeli uda nam się elegancko rozwiązać problem projektowy, szukamy innych miejsc, w których można by zastosować podobny pomysł. Dla każdej

w miarę złożonej aplikacji można by stworzyć nieskończenie wielką liczbę różnych projektów. Na spójność i zrozumiałość projektu wpływa wiele czynników:

- Obiekty pogrupowane są w sąsiedztwa i podsystemy.
- Komunikacja między sąsiedztwami jest ograniczona.
- Żaden obiekt nie wie, nie robi i nie kontroluje „zbyt dużo".
- Obiekty działają zgodnie z przydzielonymi rolami.
- Jeżeli rozwiązanie się sprawdza, jego warianty są stosowane tam, gdzie to możliwe.
- Istnieje tylko kilka wzorców współpracy, które powtarzają się w całym projekcie.

Pracując nad stworzeniem konsekwentnego, przewidywalnego projektu, powinniśmy zachować równowagę między wieloma siłami. Nie ma na to uniwersalnej recepty. Musimy rozważać różnorodne kompromisy i wkładać dużo wysiłku w zachowanie spójności całego projektu. W niektórych przypadkach architektura systemu albo schemat aplikacji wymuszają określone style współpracy lub sterowania na projekcie. W innych sami musimy odkryć lub rozwinąć spójny styl.

Podsumowanie

Podobnie jak dobry kucharz, któremu intuicja podpowiada, kiedy można zmienić proporcje składników czy kolejność przygotowań, dobry projektant traktuje metodę projektową tylko jako przewodnik. Kiedy dobrze zaznajomimy się z podstawami, łatwiej będzie nam dostosować proces projektowy do swoich potrzeb — tworzenia tylko tego, co później przydatne, przechodzenia do meritum zagadnienia, rozwiązywania najtrudniejszych problemów. Nabywając doświadczenia, nauczymy się jak widzieć i opisywać problemy, aby łatwiej projektować i budować obiekty, które modelują rozwiązanie.

Rozdział 10., „O projektowaniu", pokazuje, jak podzielić napotykane podczas projektowania problemy na trzy kategorie — problemy dotyczące jądra, problemy odkrywcze i całą resztę — oraz jak odpowiednio podchodzić do każdej kategorii. Jeśli znamy naturę problemu projektowego, nad którym pracujemy, możemy się do niego odpowiednio przygotować.

Projektowanie sterowane odpowiedzialnością znajduje zastosowanie w wielu różnych projektach, ponieważ kładzie nacisk na myślenie i kreatywność. Na początku, na podstawie wymagań dostarczonych przez zleceniodawców, określamy, jak powinna zachowywać się nasza aplikacja. Następnie badamy to, co już wiemy, aby dowiedzieć się, czego jeszcze nie wiemy. Pamiętając, że projekty kształtują się w czasie, do zapisu początkowych pomysłów używamy mało skomplikowanych narzędzi, jak karty CRC, co pozwala nam łatwiej zmieniać zdanie i rozważać inne możliwości.

W końcu skupiamy się na pomijanych wcześniej obszarach. Precyzujemy szczegóły, które dotąd ignorowaliśmy. Szukamy rozwiązań, które sprawdziły się już gdzie indziej. Nasz sukces zależy wprost proporcjonalnie od tego, ile okazji wykorzystaliśmy, ile daliśmy sobie czasu na odkrywanie, refleksję i poprawianie, a także, oczywiście, od zadowolenia naszych zleceniodawców.

Zalecane lektury

Projektowanie sterowane odpowiedzialnością po raz pierwszy zaprezentowane zostało na konferencji OOPSLA '89, w odczycie pod tytułem *Projektowanie obiektowe — podejście sterowane odpowiedzialnością*, autorstwa Rebecki Wirfs-Brock i Briana Wilkersona. W rok później, w książce *Designing Object-Oriented Software* (Prentice Hall, 1990) autorstwa Rebecki Wirfs-Brock, Briana Wilkersona i Lauren Wiener, rozwinięte zostały pomysły przedstawione podczas konferencji. Od tej pory pojęcie odpowiedzialności obiektu bardzo się upowszechniło.

