James O. **Coplien**, Gertrud **Bjørnvig**

ARCHITEKTURA LEAN W PROJEKTACH AGILE

Helion

Tytuł oryginału: Lean Architecture: for Agile Software Development

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-8672-8

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/arlean
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

O autorach **11**

Wstęp **13**

1. Wprowadzenie **19**
1.1. Podwaliny: Lean i Agile 19
1.2. Architektura Lean i wytwarzanie funkcji zgodnie z metodyką Agile 22
1.3. Produkcja Agile 24
1.3.1. Agile bazuje na architekturze Lean 24
1.3.2. Zakres systemów Agile 25
1.3.3. Agile i DCI 26
1.4. Książka w bardzo małej pigułce 27
1.5. Lean i Agile: kontrastujące i uzupełniające 28
1.5.1. Sekret Lean 30
1.6. Utracone praktyki 30
1.6.1. Architektura 31
1.6.2. Obsługa zależności pomiędzy wymaganiami 31
1.6.3. Podstawy użyteczności 31
1.6.4. Dokumentacja 32
1.6.5. Zdrowy rozsądek, myślenie i opieka 35
1.7. O czym ta książka nie jest? 36
1.8. Agile, Lean, Scrum i inne metodologie 37
1.9. Rys historyczny 38

2 . Produkcja Agile w pigułce **41**
2.1. Zaangażuj interesariuszy 41
2.2. Zdefiniowanie problemu 43
2.3. Czym system jest: podstawa formy 43
2.4. Czym system jest: siła napędowa systemu 45
2.5. Projekt i kodowanie 46
2.6. Odliczanie: 3, 2, 1... 47

3. Zaangażowanie interesariuszy 49
3.1. Strumień wartości 49
3.1.1. Użytkownicy końcowi i inni interesariusze jako kotwice strumienia wartości 50
3.1.2. Architektura w ramach strumienia wartości 51
3.1.3. Sekret Lean 52
3.2. Najważniejsi interesariusze 54
3.2.1. Użytkownicy docelowi 56
3.2.2. Biznes 60
3.2.3. Klienci 61
3.2.4. Eksperci dziedzinowi 64
3.2.5. Deweloperzy i testerzy 66
3.3. Elementy procesu angażowania interesariuszy 68
3.3.1. Od czego zacząć? 68
3.3.2. Zaangażowanie klienta 70
3.4. Sieć interesariuszy: eliminowanie marnotrawstwa czasu 71
3.4.1. Linia produkcyjna czy rój? 71
3.4.2. Pierwsza rzecz, którą należy zbudować 73
3.4.3. Utrzymuj jedność zespołu 74
3.5. Nie ma szybkich rozwiązań, ale jest nadzieja 75

4. Definiowanie problemu 77
4.1. Jakie cechy Agile mają definicje problemów? 78
4.2. Jakie cechy Lean mają definicje problemów? 78
4.3. Dobre i złe definicje problemów 79
4.4. Problemy i rozwiązania 81
4.5. Proces definiowania problemów 82
4.5.1. Ceń bardziej polowanie niż nagrodę 82
4.5.2. Własność problemu 83
4.5.3. Przerost funkcjonalności 84
4.6. Definicje problemu, cele, czartery, wizje i zamierzenia 84
4.7. Dokumentacja? 85

5. Czym jest system? Część I. Architektura Lean 87
5.1. Kilka niespodzianek o architekturze 88
5.1.1. Co Lean ma z tym wspólnego? 90
5.1.2. Co Agile ma wspólnego z architekturą? 91
5.2. Pierwszy krok w projekcie: podział na części 94
5.2.1. Pierwszy podział: forma dziedzinowa a forma behawioralna 95
5.2.2. Drugi podział: prawo Conwaya 96
5.2.3. Rzeczywista złożoność podziału systemu 98
5.2.4. Wymiary złożoności 99
5.2.5. Dziedziny. Wyjątkowy podział 99
5.2.6. Wracamy do wymiarów złożoności 101
5.2.7. Architektura i kultura 104
5.2.8. Wnioski na temat prawa Conwaya 105

