Andrew Hunt, David Thomas

Pragmatyczny programista

OD CZELADNIKA DO MISTRZA

Od ambitnego do najlepszego — czyli jak stać się programistą wydajnym, dociekliwym i gotowym na wszelkie zawodowe wyzwania!

- Poznaj najlepsze praktyki i najczęstsze pułapki procesu wytwarzania oprogramowania
- Naucz się pisać elastyczny, dynamiczny i łatwy w dostosowywaniu kod
- Opanuj sprawdzone techniki efektywnego testowania oprogramowania

Pragmatyczny programista. Od czeladnika do mistrza

Autorzy: Andrew Hunt, David Thomas
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-3237-4
Tytuł oryginału: The Pragmatic Programmer: From Journeyman to Master
Format: 158×235, stron: 348

Od ambitnego do najlepszego – czyli jak stać się programistą wydajnym, dociekliwym i gotowym do wszelkich zawodowych wyzwań!

- Poznaj najlepsze praktyki i najczęstsze pułapki procesu wytwarzania oprogramowania
- Naucz się pisać elastyczny, dynamiczny i łatwy w dostosowywaniu kod
- Opanuj sprawdzone techniki efektywnego testowania oprogramowania

Twórcy rozmaitych narzędzi programistycznych nieustannie próbują nas przekonać o niewiarygodnych możliwościach swoich produktów, a specjaliści od metodyk obiecują, że to właśnie ich techniki zagwarantują nam największą wydajność. Każdy oczywiście twierdzi, że jego język programowania jest najlepszy. . . A jak wszyscy doskonale wiemy, w naszej pracy nie istnieją NAJLEPSZE rozwiązania – są tylko rozwiązania NAJLEPIEJ sprawdzające się w danym projekcie. Większy wpływ na efektywność naszej pracy ma więc doświadczenie oraz znajomość różnych, sprawdzonych praktyk wytwarzania oprogramowania. Zawodowcy, którym na sercu leży przede wszystkim jakość realizowanych projektów, są zwykle zgodni – nigdy nie wiążą swojej zawodowej kariery z jedną, konkretną technologią. To jedna z cech wyróżniających pragmatycznych programistów – produktywnych speców, którzy w pełni wykorzystują swój potencjał i szybko osiągają zawodowy sukces. A oto pierwsza książka, która w pełni odsłania system ich codziennej pracy!

Nie ma znaczenia, czy jesteś wolnym strzelcem, członkiem wielkiego zespołu projektowego, czy konsultantem równocześnie współpracującym z wieloma klientami. Ta skoncentrowana na przekazywaniu praktycznej wiedzy publikacja pokaże Ci, jak efektywnie wykorzystywać swoje umiejętności i doświadczenie do sprawnej realizacji nawet najbardziej złożonych projektów. Podręcznik ilustruje najlepsze praktyki i najczęstsze pułapki wielu różnych aspektów wytwarzania oprogramowania. Znajdziesz w nim zarówno zagadnienia związane ze strategicznym planowaniem swojego zawodowego rozwoju, jak i techniki takiego projektowania architektury, aby przyszły kod był elastyczny, łatwy w dostosowywaniu do różnych okoliczności i przygotowany do wielokrotnego użytku.

Z książki dowiesz się między innymi, jak:

- unikać pułapki powielania wiedzy
- pisać elastyczny, dynamiczny i łatwy w dostosowywaniu kod
- unikać programowania przez koincydencję
- zabezpieczać kod za pomocą kontraktów, asercji i wyjątków
- gromadzić rzeczywiste wymagania
- bezlitośnie i efektywnie testować oprogramowanie
- zachwycać swoich użytkowników
- budować zespoły pragmatycznych programistów
- automatyzować pracę w celu zapewnienia większej precyzji

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

Spis treści

Rozdział 7.

Przed projektem

Czy kiedykolwiek czułeś, że projekt jest nie do zrealizowania, zanim jeszcze rozpoczęła się jego realizacja? Taka sytuacja może mieć miejsce, jeśli prac nad projektem nie poprzedzimy ustaleniem pewnych reguł. W przeciwnym razie równie dobrze można od razu odmówić realizacji projektu i oszczędzić pieniądze jego sponsora.

Na samym początku projektu musimy określić wymagania. Samo wsłuchiwanie się w głos użytkowników nie wystarczy — więcej informacji na ten temat można znaleźć w podrozdziale „Kopalnia wymagań".

Konwencjonalną wiedzą i sposobami zarządzania ograniczeniami zajmiemy się w podrozdziale „Rozwiązywanie niemożliwych do rozwiązania łamigłówek". W zależności od tego, czy pracujemy nad wymaganiami, analizą, kodowaniem, czy testami, możemy spodziewać się różnych problemów. W większości przypadków wspomniane problemy nie są takie trudne, na jakie początkowo wyglądają.

Nawet po rozwiązaniu tych problemów wciąż możemy nie mieć pewności, czy projekt rzeczywiście ma szanse powodzenia. Czy jest to tylko jakiś niepokojący nawyk, odruch, czy coś więcej? W podrozdziale „Nie, dopóki nie jesteś gotowy" można znaleźć sytuacje, kiedy należy zachować zdrowy rozsądek i poważnie traktować te ostrzegające głosy w naszych głowach.

Zbyt szybkie rozpoczęcie projektu to jeden problem, ale zbyt długie oczekiwanie bywa jeszcze bardziej niebezpieczne. W podrozdziale „Pułapka specyfikacji" omówimy zalety specyfikacji na konkretnym przykładzie.

I wreszcie, w podrozdziale „Okręgi i strzałki" przeanalizujemy kilka typowych pułapek czyhających na programistów w ramach formalnych procesów i metodyk wytwarzania. Żadna metoda nie zastąpi myślenia, choćby byłą najlepiej zaplanowana i obejmowała wszystkie znane „najlepsze praktyki".

Jeśli uda się wyeliminować te krytyczne problemy przed przystąpieniem do właściwego projektu, będziemy mogli dużo skuteczniej unikać paraliżu analitycznego i sprawnie rozpocząć prace nad projektem.

36 Kopalnia wymagań

Doskonałość osiąga się nie wtedy, kiedy nie można już nic dodać, lecz gdy już niczego nie można ująć…

Antoine de St. Exupéry, *Ziemia, planeta ludzi*, 1939

W wielu książkach i podręcznikach **zbieranie wymagań** jest prezentowane jako wczesna faza projektu. Samo słowo „zbieranie" sugeruje istnienie jakiejś grupy beztroskich analityków, którzy żywią się leżącymi wokół orzeszkami wiedzy przy pobrzmiewających cicho dźwiękach symfonii Pastoralnej. „Zbieranie" wskazuje na to, że wymagania już istnieją, a nasza rola sprowadza się tylko do ich odnalezienia, umieszczenia w koszyku i radosnego podążania naprzód.

Rzeczywistość jest nieco inna. Wymagania rzadko są dostępne od ręki. Zwykle są raczej dobrze ukryte pod warstwami założeń, nieporozumień i decyzji politycznych.

WSKAZÓWKA NR 51

Nie należy zbierać wymagań — należy je wydobywać z ukrycia.

Poszukiwanie wymagań

Jak rozpoznać prawdziwe wymagania podczas przekopywania się przez otaczający je muł? Odpowiedź jest jednocześnie prosta i skomplikowana.

Prosta odpowiedź mówi, że wymaganie to stwierdzenie dotyczące celu, który musi być osiągnięty. Dobre wymagania mogą obejmować następujące elementy:

- Rekord pracownika może być przeglądany tylko przez wyznaczoną grupę osób.
- Temperatura głowicy cylindra nie może przekraczać wartości krytycznej, której poziom zależy od rodzaju silnika.
- Edytor będzie wyróżniał słowa kluczowe zależnie od typu edytowanego pliku.

Okazuje się jednak, że bardzo niewiele wymagań jest formułowanych w tak jasnych słowach, co znacznie komplikuje proces analizy wymagań.