Myślenie zorientowane na odpowiedzialność dopasowuje się do większości procesów oraz praktyk projektowych i uzupełnia je. Na przykład firma Rational zdefiniowała proces nazywany Zunifikowanym Procesem Rationala (ang. *Rational Unified Process*, *RUP*). Określa on cztery fazy iteracyjnego, inkrementalnego procesu tworzenia: inicjacja, opracowywanie, konstruowanie i wdrażanie. Zasady projektowania sterowanego odpowiedzialnością można zastosować podczas faz inicjacji i opracowywania (w innych metodologiach zwanych też projektowaniem obiektowym), nie należy o nich także zapominać podczas konstrukcji. Dobrą książką na temat RUP jest *The Rational Unified Process: An Introduction* (Addison-Wesley, 2000), autorstwa Philippe Kruchtena.

Aktywne (ang. *agile*), adaptujące się do zmian procesy rozwoju oprogramowania są ostatnio popularnym tematem. Również w nich możemy stosować techniki projektowania sterowanego odpowiedzialnością. Aby dowiedzieć się, jakie cechy czynią proces rozwoju aktywnym, najlepiej zajrzeć do książki Jima Highsmitha *Agile Software Development Ecosystems* (Addison-Wesley, 2002). Istnieje przynajmniej kilka różnych procesów rozwojowych, które ich autorzy i zwolennicy uważają za aktywne. Najwięcej pisze się o „programowaniu ekstremalnym" (ang. *eXtreme Programming, XP*), które składa się tylko z dwunastu praktyk rozwojowych. Książka *Extreme Programming Applied: Playing to Win* (Addison-Wesley, 2001), autorstwa Kena Auera i Roya Millera sumaryzuje praktykę programowania ekstremalnego oraz zawiera wiele programistycznych perełek.

Larry Constantine i Lucy Lockwood wprowadzają pojęcie *zasadniczych* przypadków użycia. Mają one formę narracyjną, a pisane są językiem dziedziny aplikacji i jej użytkowników. Czynność wykonywaną przez użytkownika opisują w uproszczony sposób, bez technicznego słownictwa i niezależnie od implementacji. Ponieważ celowo pomijane są szczegóły, opis taki daje więcej możliwości zaprojektowania interfejsu użytkownika.

Zainteresowanym sztuką i praktyką pisania dobrych przypadków użycia możemy polecić kilka książek. Pojęcie to wprowadził Ivar Jacobson w swojej klasycznej książce, *Object-Oriented Software Engineering: A Use Case Driven Approach* (Addison-Wesley, 1994). Wielu autorów włożyło swój unikalny wkład w rozwój przypadków użycia i poczyniło wiele ulepszeń w stosunku do oryginalnego pomysłu Jacobsona. Najbardziej odpowiadają nam książki *Writing Effective Use Cases* (Addison-Wesley, 2002) oraz *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design* (Addison-Wesley, 1999) Larry'ego Constantine'a i Lucy Lockwood. Książka Alistaira Cockburna jest łatwa w czytaniu oraz wypełniona przykładami i poradami, jak radzić sobie z typowymi problemami. Druga z wymienionych książek, autorstwa Larry'ego Constantine'a i Lucy Lockwood, nie dotyczy w ścisłym sensie

przypadków użycia, chociaż do pewnego stopnia opisuje różne style przypadków użycia oraz ich siły i słabości. Ich książka prezentuje systematyczne i dogłębne podejście do tworzenia użytecznych systemów i interfejsów użytkownika, przez rozwijanie modeli ról, zadań i treści. Każdy, kto interesuje się lub zajmuje użytecznością projektowanych systemów, znajdzie w niej dużo cennej wiedzy, ciekawych historii oraz praktycznych narzędzi i technik.