5.3. Drugi krok w projekcie: wybór stylu projektu ... 105
5.3.1. Struktura a podział ... 106
5.3.2. Podstawy stylu: części stałe i zmienne ... 107
5.3.3. Zaczynamy od oczywistych części wspólnych i różnic ... 108
5.3.4. Części wspólne, różnice i zakres ... 111
5.3.5. Jawne wyrażanie części wspólnych i różnic ... 113
5.3.6. Najpopularniejszy styl: programowanie obiektowe ... 116
5.3.7. Inne style w obrębie świata von Neumanna ... 118
5.3.8. Języki dziedzinowe i generatory aplikacji ... 120
5.3.9. Formy skodyfikowane: języki wzorców ... 123
5.3.10. Oprogramowanie dostawców zewnętrznych i inne paradygmaty ... 124
5.4. Dokumentacja? ... 127
5.4.1. Słownik dziedziny ... 127
5.4.2. Przenoszenie architektury ... 128
5.5. Tło historyczne ... 128

6. Czym jest system? Część II. Kodowanie ... 131
6.1. Krok trzeci: szkic kodu ... 131
6.1.1. Abstrakcyjne klasy bazowe ... 132
6.1.2. Warunki wstępne, warunki końcowe i asercje ... 136
6.1.3. Skalowanie algorytmów: druga strona statycznych asercji ... 142
6.1.4. Forma a dostępne usługi ... 143
6.1.5. Rusztowanie ... 144
6.1.6. Testowanie architektury ... 146
6.2. Relacje w architekturze ... 149
6.2.1. Typy relacji ... 149
6.2.2. Testowanie relacji ... 150
6.3. Programowanie obiektowe „po nowemu” ... 151
6.4. Ile architektury? ... 153
6.4.1. Równowaga pomiędzy BUFD a YAGNI ... 154
6.4.2. Jeden rozmiar nie pasuje wszystkim ... 154
6.4.3. Kiedy architektura jest gotowa? ... 156
6.5. Dokumentacja? ... 156
6.6. Tło historyczne ... 157

7. Co system robi: funkcje systemu ... 159
7.1. Co system robi? ... 160
7.1.1. Opowieści użytkowników: początek ... 160
7.1.2. Wykorzystanie specyfikacji i przypadków użycia ... 161
7.1.3. Pomoc należy się także programistom ... 162
7.1.4. Kilometraż nie zawsze jest taki sam ... 163
7.2. Kto będzie korzystać z naszego oprogramowania? ... 164
7.2.1. Profile użytkowników ... 164
7.2.2. Osoby ... 164
7.2.3. Profile użytkowników czy osoby? ... 165
7.2.4. Role użytkowników i terminologia ... 165

7.3. Do czego użytkownicy chcą wykorzystać nasze oprogramowanie? ... 166
7.3.1. Lista własności ... 166
7.3.2. Diagramy przepływu danych ... 166
7.3.3. Osoby i scenariusze ... 167
7.3.4. Narracje ... 167
7.3.5. Projektowanie aplikacji sterowane zachowaniami ... 167
7.3.6. Teraz, gdy jesteśmy rozgrzani... ... 168
7.4. Dlaczego użytkownicy chcą korzystać z naszego oprogramowania? ... 169
7.5. Konsolidacja tego, co system robi ... 170
7.5.1. Widok helikoptera ... 172
7.5.2. Ustawianie sceny ... 177
7.5.3. Odtwarzanie scenariusza słonecznego dnia ... 178
7.5.4. Dodawanie interesujących rzeczy ... 183
7.5.5. Od przypadków użycia do ról ... 191
7.6. Podsumowanie ... 193
7.6.1. Wsparcie przepływu pracy użytkowników ... 193
7.6.2. Wsparcie dla testów blisko prac rozwojowych ... 193
7.6.3. Wsparcie dla wydajnego podejmowania decyzji na temat funkcjonalności ... 194
7.6.4. Wsparcie dla nowo powstających wymagań (odchyleń) ... 194
7.6.5. Wsparcie dla planowania wydań ... 194
7.6.6. Uzyskanie danych wejściowych do opracowania architektury ... 195
7.6.7. Budowanie w zespole zrozumienia przedmiotu pracy ... 195
7.7. „To zależy": kiedy przypadki użycia nie są dobre? ... 196
7.7.1. Klasyczne programowanie obiektowe: architektury atomowych zdarzeń ... 196
7.8. Testowanie użyteczności ... 197
7.9. Dokumentacja? ... 198
7.10. Tło historyczne ... 200