Pierwsze zdanie z powyższej listy równie dobrze użytkownicy mogliby wyrazić słowami: „Tylko przełożeni pracowników i pracownicy działu personalnego mogą przeglądać rekordy pracowników". Czy takie stwierdzenie rzeczywiście jest wymaganiem? Być może dzisiaj takie odczytanie tego zdania jest możliwe, ale zawarto w nim zdecydowanie zbyt dużo elementów zależnych od polityki. Decyzje polityczne stale podlegają zmianom i jako takie nie powinny być trwale wpisywane w nasze wymagania. Zachęcamy do dokumentowania tych decyzji politycznych poza wymaganiami i do ewentualnego wiązania obu dokumentów za pomocą hiperłączy. Wymaganie powinno być ogólnym stwierdzeniem i przekazywać programistom informacje polityczne w formie przykładu scenariusza, który musi być realizowany przez przyszłą implementację. Decyzje polityczne można też wyrażać w formie metadanych sterujących zachowaniem gotowej aplikacji.

To z pozoru nieistotne rozróżnienie ma zasadniczy wpływ na sytuację programistów zaangażowanych w projekt. Jeśli wymaganie jest wyrażone w ten sposób: „Tylko personel może przeglądać rekord pracownika", programista może być zmuszony do każdorazowego kodowania stosownego testu, kiedy aplikacja uzyskuje dostęp do odpowiednich plików. Jeśli jednak to samo wyrażenie zostanie wyrażone słowami: „Tylko uprawnieni użytkownicy mają dostęp do rekordu pracownika", programista najprawdopodobniej zaprojektuje i zaimplementuje jakiś system kontroli dostępu. W razie zmiany polityki (co na pewno kiedyś nastąpi) aktualizacji będą wymagały tylko metadane używane przez ten system. W praktyce gromadzenie wymagań w ten sposób naturalnie prowadzi do zaprojektowania systemu działającego w oparciu o metadane.

Podział na wymagania, politykę i implementację bywa bardzo nieczytelny w przypadku rozważań poświęconych interfejsom użytkownika. „System musi umożliwiać użytkownikowi wybór pożyczki terminowej" — tak może brzmieć jasne wymaganie. „Potrzebujemy listy wyboru z możliwością wskazania pożyczki terminowej" — to może, ale nie musi być wymaganie. Jeśli użytkownicy koniecznie muszą dysponować listą wyboru, mamy do czynienia z wymaganiem. Jeśli jednak chodzi tylko o możliwość wyboru, a kontrolka listy wyboru jest tylko przykładem, sytuacja nie jest już taka oczywista. W ramce w dalszej części tego podrozdziału zostanie omówiony projekt, który zakończył się fiaskiem tylko dlatego, że ignorowano wymagania dotyczące interfejsu użytkownika.

Bardzo ważne jest odkrycie **powodów**, dla których użytkownicy wykonują określone czynności, nie tylko **sposobu** ich wykonywania. Właściwym celem tworzenia oprogramowania jest przecież rozwiązywanie problemów biznesowych, nie bezrefleksyjna realizacja wymagań. Dokumentowanie powodów uzasadniających poszczególne wymagania jest dla zespołu projektowego bezcennym źródłem informacji podczas podejmowania codziennych decyzji dotyczących samej implementacji.

Istnieje pewna prosta technika bliższego poznawania wymagań użytkowników, która o dziwo nie cieszy się zbyt dużą popularnością — wystarczy na chwilę zostać użytkownikiem. Piszemy system dla pracowników działu wsparcia technicznego? Warto poświęcić kilka dni na odbieranie telefonów od klientów

(w towarzystwie doświadczonego pracownika odpowiedniego działu). Otrzymaliśmy zadanie automatyzacji systemu kontroli zasobów magazynowych? Może warto spędzić tydzień w magazynie[1]. Oprócz możliwości zyskania głębszej wiedzy o przyszłych sposobach **używania** naszego systemu ze zdziwieniem odkryjemy, jak pytanie: „Czy mogę usiąść obok i przez tydzień przypatrywać się twojej pracy?" może pomóc w budowaniu zaufania i nawiązywaniu komunikacji z przyszłymi użytkownikami. Musimy tylko pamiętać, aby nigdy nie przeszkadzać przyszłym użytkownikom!

WSKAZÓWKA NR 52

Aby myśleć jak użytkownik, należy z nim popracować.

Proces gromadzenia i poznawania wymagań to także czas na nawiązywanie kontaktów z przyszłymi użytkownikami, próby zrozumienia ich oczekiwań i nadziei wiązanych z budowanym przez nas systemem. Więcej informacji na ten temat można znaleźć w podrozdziale „Wielkie oczekiwania" w rozdziale 8.

Dokumentowanie wymagań

Załóżmy, że usiedliśmy już przy jednym stole z użytkownikami i próbujemy wyciągnąć od nich jakieś ogólne wymagania. Wspólnie wypracowujemy kilka prawdopodobnych scenariuszy, które dość dobrze opisują funkcje przyszłej aplikacji. Jako profesjonaliści chcemy zapisać te wnioski i opublikować gotowy dokument, tak aby każdy (programiści, użytkownicy końcowi i sponsorzy projektu) mógł go używać jako podstawy do dalszych dyskusji.

Grupa odbiorców jest dość szeroka.

Ivar Jacobson [Jac94] zaproponował model, w którym do gromadzenia wymagań wykorzystuje się **przypadki użycia**. W ten sposób można opisywać konkretne scenariusze pracy z systemem — nie przez pryzmat interfejsu użytkownika, tylko w sposób bardziej abstrakcyjny. W pracy Jacobsona zabrakło niestety szczegółów, stąd tak wiele współczesnych, zupełnie odmiennych koncepcji dotyczących kształtu samych przypadków użycia. Czy przypadki użycia powinny mieć formalny, czy nieformalny charakter; czy powinny mieć postać opisowego tekstu, czy raczej dokumentu z precyzyjnie określoną strukturą (na przykład zbliżoną do formularza)? Jaki poziom szczegółowości będzie właściwy (zważywszy na szeroką grupę odbiorców)?

[1] Czy tydzień to zbyt długo? Z pewnością nie, szczególnie jeśli w tym czasie mamy obserwować procesy, w których kierownictwo i szeregowi pracownicy działają w zupełnie innych światach. W takim przypadku kierownictwo przedstawi nam jedną wizję funkcjonowania przedsiębiorstwa, ale kiedy zejdziemy piętro niżej, odkryjemy zupełnie inną rzeczywistość, której zrozumienie z natury rzeczy wymaga czasu.

Czasem to interfejs jest właściwym systemem

W artykule opublikowanym w magazynie „Wired" (styczeń 1999, strona 176.) producent i muzyk Brian Eno opisał niewiarygodny przykład technologii — nowoczesną konsoletę mikserską. Taka konsoleta pozwala na dosłownie wszystko, co tylko można zrobić z dźwiękiem. Mimo to, zamiast umożliwiać muzykom tworzenie lepszych utworów oraz szybciej i taniej nagrywać kolejne dzieła, konsoleta staje na drodze procesu twórczego i mocno go zakłóca.

Aby lepiej zrozumieć, dlaczego tak się dzieje, warto przyjrzeć się pracy inżynierów dźwięku. Okazuje się, że ustawiają parametry dźwięku intuicyjnie. Przez lata wypracowują swoistą pętlę zmysłów łączącą ich uszy i palce — na tej podstawie ustawiają suwaki wyciszania, obracają gałki itp. Co ciekawe, interfejs nowego miksera w żaden sposób nie wykorzystuje tych zdolności i przyzwyczajeń. Przeciwnie — zmusza użytkowników do wydawania poleceń za pomocą klawiatury i myszy. Zestaw funkcji oferowany przez nowy mikser był wystarczająco bogaty, jednak sposób ich opakowania odbiegał od standardów i był wręcz egzotyczny. Funkcje potrzebne inżynierom dźwięku zostały ukryte za niezrozumiałymi nazwami lub są dostępne dopiero po użyciu nieintuicyjnych kombinacji podstawowych elementów.

Podstawowym wymaganiem stawianym przed nowym środowiskiem jest efektywne wykorzystanie istniejących umiejętności użytkownika. Bezmyślne powielanie tego, co już istnieje, nie gwarantuje co prawda postępu, jednak musimy znaleźć sposób płynnego **przejścia** do przyszłości.

Na przykład inżynierowie dźwięku najprawdopodobniej woleliby otrzymać interfejs z ekranem dotykowym, aby nadal mieć pod ręką namacalne kontrolki przynajmniej zbliżone do tych oferowanych przez tradycyjne konsolety mikserskie, a jednocześnie zyskać szersze możliwości zapewniane przez oprogramowanie (większe niż w przypadku fizycznych gałek i przełączników). Warunkiem sukcesu jest zapewnienie płynnego, komfortowego przejścia pomiędzy znanymi rozwiązaniami a nowymi mechanizmami.