8. Kodowanie: podstawowy montaż ... 201

8.1. Obraz z góry: wzorzec projektowy Model-View-Controller-User ... 201
8.1.1. Czym jest program? ... 202
8.1.2. Czym jest program Agile? ... 203
8.1.3. MVC bardziej szczegółowo ... 204
8.1.4. MVC-U: to nie koniec opowieści ... 205
8.2. Forma i architektura systemów zdarzeń atomowych ... 208
8.2.1. Obiekty dziedziny ... 208
8.2.2. Role obiektów, interfejsy i Model ... 208
8.2.3. Refleksje: przypadki użycia, architektury zdarzeń atomowych i algorytmy ... 211
8.2.4. Przypadek specjalny: odwzorowanie ról obiektów na obiekty typu jeden do wielu ... 212
8.3. Aktualizacja logiki dziedziny: rozwijanie metod, faktoryzacja i refaktoryzacja ... 212
8.3.1. Tworzenie nowych klas i wypełnianie istniejących namiastek funkcji ... 213
8.3.2. Powrót do przyszłości: po prostu stare, dobre programowanie obiektowe ... 215
8.3.3. Narzędzia analizy i projektowania ... 215
8.3.4. Faktoryzacja ... 216
8.3.5. Uwaga na refaktoryzację ... 216
8.4. Dokumentacja? ... 217

8.5. Do czego te wszystkie artefakty? ... 217
8.6. Tło historyczne ... 218

9. Kodowanie: architektura DCI ... 219
9.1. Czasami inteligentne obiekty po prostu są niewystarczające ... 219
9.2. DCI w pigułce ... 220
9.3. Przegląd architektury DCI ... 221
9.3.1. Części modelu mentalnego użytkownika końcowego, o których zapomnieliśmy ... 221
9.3.2. Wprowadzenie ról obiektowych z metodami ... 223
9.3.3. Sztuczki z cechami ... 225
9.3.4. Klasy kontekstu: jedna w każdym przypadku użycia ... 226
9.4. DCI na przykładzie ... 229
9.4.1. Dane wejściowe do projektu ... 229
9.4.2. Od przypadków użycia do algorytmów ... 230
9.4.3. Bezmetodowe role obiektów: framework dla identyfikatorów ... 232
9.4.4. Podział algorytmów pomiędzy role obiektowe z metodami ... 234
9.4.5. Framework kontekstu ... 241
9.4.6. Warianty i sztuczki w architekturze DCI ... 259
9.5. Aktualizacja logiki dziedziny ... 261
9.5.1. Porównanie DCI ze stylem architektury zdarzeń atomowych ... 261
9.5.2. Szczególne aspekty logiki dziedzinowej w architekturze DCI ... 263
9.6. Obiekty kontekstu w modelu mentalnym użytkownika końcowego: rozwiązanie odwiecznego problemu ... 265
9.7. Do czego te wszystkie artefakty? ... 269
9.8. Nie tylko C++: DCI w innych językach ... 272
9.8.1. Scala ... 272
9.8.2. Python ... 273
9.8.3. C# ... 273
9.8.4. ...a nawet w Javie ... 273
9.8.5. Przykład z rachunkami w Smalltalku ... 274
9.9. Dokumentacja? ... 274
9.10. Tło historyczne ... 275
9.10.1. DCI a programowanie aspektowe ... 275
9.10.2. Inne podejścia ... 276

10. Epilog ... 277

A. Implementacja przykładu architektury DCI w Scali ... 279

B. Przykład implementacji rachunków w Pythonie ... 283

C. Przykład implementacji rachunków w C# ... 287

D. Przykład implementacji rachunków w Ruby ... 291

E. Qi4j ... 297

F. Przykład implementacji rachunków w Squeaku ... 299

Bibliografia ... 307

Skorowidz ... 317

ROZDZIAŁ

7

Co system robi: funkcje systemu

Jednym z zaskakujących produktów ubocznych procesu planowania scenariuszy jest zwiększająca się odpowiedzialność.
The Clock of the Long Now*, s. 118.*

Każdy system ma dwa projekty: projekt tego, co system *robi*, oraz projekt tego, czym system *jest*. Użytkowników końcowych najbardziej interesują usługi, które realizuje oprogramowanie, a te prawie zawsze należą do kategorii co-system-robi. Jednak przez interfejs usług ujawnia się model dziedziny. Przypomnijmy sobie, jak ważna dla programowania obiektowego jest metafora bezpośredniej manipulacji. Ponadto ze względu na dążenie do obniżania długoterminowych kosztów (co jest dobre z punktu widzenia klientów i użytkowników końcowych) oraz przygotowanie się na zmiany (oczekiwane zarówno przez klientów, jak i większość końcowych użytkowników) producent (którego jesteśmy przedstawicielami) także chce rozpocząć od dobrej ogólnej formy całego systemu, a ta forma jest ugruntowana w części czym-system-jest. Na początku projektu trzeba skupić się na obu tych aspektach. Ta koncentracja na dwóch krańcach nie tylko odnosi się do dobrego początku, ale i jest podstawą długoterminowej kondycji produktu.

Ważne jest, aby zawsze pamiętać, że sednem tego, co oprogramowanie dostarcza, są usługi, a usługi dotyczą działania. W przeciwieństwie do budynków komputery działają w ludzkich skalach czasowych. Ta dynamika jest kluczem do ich roli w życiu. Luke Hohmann uważa, że zamiast sięgać do architektury budynków jako metafory projektowania systemów oprogramowania, powinniśmy raczej skorzystać z innego rodzaju sztuki: tańca. Podobnie jak architektura, taniec dotyczy formy, ale jest żywy i dynamiczny. Rozdziały 5. i 6. przygotowały nam „salę balową”. Teraz możemy przejść do tańca.

Biorąc pod uwagę, że jest to książka poświęcona Agile, można się spodziewać, że więcej użytkowników będzie przedkładać część o tym, co system robi, nad część poświęconą czystej architekturze. W idealnym świecie postąpilibyśmy w taki sposób: prowadzilibyśmy projekt wyłącznie według dążeń użytkownika końcowego, nie zajmując się niczym poza tym. Analizujemy trochę formę dziedziny przed szczegółowym omówieniem funkcjonalności szczegółowo, ponieważ taki sposób zapewnia lepszą wydajność. Jeśli terminologia związana z architekturą wspiera wzajemne zrozumienie pomiędzy dostawcą a użytkownikiem, możemy z ufnością przejść do funkcjonalności oraz zająć się poprawianiem długoterminowej elastyczności.

7.1. Co system robi?

Aby zdecydować, co system robi, najpierw musimy znać odpowiedzi na pytania: *kto?*, *co?* i *dlaczego? Kto* będzie korzystać z naszego oprogramowania? *Co* będzie z nim robić? I *dlaczego* to robi? W świecie wytwarzania oprogramowania zgodnie z metodyką Agile popularne *opowieści użytkowników* formułuje się wokół tej mantry „kto, co i dlaczego". „Jako posiadacz rachunku (kto) chcę przelewać pieniądze pomiędzy moimi rachunkami (co) po to, by mieć pewność, że na żadnym z rachunków nie powstanie debet (dlaczego)".

7.1.1. Opowieści użytkowników: początek

Opowieści użytkowników są dobrym punktem wyjścia, ale prawdopodobnie nie są dość dobre, by zapewnić dobry rezultat końcowy. Co właściwie wiemy o posiadaczu rachunku? Jeśli to ja mam być posiadaczem rachunku, to będę korzystał z konta internetowego do realizowania przelewów. Jeśli posiadaczem rachunku jest mój 11-letni syn, to ma on tylko jeden rachunek i nie będzie potrzebował tej funkcjonalności. Jeśli posiadaczem rachunku jest moja 85-letnia ciocia, to pójdzie do banku i poprosi pracownika, aby zrealizował dla niej przelew. Jeżeli posiadacz rachunku nie może sam pójść do banku i nie ma dostępu do internetu, telefon do banku będzie preferowanym rozwiązaniem. A zatem powinniśmy wiedzieć więcej na temat *kontekstu*. W rozdziale 4. zwróciliśmy uwagę na znaczenie kontekstu w formułowaniu problemu do rozwiązania. Czy posiadacz rachunku jest użytkownikiem internetu i będzie realizował przelewy z domu? Czy posiadacz rachunku korzysta z pośrednika w banku do realizacji przelewu? Czy bank chce udostępnić tę usługę z poziomu swoich bankomatów?