Przytoczony przykład ilustruje też naszą tezę, zgodnie z którą najlepsze narzędzia muszą dostosowywać się do rąk rzemieślnika, który ich używa. W tym przypadku to narzędzia budowane przez nas dla innych muszą dostosowywać się do ich potrzeb i możliwości.

Jednym ze sposobów odczytywania przypadków użycia jest zwracanie szczególnej uwagi na opisywane cele. Alistair Cockburn opracował artykuł opisujący ten model i zaproponował szablony, które można wykorzystać (w oryginalnej lub zmienionej formie) jako punkt wyjścia ([Coc97a] oraz w internecie pod adresem [URL 46]). Na rysunku 7.1 pokazano uproszczony przykład takiego szablonu. Na rysunku 7.2 przedstawiono przykładowy przypadek użycia.

Stosowanie formalnego szablonu w roli **przypomnienia** daje nam pewność, że nasze przypadki użycia zawsze będą obejmowały wszystkie niezbędne informacje: parametry wydajności, informacje o innych zaangażowanych stronach, priorytet, częstotliwość oraz rozmaite błędy i oczekiwane problemy, które mogą pojawić się w przyszłości (tzw. wymagania niefunkcjonalne). Przypadki użycia

A. CHARAKTERYSTYCZNE INFORMACJE
- Cel w szerszym kontekście
- Zasięg
- Poziom
- Warunki wstępne
- Warunek pomyślnego końca
- Warunek niepowodzenia
- Główny aktor
- Przyczyna

B. PODSTAWOWY POMYŚLNY SCENARIUSZ

C. ROZSZERZENIA

D. ODMIANY

E. INFORMACJE POKREWNE
- Priorytet
- Docelowa wydajność
- Częstotliwość
- Nadrzędny przypadek użycia
- Podrzędne przypadki użycia
- Kanał komunikacji z głównym aktorem
- Aktorzy drugorzędni
- Kanał komunikacji z aktorami drugorzędnymi

F. HARMONOGRAM

G. PROBLEMY DO ROZWIĄZANIA

Rysunek 7.1. Szablon przypadku użycia autorstwa Cockburna

w tej formie stanowią też doskonałe miejsce dla komentarzy użytkowników, jak: „Nie, w razie wystąpienia warunku *X* musimy raczej zrobić *Y*". Szablon pełni funkcję gotowej agendy spotkań z przyszłymi użytkownikami naszych produktów.

Proponowana organizacja umożliwia łatwe porządkowanie przypadków użycia w ramach struktur hierarchicznych, gdzie bardziej szczegółowe przypadki użycia są zagnieżdżane w ramach przypadków wyższego poziomu. Na przykład płatności **kartą debetową** i **kartą kredytową** to wyspecjalizowane formy transakcji **kartą płatniczą**.

Diagramy przypadków użycia

Przepływ pracy można wyrażać na diagramach czynności UML, zaś diagramy klas na poziomie pojęciowym mogą być z powodzeniem wykorzystywane do modelowania interesujących nas procesów biznesowych. Prawdziwe przypadki użycia mają jednak postać tekstowych opisów z określoną hierarchią i wzajemnymi odwołaniami. Przypadki użycia mogą zawierać hiperłącza do innych przypadków użycia i mogą być zagnieżdżane w ramach pozostałych przypadków.

Wielu programistów nie może uwierzyć, że ktokolwiek może poważnie traktować pomysł dokumentowania tak szczegółowych informacji, rysując ludziki złożone z kilku linii (patrz rysunek 7.3). Nie powinniśmy jednak przywiązywać się do jakiejkolwiek notacji; powinniśmy raczej wybierać tę metodę, która pozwala najskuteczniej komunikować wymagania naszym odbiorcom.

PRZYPADEK UŻYCIA NR 5: ZAKUP TOWARÓW

A. CHARAKTERYSTYCZNE INFORMACJE
- Cel w szerszym kontekście: Kupujący przekazuje żądanie bezpośrednio do naszej firmy i oczekuje, że dostarczymy mu towar i wystawimy rachunek.
- Zasięg: firma.
- Poziom: podsumowanie.
- Warunki wstępne: znamy kupującego, jego adres itp.
- Warunek pomyślnego końca: kupujący dysponuje towarem, a my otrzymujemy zapłatę za ten towar.
- Warunek niepowodzenia: nie wysłaliśmy towaru, a klient nie wysłał nam pieniędzy.
- Główny aktor: kupujący bądź dowolny agent (lub komputer) działający na rzecz klienta.
- Przyczyna: do systemu przychodzi żądanie zakupu.

B. PODSTAWOWY POMYŚLNY SCENARIUSZ
1. Kupujący przekazuje do systemu żądanie zakupu.
2. Firma otrzymuje nazwisko kupującego, jego adres, zamawiane towary itp.
3. Firma przekazuje informacje na temat towarów, cen, dat dostawy itp.
4. Kupujący podpisuje zamówienie.
5. Firma tworzy zamówienie i dostarcza towar kupującemu.
6. Firma przekazuje kupującemu fakturę.
7. Kupujący dokonuje płatności na podstawie faktury.

C. ROZSZERZENIA
3a. Firma wyczerpała zapasy jednego z zamówionych towarów: należy renegocjować zamówienie.
4a. Firma od razu płaci kartą kredytową: należy przyjąć płatność kartą kredytową (przypadek użycia nr 44).
7a. Kupujący zwraca towar: należy przyjąć zwrócony towar (przypadek użycia nr 105).

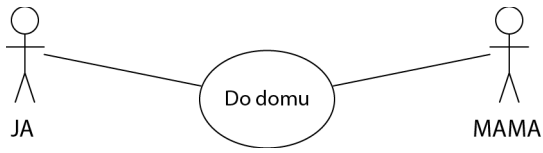
D. ODMIANY
1. Kupujący może złożyć zamówienie telefonicznie, wysyłając faks, wypełniając formularz internetowy lub wysyłając wiadomość poczty elektronicznej.
7. Kupujący może zapłacić gotówką, przelewem, czekiem lub kartą kredytową.

E. INFORMACJE POKREWNE
- Priorytet: najwyższy.
- Docelowa wydajność: 5 minut dla żądania, 45 dni na uregulowanie płatności.
- Częstotliwość: 200/dzień.
- Nadrzędny przypadek użycia: zarządzanie relacjami z klientem (przypadek użycia nr 2).
- Podrzędne przypadki użycia: tworzenie zamówienia (15); przyjęcie płatności kartą kredytową (44); przyjęcie zwróconego towaru (105).
- Kanał komunikacji z głównym aktorem: Możliwe rozmowy telefoniczne, wymiana plików lub formy interaktywne.
- Aktorzy drugorzędni: operator płatności kartami kredytowymi, bank, firma kurierska.

F. HARMONOGRAM
- Data wykonania: wydanie 1.0.

G. PROBLEMY DO ROZWIĄZANIA
- Co stanie się, jeśli otrzymamy tylko część zamówienia?
- Co będzie, jeśli zostanie użyta kradziona karta kredytowa?

Rysunek 7.2. Przykładowy przypadek użycia



Rysunek 7.3. Przypadki użycia w notacji UML — nawet dziecko to potrafi!

Zbyt duża liczba szczegółów

Jednym z największych zagrożeń podczas sporządzania dokumentu z wymaganiami jest dążenie do zapisania zbyt wielu szczegółów. Dobre dokumenty o wymaganiach zachowują swoją abstrakcyjność. W przypadku wymagań najprostsze stwierdzenia, które możliwie precyzyjnie wyrażają potrzeby biznesowe, sprawdzają się zdecydowanie najlepiej. Nie chodzi jednak o przesadną ogólnikowość — w naszych wymaganiach musimy uwzględnić niezmienniki semantyczne, a konkretne lub bieżące praktyki należy udokumentować raczej w formie polityki.

Wymagania to nie architektura. Wymagania to nie projekt ani interfejs użytkownika. Wymagania to konieczność.

Widzieć dalej

Problem roku 2000 często kojarzy się z krótkowzrocznymi programistami, którzy desperacko poszukiwali możliwości oszczędzenia choćby kilku bajtów pamięci w czasach, gdy największe komputery dysponowały mniejszą ilością pamięci niż współczesny pilot do telewizora.

W rzeczywistości nie była to ani wina programistów, ani problem niedostatecznej ilości pamięci. Gdybyśmy mieli kogokolwiek winić za to niedopatrzenie, za błąd odpowiadają raczej analitycy i projektanci systemów. Problem roku 2000 miał dwie przyczyny — nieumiejętność przewidywania sytuacji poza bieżącą praktyką biznesową oraz naruszanie zasady DRY.

Firmy posługiwały się skróconym, dwucyfrowym formatem zapisu lat na długo przed pojawieniem się komputerów. Była to powszechna praktyka. Działanie wczesnych aplikacji przetwarzających dane ograniczało się do automatyzacji istniejących procesów biznesowych, stąd powielenie błędnego zapisu. Nawet gdyby architektura narzucała programistom stosowanie dwucyfrowych reprezentacji lat w danych wejściowych, raportach i bazie danych, powinna istnieć jakaś abstrakcja DATA, która „wiedziałaby", że te dwie cyfry to tylko skrócona forma rzeczywistej daty.

WSKAZÓWKA NR 53

Abstrakcje żyją dłużej niż szczegóły.

Czy „widzieć dalej" wymaga od nas przewidywania przyszłości? Nie — chodzi raczej o zapisywanie wymagań w następujący sposób:

> *System często korzysta z abstrakcji DATA. System będzie implementował usługi związane z tą abstrakcją, jak formatowanie, zapisywanie czy operacje matematyczne, w spójny i uniwersalny sposób.*

Wymagania w tej formie określają tylko to, że system będzie operował na datach. Mogą też sugerować, że na datach będą wykonywane jakieś działania matematyczne. Mogą wskazywać, że daty będą dodatkowo przechowywane w rozmaitych formatach. Mamy tutaj do czynienia z ogólnymi wymaganiami dotyczącymi modułu lub klasy DATA.

Jeszcze tylko jedna malutka funkcja…

Wiele projektów kończy się niepowodzeniem wskutek niekontrolowanego rozszerzania zakresu prac, czyli zjawiska określanego mianem przerostu funkcji. Mamy tutaj do czynienia z pewnym aspektem syndromu gotowanej żaby z podrozdziału „Zupa z kamieni i gotowane żaby" w rozdziale 1. Co możemy zrobić, aby zapobiec wpadnięciu w pułapkę zbyt wielu wymagań?

W literaturze można znaleźć opisy wielu różnych miar, jak liczba zgłoszonych i usuniętych błędów, gęstość usterek, spójność, związki, punkty funkcyjne, wiersze kodu itp. Wartości tych miar można śledzić ręcznie lub za pomocą odpowiedniego oprogramowania.

Okazuje się jednak, że tylko w niewielkiej części projektów aktywnemu śledzeniu podlegają wymagania. Oznacza to, że uczestnicy tych raportów nie mają możliwości raportowania o zmianach zakresu prac — tego, kto żądał poszczególnych funkcji, kto zatwierdził te wnioski, jaka jest łączna liczba zaakceptowanych wymagań itp.

Kluczem do zarządzania wzrostem liczby wymagań jest jasne stwierdzenie, że każda nowa funkcja wydłuża termin przekazania gotowego produktu sponsorom projektu. Kiedy projekt jest opóźniony o rok względem początkowych szacunków i kiedy wszyscy dookoła zaczynają formułować wzajemne oskarżenia, warto dysponować precyzyjnym, kompletnym obrazem tego, jak i kiedy zanotowano wzrost liczby wymagań.

Bardzo łatwo wpaść w wir „tylko jednej dodatkowej funkcji", jednak uważne śledzenie wymagań może nam ułatwić odkrycie, że ta tylko jedna dodatkowa funkcja to tak naprawdę już piętnasty element dodany w tym miesiącu.

Utrzymywanie glosariusza

W momencie przystąpienia do rozmowy o wymaganiach użytkownicy i eksperci z danej dziedziny zaczynają używać pewnych terminów, które mają dla nich specyficzne znaczenie. Mogą na przykład odróżniać klienta od kupującego. W takim przypadku zamienne stosowanie obu słów w systemie byłoby niewłaściwe.

Warto więc utworzyć i utrzymywać **glosariusz** na potrzeby projektu, czyli jedno miejsce, w którym będą definiowane wszystkie terminy i słownictwo używane w ramach projektu. Wszyscy uczestnicy projektu, od użytkowników końcowych po pracowników działu wsparcia, powinni posługiwać się tym glosariuszem, aby

zachowywać spójność terminologii. Oznacza to, że glosariusz powinien być powszechnie dostępny — to jeden z argumentów przemawiających za dokumentacją udostępnianą na stronach WWW (wrócimy do tego tematu w dalszej części tego podrozdziału).

WSKAZÓWKA NR 54

Należy stosować glosariusz projektu.

Bardzo trudno pomyślnie zakończyć projekt, w którym użytkownicy i programiści stosują odmienne nazwy dla tych samych elementów czy zdarzeń lub — co gorsza — odwołują się do różnych aspektów, posługując się tą samą nazwą.

Dokumenty są dla wszystkich

W podrozdziale „Pisanie przede wszystkim" w rozdziale 8. omówimy problem publikowania dokumentów projektu w wewnętrznych serwisach WWW, tak aby zapewnić łatwy dostęp do tych dokumentów wszystkim uczestnikom projektu. Proponowana metoda dystrybucji dokumentacji jest szczególnie przydatna w przypadku dokumentów poświęconych wymaganiom.

Prezentując wymagania w formie dokumentu hipertekstowego, możemy skuteczniej odpowiadać na oczekiwania różnych odbiorców — każdy czytelnik może znaleźć w tego rodzaju dokumentach to, co go interesuje. Sponsorzy projektu mogą otrzymywać informacje na wysokim poziomie abstrakcji, które dadzą im pewność co do spełniania celów biznesowych. Programiści mogą używać hiperłączy do wygodnego przechodzenia do coraz bardziej szczegółowych informacji (nawet na poziomie odwołań do odpowiednich definicji lub specyfikacji inżynierskich).

Model, w którym dokumentacja jest umieszczana na stronach internetowych, eliminuje też problem typowych, opasłych tomów zatytułowanych *Analiza wymagań*, których nikt nigdy nie czyta i które stają się nieaktualne, zanim obeschnie tusz na papierze.

Jeśli coś jest internecie, jest szansa, że nawet programiści to przeczytają.

Pokrewne podrozdziały

- „Zupa z kamieni i gotowane żaby" w rozdziale 1.
- „Odpowiednio dobre oprogramowanie" w rozdziale 1.
- „Okręgi i strzałki" w rozdziale 7.
- „Pisanie przede wszystkim" w rozdziale 8.
- „Wielkie oczekiwania" w rozdziale 8.

Wyzwania

- Czy używasz oprogramowania, które piszesz? Czy można dobrze zgromadzić i zrozumieć wymagania bez samodzielnego sprawdzenia oprogramowania?
- Wybierz jakiś problem niezwiązany z komputerami, który właśnie musisz rozwiązać. Opracuj wymagania dla rozwiązania tego problemu (bez użycia komputera).

Ćwiczenia

42. Które z poniższych zdań zasługują na miano pełnowartościowych wymagań? Spróbuj (jeśli to możliwe) inaczej wyrazić zdania, które nie spełniają warunków dobrych wymagań.

Patrz odpowiedź 42. w dodatku B.

1. Czas odpowiedzi musi być krótszy niż 500 ms.
2. Okna dialogowe będą miały szary kolor tła.
3. Aplikacja zostanie zorganizowana jako pewna liczba procesów frontowych oraz jeden serwer wewnętrzny.
4. Jeśli użytkownik poda znaki nienumeryczne w polu numerycznym, system odtworzy dźwięk ostrzegawczy i odrzuci wprowadzoną wartość.
5. Kod i dane aplikacje nie mogą zajmować więcej niż 256 kB.

37

Rozwiązywanie niemożliwych do rozwiązania łamigłówek

Gordios, król Frygii, zawiązał kiedyś węzeł, którego nikt nie potrafił rozsupłać. Mówiono, że ten, kto rozwiąże zagadkę węzła gordyjskiego, zdobędzie władzę nad Azją. Zagadkę rozwiązał dopiero Aleksander Wielki, który przeciął węzeł mieczem. Okazało się, że wystarczyła tylko inna interpretacja wymagań — to wszystko… i rzeczywiście Aleksander podbił znaczną część Azji.