Przelew pieniędzy powinien być prosty. Użytkownik wybiera rachunek źródłowy, rachunek docelowy i kwotę. System przeprowadza proste obliczenia i aktualizuje saldo na obu rachunkach. Dwa kroki i na tym koniec. Prawie... Po prostu powinniśmy wiedzieć *kilka* dodatkowych rzeczy: czy oba rachunki muszą być w tym samym banku? Jakie będą konsekwencje przekazania pieniędzy z rachunku, jeśli po operacji pozostanie na nim ujemne saldo? Czy przelew powinien odbyć się natychmiast? Czy użytkownik może zaplanować realizację przelewu w przyszłości? Czy przyszłość może oznaczać „za 10 lat?". Co powiemy posiadaczowi rachunku mającemu tylko jeden rachunek, a próbującemu przelać pieniądze? (Błąd?) Czy dzieci powinny mieć dostęp do tej funkcji? Czy użytkownik może zlecić przelew cykliczny, na przykład raz na miesiąc?

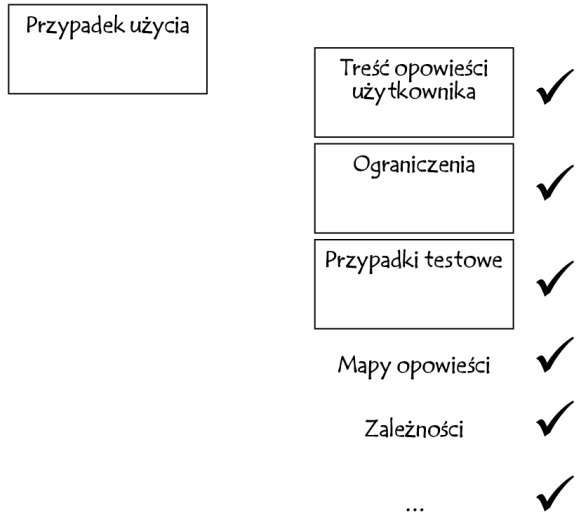
Ktoś, kto od lat pracuje w określonym banku i posiada głęboką wiedzę dziedzinową, być może zna odpowiedź na wiele z tych pytań. Ale strategie bankowe są ulotne i nadążenie za aktualnymi zmianami może być trudne. Projektantowi lub nawet architektowi zazwyczaj ktoś dostarcza odpowiedzi na pytania dotyczące wybranej dziedziny. Może nam się wydawać, że implementacja przelewu cyklicznego nie jest trudna, a jej zakodowanie nie zajmie zbyt dużo czasu. Ale to wymaga również prawidłowego interfejsu użytkownika. Trzeba przeprowadzić testy. Powstają nowe pytania w stylu: w jaki sposób poinformować posiadacza rachunku o tym, że nie ma wystarczających środków do zrealizowania przelewu? Jest to zatem decyzja biznesowa, a nie decyzja bazująca na interpretacji opowieści użytkownika z punktu widzenia programisty.

Moglibyśmy również przeanalizować perspektywę *dlaczego*. Być może posiadacz rachunku przelewa pieniądze, ponieważ na innym rachunku jest lepsze oprocentowanie albo ze względu na korzyści podatkowe, które wynikają z przelewu pieniędzy na inny rachunek na początku nowego roku. Czy to ma znaczenie? Nie wiemy tego na pewno, dopóki nie przeprowadzimy analizy. Wiemy jednak, że ukryte założenia projektanta mają ogromny wpływ na projekt interfejsu użytkownika.