Od czasu do czasu odkrywamy gdzieś w środku projektu, że nie potrafimy zrobić choćby kroku naprzód. Trafiamy na przeszkodę niemożliwą do rozwiązania, jak nieumiejętność radzenia sobie z jakąś technologią czy fragment kodu, który okazuje się dużo trudniejszy do napisania, niż początkowo zakładaliśmy. Być może problem rzeczywiście wydaje się niemożliwy do rozwiązania. Czy jednak rzeczywiście jest taki trudny, na jaki wygląda?

Przeanalizujmy tradycyjne układanki — wszystkie te kłopotliwe kształty z drewna, stali lub plastiku, które tak często znajdujemy pod choinką lub na wyprzedażach niepotrzebnych rzeczy. Zwykle wystarczy przenieść okrągły kształt w inne miejsce, umieścić klocek w kształcie T w określonym miejscu itp.

Przenosimy więc okrągły kształt lub próbujemy umieścić klocek w kształcie litery T w określonym miejscu, aby szybko odkryć, że to oczywiste rozwiązanie nie zdaje egzaminu. Układanek nie można rozwiązywać w ten sposób. To, że rozwiązanie nie jest oczywiste, nie powstrzymuje ludzi przed próbami wielokrotnego powtarzania tych samych czynności w przekonaniu, że łamigłówka musi mieć jakieś rozwiązanie.

To oczywiste, że w ten sposób nie można dojść do rozwiązania. Rozwiązanie leży gdzie indziej. Sekretem układanki jest identyfikacja rzeczywistych (nie wyobrażonych) ograniczeń i znalezienie rozwiązania w ich ramach. Niektóre ograniczenia mają **bezwzględny** charakter; inne mają raczej postać **nieuzasadnionych uprzedzeń**. Ograniczenia bezwzględne muszą być przestrzegane niezależnie od tego, czy sprawiają wrażenie nielogicznych lub wręcz głupich. Istnieją też pozorne ograniczenia, które nie mają nic wspólnego z rzeczywistością. Istnieje na przykład stara sztuczka znana bywalcom barów, która polega na wzięciu nowej, zamkniętej butelki szampana i przyjmowaniu zakładów, jakoby można z niej wypić piwo. Cała sztuka polega na odwróceniu butelki do góry nogami i wlaniu niewielkiej ilości piwa do wgłębienia na jej spodzie. Wiele problemów dotyczących oprogramowania można rozwiązać w równie przebiegły sposób.

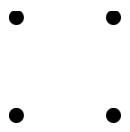
Stopnie swobody

Popularne wyrażenie „wykraczać myślami poza schematy" (ang. *thinking outside the box*) zachęca nas do identyfikacji ograniczeń, które w naszym przypadku nie znajdują zastosowania, i do ich ignorowania.

Przytoczona koncepcja nie jest jednak w pełni słuszna. Jeśli tym „schematem" jest warunek graniczny, problem polega raczej na **znalezieniu** schematu, który co najwyżej może być istotnie szerszy, niż początkowo sądzimy.

Kluczem do rozwiązania układanki jest zarówno rozpoznanie krępujących nas ograniczeń, jak i stopni swobody, którymi dysponujemy — dopiero na tej podstawie możemy znaleźć wyjście z sytuacji. Właśnie dlatego układanki są takie kłopotliwe; często zbyt pochopnie rezygnujemy z potencjalnych rozwiązań.

Czy potrafimy na przykład połączyć wszystkie punkty na poniższym rysunku i wrócić do punktu wyjścia, rysując zaledwie trzy proste odcinki (bez odrywania długopisu od papieru ani dwukrotnego rysowania odcinka łączącego te same punkty) [Hol78]?



Musimy zmierzyć się ze wszystkimi przyjętymi z góry wyobrażeniami i ocenić, czy rzeczywiście reprezentują fizyczne ograniczenia.

Problemem nie jest więc to, czy myślimy schematycznie, czy potrafimy wyjść poza schematy. Kluczem do rozwiązania jest raczej **znalezienie** schematu — identyfikacja faktycznych ograniczeń.

WSKAZÓWKA NR 55

Nie należy wykraczać myślami poza schemat — należy raczej znaleźć ten schemat.

W razie napotkania szczególnie kłopotliwego problemu warto zapisać sobie wszystkie możliwe ścieżki rozwiązania, które na tym etapie potrafimy dostrzec. Nie należy niczego pomijać, choćby wydawało się zupełnie niepraktyczne lub wręcz głupie. Dopiero po sporządzeniu tej listy warto ją uważnie przejrzeć i wyjaśnić, dlaczego ta czy inna ścieżka nie doprowadzi do szczęśliwego końca. Czy na pewno? Potrafimy to udowodnić?

Przypomnijmy sobie historię konia trojańskiego — nowatorskiego rozwiązania problemu, który wydawał się niemożliwy do rozwiązania. Jak niepostrzeżenie przerzucić wojsko do dobrze ufortyfikowanego miasta? Jesteśmy pewni, że koncepcja „przez główną bramę" początkowo była odrzucana jako samobójcza.

Warto przypisywać poszczególne ograniczenia do kategorii i nadawać im priorytety. Kiedy stolarze przystępują do projektu, zaczynają od cięcia najdłuższych fragmentów drewna, by następnie odpowiednio pociąć pozostałe fragmenty. W ten sam sposób chcemy najpierw identyfikować najbardziej krępujące ograniczenia i umieszczać pozostałe ograniczenia w ich ramach.

Rozwiązanie zagadki czterech punktów łączonych trzema odcinkami można znaleźć w dodatku B.

Musi istnieć prostszy sposób!

Zdarza się, że pracujemy nad rozwiązaniem problemu, który sprawia wrażenie dużo trudniejszego, niż jest w rzeczywistości. Często sądzimy, że obraliśmy niewłaściwą drogę — musi przecież istnieć prostszy sposób osiągnięcia celu! Być może już teraz nie jesteśmy w stanie dotrzymać harmonogramu lub wręcz popadamy w rozpacz, tracąc wiarę w możliwość prawidłowego funkcjonowania systemu, ponieważ jakiś problem wydaje się „niemożliwy do rozwiązania".

W takich przypadkach powinniśmy zatrzymać się na chwilę i zadać sobie kilka pytań:

- Czy **istnieje** prostszy sposób?
- Czy rozwiązujemy właściwy problem, czy natrafiliśmy tylko na zewnętrzną przeszkodę techniczną?
- **Dlaczego** w ogóle analizowana kwestia jest problemem?
- Co sprawia, że jego rozwiązanie jest trudne?

- Czy nie ma innego rozwiązania?
- Czy w ogóle musimy to robić?

Próby odpowiedzenia sobie na te pytania nierzadko prowadzą do zaskakujących odkryć. W wielu przypadkach wystarczy ponowna interpretacja wymagań, aby pozbyć się całego zbioru problemów (tak jak w przypadku węzła gordyjskiego).

Wszystko, czego nam trzeba, to prawdziwe ograniczenia, nietrafione ograniczenia oraz wiedza, jak je rozróżniać.

Wyzwania

- Spróbuj z dystansu spojrzeć na dowolny trudny problem, który właśnie próbujesz rozwiązać. Czy możesz po prostu przeciąć ten węzeł gordyjski? Zadaj sobie wymienione powyżej pytania, w szczególności: **„Czy nie ma innego rozwiązania?”**.
- Czy zbiór ograniczeń był znany w momencie podpisywania kontraktu na bieżący projekt? Czy zdefiniowane wówczas ograniczenia wciąż są aktualne i czy ich interpretacja zachowała swoją wartość?

38 Nie, dopóki nie jesteś gotowy

Czasem chwila zawahania może być wybawieniem.

James Thurber, *The Glass in the Field*

Najlepsi wykonawcy mają jedną wspólną cechę: wiedzą, kiedy zacząć i kiedy skończyć. Nurek stoi na trampolinie i czeka na idealny moment do skoku. Dyrygent nieruchomo stoi przed orkiestrą z uniesionymi rękami aż do momentu, w którym uzna, że to najlepszy czas na rozpoczęcie koncertu.