7.1.2. Wykorzystanie specyfikacji i przypadków użycia

Powyższe pytania pokazują to, co stało się powszechnym doświadczeniem wytwarzania oprogramowania zgodnie z metodyką Agile: *opowieści użytkowników nie są wystarczające*. Liderzy Agile uzupełnili przypadki użycia przypadkami testowymi (Cohn 2004, s. 7) oraz ograniczeniami i regułami biznesowymi (Cohn 2004, s. 77). Zalecają nawet wykorzystanie narzędzi w celu przezwyciężenia ograniczeń kart (Cohn 2004, s. 179). A opowieści można pogrupować w obszary biznesowe wyższego poziomu, podobnie do scenariuszy grupowania przypadków użycia (Patton 2009). W miarę jak społeczność Agile zdobywała doświadczenie w ciągu ostatnich dziesięciu lat, współczesne opowieści użytkowników w rzeczywistości stały się przypadkami użycia (rysunek 7.1). Alistair Cockburn uzasadnia, dlaczego w dalszym ciągu używa przypadków użycia (Cockburn 2008):

> Opowieści użytkowników i pozycje rejestru wymagań nie dają projektantom *kontekstu*, który stworzyłby podstawę do pracy (...) nie dają zespołowi projektowemu poczucia „kompletności" (...) nie dostarczają wystarczająco dobrego mechanizmu *patrzenia w przyszłość* na trudności wynikające z prac, które będą wykonywane.
>
> (...)
>
> Przypadki użycia są rzeczywiście „cięższe" i trudniejsze niż zarówno opowieści użytkowników, jak i elementy rejestru wymagań, ale wnoszą wartość w zamian za ten dodatkowy ciężar... W szczególności przypadki użycia rozwiązują następujące trzy problemy.



Rysunek 7.1. Przypadki użycia integrują wiele dodatków do opowieści użytkowników

1. Lista nazw celów zapewnia kierownictwu krótkie podsumowanie tego, co system zaoferuje użytkownikom biznesowym i użytkownikom końcowym...
2. Główny scenariusz sukcesu każdego przypadku użycia zapewnia porozumienie wszystkich zainteresowanych stron odnośnie do tego, co system będzie robił, a czasem, co ważniejsze, *czego nie zrobi*...
3. Rozszerzenia każdego przypadku użycia zapewniają analitykom wymagań framework do analizy niewielkich elementów, które niekiedy w niezrozumiały sposób zajmują 80% czasu i pieniędzy przeznaczonych na projekt...

4. Fragmenty opisujące rozszerzenia przypadków użycia dostarczają odpowiedzi na wiele szczegółowych, często trudnych pytań biznesowych zadawanych przez programistów... jest to framework (dokumentacja) myślenia...
5. Pełny zestaw przypadków użycia pokazuje, że analitycy przemyśleli wszystkie potrzeby użytkowników, zajęli się wszystkimi celami, jakie ma spełnić system, oraz wszystkimi wariantami...

Zwolennicy Agile odpowiadają, że opowieści użytkowników są jedynie „obietnicą przyszłej konwersacji". Prawdą jest, że prowadzenie takiej konwersacji jest ważne. Pytanie brzmi: co zrobimy z ustaleniami wynikającymi z takiej rozmowy? Jednym z efektów może być więcej opowieści użytkowników:

„Jako posiadacz rachunku chcę składać polecenia przelewów comiesięcznych, tak bym mógł opłacać rachunki z innego rachunku niż ten, na który wpływa moja pensja".

„Jako posiadacz rachunku chcę przelewać pieniądze innemu posiadaczowi rachunku, tak bym mógł płacić za wypożyczanie nart".

„Jako posiadacz rachunku chcę opłacać rachunki przez internet, abym nie musiał chodzić na pocztę".