My także chcemy być wielkimi wykonawcami. Musimy wsłuchiwać się w głos podpowiadający: „Zaczekaj”. Jeśli siadamy do pisania kodu i stale nachodzą nas jakieś wątpliwości, nie możemy ich lekceważyć.

WSKAZÓWKA NR 56

Należy słuchać uporczywych wątpliwości — nie wolno zaczynać pracy, dopóki nie jest się gotowym.

Istniał kiedyś model trenowania tenisa określany mianem „gry wewnętrznej”. Trening polegał na wielogodzinnym przebijaniu piłek nad siatką bez zwracania szczególnej uwagi na precyzję — chodziło raczej o ocenę miejsca upadania piłki względem jakiegoś celu (zwykle krzesła).Celem tych ćwiczeń było trenowanie podświadomości i refleksu, tak aby zawodnik potrafił bez zastanowienia wybierać właściwy sposób uderzenia piłki.

Jako programiści robimy mniej więcej to samo przez całą karierę. Próbujemy rozmaitych rozwiązań i sprawdzamy, które z nich zdają egzamin, a które okazały się nietrafione. Z czasem gromadzimy rozmaite doświadczenia i wiedzę. Kiedy zmagamy się z uporczywymi wątpliwościami lub nasze doświadczenie podpowiada nam, aby obrać inną drogę, warto z tych „podszeptów" skorzystać. Być może nie jesteśmy w stanie dokładnie wskazać palcem, co nam się nie podoba, ale wystarczy trochę czasu, aby obecne wątpliwości przerodziły się w coś bardziej namacalnego — konkretny problem do rozwiązania. Wytwarzanie oprogramowania wciąż nie jest nauką. Możemy więc pozwolić sobie na udział instynktu w realizowanych przedsięwzięciach.

Uzasadniona obawa czy niepotrzebna zwłoka?

Każdy boi się pustej kartki papieru. Rozpoczynanie nowego projektu (lub nawet rozpoczynanie prac nad nowym modułem w ramach istniejącego projektu) bywa bardzo irytującym doświadczeniem. Wielu programistów wolałoby jak najdłużej odkładać te szczególnie trudne, początkowe fazy projektu. Jak w takim razie stwierdzić, czy mamy do czynienia z nieuzasadnioną grą na zwłokę, czy odpowiedzialnym oczekiwaniem na zgromadzenie wszystkich niezbędnych elementów?

W naszym przypadku najskuteczniejszą techniką radzenia sobie w tych okolicznościach jest tworzenie prototypów. Należy wybrać obszar, który wydaje nam się szczególnie kłopotliwy, i przystąpić do tworzenia rozwiązań potwierdzających lub obalających te założenia. W większości przypadków tworzenie prototypów prowadzi do jednej z dwóch sytuacji. Z jednej strony, krótko po przystąpieniu do tych eksperymentów możemy uznać, że tracimy czas. To zniechęcenie często pokazuje, że początkowe obawy były spowodowane tylko niechęcią do pierwszych faz projektu. Warto wówczas przerwać prace nad prototypami i przejść do właściwego wytwarzania.

Z drugiej strony, podczas tworzenia prototypów możemy nagle odkryć, że któreś z podstawowych założeń dotyczących danego projektu było błędne. Co więcej, będziemy potrafili jasno określić, jak zmienić i wyrazić na nowo to założenie. W takim przypadku możemy w poczuciu komfortu przerwać prace nad prototypami i przystąpić do realizacji właściwego projektu (z uwzględnieniem skorygowanej wiedzy). Instynkt nas nie zawiódł — właśnie oszczędziliśmy sobie i naszemu zespołowi mnóstwo wysiłku, który w przeciwnym razie poszedłby na marne.

Jeśli decydujemy się na przygotowanie prototypu jako sposobu lepszego zbadania źródeł swojego niepokoju, koniecznie musimy pamiętać o pierwotnych przyczynach tej decyzji. Ostatnią rzeczą, której nam potrzeba, jest poświęcenie wielu tygodni na poważne prace programistyczne tylko po to, aby wreszcie przypomnieć sobie, że pracujemy tylko nad prototypem.

Proponowany model jest też przejawem cynizmu — łatwiej zyskać polityczną akceptację dla eksperymentu z prototypem niż prostego stwierdzenia: „Mam obawy co do tego projektu" i ostentacyjnego rozpoczęcia układania pasjansa.

Wyzwania

- Omów syndrom obaw przed początkiem projektu ze swoimi współpracownikami. Czy inni doświadczają tego samego? Czy głośno wyrażają swoje obawy? Jakich sztuczek używają do radzenia sobie z tym problemem? Czy cała grupa jest zaangażowana w rozwiewanie obaw poszczególnych członków zespołu, czy raczej wywiera dodatkową presję?

39

Pułapka specyfikacji

Pilot lądujący zachowuje status pilota prowadzącego do momentu zejścia na wysokość decyzyjną, kiedy prowadzący pilot nielądujący przejmuje zadania nieprowadzącego pilota lądującego, chyba że ten drugi wyda komendę „odejście". W takim przypadku nielądujący pilot prowadzący dalej prowadzi, a lądujący pilot nieprowadzący dalej nie prowadzi aż do następnej komendy „ląduj" lub „odejście". W związku z ostatnimi nieporozumieniami dotyczącymi tych przepisów, uznałem, że należy je doprecyzować.

Memorandum linii British Airways
cytowane w piśmie „Pilot Magazine", grudzień 1996

Specyfikacja programu to proces przetwarzania wymagań i ich redukowania do punktu, w którym programista może efektywnie korzystać ze swoich umiejętności. Tworzenie specyfikacji to akt komunikacji, wyjaśniania i precyzowania faktów w sposób pozwalający wyeliminować najważniejsze niejasności. Oprócz przesłania dla programisty, który będzie pracował nad początkową implementacją, specyfikacja jest też zapisem dla przyszłych pokoleń programistów, którzy będą konserwowali i rozszerzali ten kod. Specyfikacja jest też swoistą umową z użytkownikiem — zapisem jego potrzeb i nieformalnym kontraktem potwierdzającym, że system w swojej ostatecznej formie będzie spełniał konkretne wymaganie.

Pisanie specyfikacji to duża odpowiedzialność.

Problem w tym, że wielu projektantów nie wie, kiedy przestać. Pracują w poczuciu, że zadanie jest wykonane dopiero po zapisaniu każdego, choćby najdrobniejszego szczegółu.

Taka metoda jest błędna z kilku powodów. Po pierwsze, wiara w to, że jakakolwiek specyfikacja może uwzględniać wszystkie szczegóły i niuanse systemu bądź jego wymagań, jest przejawem naiwności. W ograniczonych dziedzinach problemów istnieją zwykle formalne metody opisywania systemów, co jednak nie zwalnia projektanta z obowiązku wyjaśnienia znaczenia stosowanej notacji użytkownikom końcowym — żadna notacja nie eliminuje problemu interpretacji przez człowieka. Nawet bez problemów związanych z tą interpretacją trudno oczekiwać, aby przeciętny użytkownik potrafił precyzyjnie określić, czego oczekuje od nowego systemu. Nawet jeśli użytkownicy twierdzą, że rozumieją wymagania,

i podpisują przygotowany przez nas 200-stronicowy dokument, możemy być pewni, że kiedy zobaczą działający system, zasypią nas żądaniami zmian.

Po drugie, pewnym problemem jest ograniczony potencjał wyrażania myśli w naszym języku. Wszystkie techniki prezentowania informacji w formie diagramów oraz metody formalne opierają się na zapisach dotyczących wykonywanych operacji wyrażonych w konkretnym języku[2]. Praktyka pokazuje, że języki naturalne nie najlepiej sprawdzają się w tej roli. Wystarczy spojrzeć na słownictwo stosowane w dowolnej umowie — prawnicy dążący do maksymalnej precyzji posługują się wyjątkowo nienaturalnym językiem.

Zachęcamy do prostego eksperymentu. Spróbujmy napisać krótki tekst, który wyjaśni odbiorcy, jak wiązać sznurowadła. Do dzieła!

Każdy, kto ma z tym zadaniem podobne problemy do nas, napisze: „Owiń teraz kciuk i palec wskazujący, tak aby wolny koniec wszedł pod lewe sznurowadło…" lub coś równie niezrozumiałego. To zadziwiająco trudne zadanie. Co ciekawe, większość z nas wiąże buty, w ogóle nie myśląc o tej czynności.

WSKAZÓWKA NR 57

Niektóre rzeczy lepiej robić, niż o nich mówić.

I wreszcie po trzecie, istnieje problem kaftana bezpieczeństwa. Projekt, który nie pozostawia kodującemu żadnego pola do implementacji, uniemożliwia mu pełne pokazanie swoich umiejętności. Niektórzy twierdzą, że właśnie takie rozwiązanie jest najlepsze, ale nie mają racji. Często właśnie podczas kodowania ujawniają się pewne potencjalne opcje. Nierzadko podczas kodowania myślimy sobie: **„Ciekawe, ponieważ zakodowałem tę funkcję w ten sposób, mógłbym uzupełnić ją o pewne dodatkowe rozwiązanie dosłownie w pięć minut"** lub **„Specyfikacja mówi, że mam zrobić to i to, ale niemal identyczne rezultaty mogę osiągnąć w inny sposób, poświęcając na to dwa razy mniej czasu"**. Nie powinniśmy, oczywiście, zbyt pochopnie wprowadzać zmian w projekcie, ale warto pamiętać, że nawet nie dostrzeglibyśmy wspomnianych okazji, gdy ten projekt był zbyt precyzyjny.

Jako pragmatyczni programiści powinniśmy postrzegać gromadzenie wymagań, projektowanie i implementacje jako odmienne aspekty tego samego procesu — procesu dostarczania systemu wysokiej jakości. Nie warto inwestować w środowiska, gdzie zbieranie wymagań, pisanie specyfikacji i samo kodowanie ma postać odrębnych, odizolowanych czynności. Powinniśmy raczej wdrażać modele łączące te elementy, gdzie specyfikacja i implementacja stanowią tylko różne aspekty jednego procesu identyfikacji i kodowania wymagań. Każda z tych czynności powinna prowadzić wprost do następnej bez sztucznych granic. Szybko

[2] Istnieją co prawda formalne metody algebraicznego wyrażania operacji, jednak rzadko są stosowane w praktyce. Tego rodzaju techniki wciąż wymagają od analityka tłumaczenia poszczególnych zapisów użytkownikom końcowym.

odkryjemy, że właściwy proces wytwarzania oprogramowania zachęca jego uczestników do uwzględniania wniosków płynących z implementacji i testów w procesie przygotowywania specyfikacji.

Dla jasności podkreślamy, że nie jesteśmy przeciwnikami generowania specyfikacji. Przeciwnie — zdajemy sobie sprawę z tego, że w pewnych przypadkach nawet najbardziej szczegółowe specyfikacje są niezbędne (na przykład dla jasności kontraktu, z uwagi na specyficzne środowisko, w którym pracujemy, lub z powodu nietypowego charakteru tworzonego produktu)[3]. Musimy przy tym mieć świadomość, że zapisując coraz bardziej szczegółowe specyfikacje, prędzej czy później osiągniemy punkt, od którego dalsze uszczegóławianie tych zapisów nie będzie przynosiło żadnych korzyści lub wręcz będzie powodowało straty. Powinniśmy też unikać budowy specyfikacji ponad innymi specyfikacjami bez uprzedniego opracowywania implementacji czy choćby prototypów — bardzo łatwo zapisać w specyfikacji rozwiązania, których w praktyce nie będzie można zbudować.

Im dłużej trwa tworzenie specyfikacji i im bardziej ten proces jest wykorzystywany w roli tarczy chroniącej programistów przed przerażającym zadaniem pisania kodu, tym trudniej przystąpić do właściwego kodowania. Nie możemy wpadać w spiralę specyfikacji — w pewnym momencie musimy zacząć kodować! Jeśli odkrywamy, że nasz zespół przyjął wygodną postawę pisania specyfikacji w nieskończoność, musimy to przerwać. Warto wówczas rozważyć opracowanie prototypów lub zastosowanie modelu pocisków smugowych.

Pokrewne podrozdziały

- „Pociski smugowe” w rozdziale 2.

Wyzwania

- Przytoczony wcześniej przykład sznurowadeł jest interesującą ilustracją problemu wyrażania prostych czynności słowami. Czy nie warto byłoby opisać ten proces za pomocą diagramów zamiast słów? A może zastosować zdjęcia? Może warto skorzystać z jakiejś formalnej notacji zaczerpniętej z topologii? Może sprawdziłyby się modele z drucianymi sznurowadłami? Jak nauczyłbyś tej czynności małe dziecko?

  Obraz jest czasem wart więcej niż dowolna liczba słów. Innym razem obraz jest bezwartościowy. Czy w razie stwierdzenia, że budowana specyfikacja jest zbyt szczegółowa, obrazy lub specjalne notacje mogą w czymś pomóc? Jak szczegółowe powinny być same obrazy lub notacje? Czy narzędzie graficzne byłoby lepsze od fizycznej tablicy?

---

[3] Zapisywanie szczegółowych specyfikacji jest, oczywiście, uzasadnione w przypadku systemów, od których zależy ludzkie życie. Wydaje się, że podobne zasady powinny dotyczyć także interfejsów i bibliotek tworzonych z myślą o innych programistach. Jeśli jedynym efektem naszej pracy ma być zbiór wywołań funkcji, powinniśmy zrobić wszystko, aby te wywołania były precyzyjnie zdefiniowane.

40 Okręgi i strzałki

[zdjęcia] z okręgami i strzałkami oraz jednym zdaniem wyjaśnienia na drugiej stronie będą świadczyć przeciwko nam…

Arlo Guthrie, *Alice's Restaurant*

Od czasów programowania strukturalnego, przez koncepcje zespołów pod wodzą głównego programisty, narzędzia CASE, wytwarzanie kaskadowe, model spirali, propozycje Jacksona, diagramy ER, chmury Boocha, technikę OMT, obiektowość, metodę Coada-Yourdona, aż po współczesne diagramy UML informatycy nigdy nie mogli narzekać na brak metod tworzonych z myślą o upodabnianiu ich pracy do przedsięwzięć inżynierskich. Każda metoda znalazła wiernych wyznawców i cieszyła się popularnością w pewnym okresie. Niedługo potem każda była zastępowana przez następną. Spośród wszystkich wymienionych metod chyba tylko pierwsza — programowanie strukturalne — istniała naprawdę długo.

Mimo to niektórzy programiści, dryfując po morzu zatopionych projektów, kurczowo trzymają się najnowszych odkryć i metodyk. Przypominają przerażonych marynarzy łapiących pływające deski. Każdy nowy element pojawiający się na powierzchni traktują jako szansę na poprawę swojej sytuacji. Ostatecznie jednak okazuje się, że niezależnie od jakości pływających szczątków statku programiści wciąż dryfują bez celu.

Nie chcemy być źle zrozumiani. Lubimy (niektóre) formalne techniki i metody. Uważamy jednak, że bezmyślne wdrażanie każdej nowej techniki bez analizy jej przydatności w kontekście praktyk wytwarzania i własnych możliwości jest najkrótszą drogą do rozczarowania.

WSKAZÓWKA NR 58

Nie możemy być niewolnikami formalnych metod.

Formalne metody mają pewne poważne ograniczenia.

- Większość formalnych metod wymusza zbieranie wymagań przy użyciu kombinacji diagramów i jakichś form dodatkowych opisów. Tworzone w ten sposób obrazy reprezentują wymagania, tak jak rozumieją je programiści. Okazuje się jednak, że w wielu przypadkach takie diagramy są zupełnie niezrozumiałe dla użytkowników końcowych, zatem projektanci muszą je dodatkowo interpretować. Właśnie dlatego nie istnieją rzeczywiste, formalne techniki weryfikacji wymagań przez docelowych użytkowników przyszłego systemu — wszystko opiera się na wyjaśnieniach projektantów, a więc dokładnie tak jak w tradycyjnych dokumentach z wymaganiami. Dostrzegamy pewne zalety takiego sposobu gromadzenia wymagań, jednak zdecydowanie wolimy (o ile to możliwe) przekazywanie użytkownikowi prototypu, z którym sam będzie mógł eksperymentować.

- Metody formalne pozornie zachęcają do specjalizacji. Jedna grupa ludzi pracuje nad modelem danych, inna dba o architekturę, a pracownicy odpowiedzialni za gromadzenie wymagań przygotowują przypadki użycia (lub ich odpowiedniki). Z doświadczenia wiemy, że taki model pracy utrudnia komunikację i prowadzi do straty czasu. Co więcej, przytoczony podział ról wiąże się z ryzykiem postawy „my kontra oni" na linii projektanci – programiści. Wolimy model, w którym każdy rozumie cały system, nad którym pracuje. Zgromadzenie szczegółowej wiedzy na temat każdego aspektu systemu nie zawsze jest możliwe, ale powinniśmy przynajmniej wiedzieć, jak przebiega interakcja poszczególnych komponentów, gdzie są przechowywane dane i jakie są wymagania.
- Wszyscy lubimy pisać systemy dynamiczne, które dostosowują się do warunków i które pozwalają zmieniać charakter aplikacji w czasie wykonywania (za pomocą metadanych). Większość współczesnych metod formalnych łączy obiekty statyczne lub modele danych z rozmaitymi mechanizmami zdarzeń lub czynności. Nie znaleźliśmy jednak żadnej metody umożliwiającej ilustrowanie dynamiki, której oczekujemy od naszych systemów. W praktyce większość formalnych metod wyznacza zupełnie przeciwny kierunek, zachęcając nas do definiowania statycznych relacji łączących obiekty, których interakcja powinna mieć charakter dużo bardziej dynamiczny.

Czy te metody się opłacają?

W 1999 roku w swoim artykule dla miesięcznika CACM [Gla99b] Robert Glass dokonał przeglądu badań nad wzrostem produktywności i jakości oferowanym przez siedem różnych technologii wytwarzania oprogramowania (języki programowania czwartej generacji, technologie strukturalne, narzędzia CASE, metody formalne, metodyki tzw. czystego pokoju, modele procesów oraz techniki obiektowe). Wykazał, że początkowy zachwyt towarzyszący wszystkim wymienionym metodom był nieuzasadniony. Stosowanie niektórych spośród tych metod przynosi co prawda pewne korzyści, jednak zwykle można je dostrzec dopiero po początkowym okresie spadku produktywności i jakości, kiedy nowa technika jest wdrażana i poznawana przez użytkowników. Nigdy nie należy lekceważyć kosztów wdrażania nowych narzędzi i metod. Musimy być przygotowani na traktowanie pierwszych projektów realizowanych przy użyciu tych technik jako procesów poznawczych.

Czy powinniśmy stosować formalne metody?

Oczywiście. Zawsze powinniśmy pamiętać, że formalne metody wytwarzania to tylko kolejne narzędzia w naszym zestawie. Jeśli po uważnej analizie czujemy potrzebę zastosowania jakiejś formalnej metody, możemy iść tą drogą — musimy jednak pamiętać, kto podejmuje decyzje. Nie możemy dopuścić do sytuacji, w której będziemy niewolnikami tej czy innej metodyki. Okręgi i strzałki kiepsko

sprawdzają się w roli przełożonych. Pragmatyczni programiści krytycznym okiem patrzą na metodyki, po czym wyciągają z każdej z nich to, co najlepsze, przekształcając je w zbiór sprawdzonych praktyk, które z każdym miesiącem zyskują na jakości. To klucz do sukcesu. Powinniśmy ustawicznie poprawiać i doskonalić swoje procesy. Nigdy nie powinniśmy akceptować skostniałych ograniczeń narzucanych przez metody jako ograniczenia własnego świata.

Nie możemy bezmyślnie przyjmować fałszywych opinii o poszczególnych metodach. Nawet jeśli ludzie przynoszą na spotkania wielkie płachty z diagramami klas i po 150 przypadków użycia, cała ta masa papieru wciąż jest tylko zawodną interpretacją wymagań i projektu. Kiedy analizujemy efekt pracy jakiegoś narzędzia, powinniśmy przynajmniej spróbować zapomnieć, ile to narzędzie kosztowało.

WSKAZÓWKA NR 59

Drogie narzędzia nie generują lepszych projektów.

Formalne metody z pewnością mają swoje miejsce w świecie wytwarzania oprogramowania. Jeśli jednak obserwujemy projekt realizowany według filozofii „diagram klas to aplikacja, reszta to tylko mechaniczne kodowanie", możemy być pewni, że mamy do czynienia zespołem zmierzającym wprost ku klęsce.

Pokrewne podrozdziały

- „Kopalnia wymagań" w rozdziale 7.

Wyzwania

- Diagramy przypadków użycia UML wchodzą w skład procesu gromadzenia wymagań (patrz podrozdział „Kopalnia wymagań" w rozdziale 7.). Czy takie diagramy są efektywnym sposobem komunikacji z użytkownikami? Jeśli nie, dlaczego ich używasz?
- Jak stwierdzić, czy jakaś formalna metoda przynosi korzyści Twojemu zespołowi? Co można mierzyć? Co jest traktowane jako poprawa? Czy potrafisz odróżnić korzyści wynikające ze stosowania tego narzędzia od zwykłych skutków rosnącego doświadczenia członków zespołu?
- Gdzie leży punkt rentowności dla wprowadzania nowych metod w Twoim zespole? Jak szacujesz przyszłe korzyści względem bieżących spadków produktywności (na etapie wprowadzania nowego narzędzia)?
- Czy narzędzia, które sprawdzają się w przypadku wielkich projektów, okazują się równie dobre w mniejszej skali? Czy w przypadku odwrotnej relacji jest podobnie?

Skorowidz

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

Twórcy rozmaitych narzędzi programistycznych nieustannie próbują nas przekonać o niewiarygodnych możliwościach swoich produktów, a specjaliści od metodyk obiecują, że to właśnie ich techniki zagwarantują nam największą wydajność. Każdy oczywiście twierdzi, że jego język programowania jest najlepszy... A jak wszyscy doskonale wiemy, w naszej pracy nie istnieją NAJLEPSZE rozwiązania — są tylko rozwiązania NAJLEPIEJ sprawdzające się w danym projekcie. Większy wpływ na efektywność naszej pracy ma więc doświadczenie oraz znajomość różnych, sprawdzonych praktyk wytwarzania oprogramowania. Zawodowcy, którym na sercu leży przede wszystkim jakość realizowanych projektów, są zwykle zgodni — nigdy nie wiążą swojej zawodowej kariery z jedną, konkretną technologią. To jedna z cech wyróżniających pragmatycznych programistów — produktywnych specjalistów, którzy w pełni wykorzystują swój potencjał i szybko osiągają zawodowy sukces. A oto pierwsza książka, która w pełni odsłania system ich codziennej pracy!

Nie ma znaczenia, czy jesteś wolnym strzelcem, członkiem wielkiego zespołu projektowego, czy konsultantem równocześnie współpracującym z wieloma klientami. Ta skoncentrowana na przekazywaniu praktycznej wiedzy publikacja pokaże Ci, jak efektywnie wykorzystywać swoje umiejętności i doświadczenie do sprawnej realizacji nawet najbardziej złożonych projektów. Podręcznik ilustruje najlepsze praktyki i najczęstsze pułapki wielu różnych aspektów wytwarzania oprogramowania. Znajdziesz w nim zarówno zagadnienia związane ze strategicznym planowaniem swojego zawodowego rozwoju, jak i techniki takiego projektowania architektury, aby przyszły kod był elastyczny, łatwy w dostosowywaniu do różnych okoliczności i przygotowany do wielokrotnego użytku.

Z książki dowiesz się między innymi, jak:

- unikać pułapki powielania wiedzy
- pisać elastyczny, dynamiczny i łatwy w dostosowywaniu kod
- unikać programowania przez koincydencję
- zabezpieczać kod za pomocą kontraktów, asercji i wyjątków
- gromadzić rzeczywiste wymagania
- bezlitośnie i efektywnie testować oprogramowanie
- zachwycać swoich użytkowników
- budować zespoły pragmatycznych programistów
- automatyzować pracę w celu zapewnienia większej precyzji

Gdybym organizował teraz jakiś projekt, zrobiłbym wszystko, aby zatrudnić autorów tej książki. (...) A gdyby to się nie udało, szukałbym ludzi, którzy przynajmniej przeczytali ich książkę.

Ward Cunningham, słynny amerykański programista, pionier w dziedzinie wzorców projektowych oraz programowania ekstremalnego

Cena 59,00 zł

ISBN 978-83-246-3237-4

9 788324 632374