Moglibyśmy także stworzyć opowieści użytkowników dla innych ról użytkowników, takich jak doradcy bankowi, księgowi lub organy podatkowe. Moglibyśmy przedstawić różne motywacje dla tej samej funkcji i mieć nadzieję na ich zintegrowanie ze sobą. Aby szybko uzyskać wiele opowieści użytkowników, można umieścić pięciu ekspertów dziedzinowych w pokoju i poprosić o przeprowadzenie burzy mózgów na ten temat. To jest dobry sposób na analizę wymagań. Ale w jaki sposób zorganizować 200 opowieści użytkowników utworzonych w zeszłym tygodniu? Trzeba stworzyć (najlepiej uporządkowaną) listę opartą na relacjach między nimi, ich kosztami, wartością biznesową i oczekiwanym czasem pojawienia się na rynku, tak aby można było podjąć właściwe decyzje biznesowe dotyczące rozpoczęcia pracy nad każdą z opowieści użytkowników.

Trzeba spróbować pomóc przedstawicielom biznesu, zanim sprawy skomplikują się za bardzo. Pomoc nie powinna polegać na tworzeniu „podopowieści" (dzieleniu dużych opowieści na mniejsze) czy też „epiki" (łączeniu mniejszych historii w większe). Nie ma powodu, aby odkrywać koło na nowo. W branży od dawna istnieją sprawdzone sposoby radzenia sobie z takimi problemami, a wiele z tych praktyk honoruje podejście zarówno Lean, jak i Agile.

7.1.3. Pomoc należy się także programistom

Większość dyskusji na temat wymagań skupia się na spełnieniu oczekiwań przedstawicieli biznesu. Chcemy, aby czuli, że przekazali nam coś, co daje obraz ich wizji, i aby byli w stanie to zrobić, nie wiedząc zbyt wiele o Ruby, C# czy Javie. Łatwo jest zapomnieć o kliencie tych wymagań: zespole projektowym.

Deweloper może teoretycznie sam przeprowadzić analizę i bezpośrednio uzyskać wymagania. Takie podejście może się sprawdzać w małych zespołach projektowych lub w nieformalnych relacjach biznesowych. Zazwyczaj jednak deweloperzy nie mają umiejętności wykonywania dobrej analizy i brakuje im wiedzy biznesowej niezbędnej do przyjęcia odpowiedzialności za formułowanie wymagań, które będą miały wpływ na kierunek biznesu. Wiemy przy tym — dotyczy to zarówno projektowania zgodnie z metodyką Agile, jak i innych metodyk — że przedstawiciele biznesu zbyt często przedwcześnie przekazują pracę zespołowi projektowemu. Zespół nie zawsze

wie, czego chce biznes, i czasami brakuje mu możliwości uzyskania istotnych wyjaśnień. Oczywiście, niemożliwe jest stworzenie wyczerpującej listy wymagań (z powodu powstawania nowych), więc zawsze potrzebne jest jakieś sprzężenie zwrotne, które pozwoli utrzymać rozwój produktu już po rozpoczęciu kodowania.

Nawet wtedy, gdy programista uzyska wymagania, powinno być możliwe szybkie rozpoczęcie prac rozwojowych. Pytania o wymagania nie powinny wynikać z nieadekwatności specyfikacji, ale raczej powinny pojawiać się jako „niespodzianki", które ujawniają się podczas szczegółowego projektowania lub kodowania. Dobre wymagania powinny *stwarzać możliwości* dla dewelopera. Specyfikacje stwarzające możliwości nie są jednak substytutem zasady *wszyscy*, *razem*, *od wczesnej fazy*: są one raczej dowodem na to, że sekret Lean jest rzeczywiście w użyciu.

W amerykańskim systemie patentowym wykorzystuje się termin *enabling specification* (z ang. *specyfikacja stwarzająca możliwości*) do opisania patentu, który jest na tyle przekonujący, aby można było rozpocząć produkcję bez zbyt wielu nowych badań. W tym samym sensie, jak patent umożliwia rzemieślnikowi odtworzenie wynalazku, tak wymagania powinny umożliwić członkom zespołu maksymalne wyeliminowanie konieczności ponownych badań tematu.

Jeff Sutherland opisuje koncepcję specyfikacji stwarzających możliwości w metodyce Scrum (Sutherland 2009). Pisze on między innymi:

> Okazuje się, że specyfikacja stwarzająca możliwości to dokładnie to, co jest potrzebne do maksymalizacji wydajności procesu realizacji opowieści użytkowników. Średnia wydajność procesu zespołów realizujących opowieści użytkownika wynosi około 20%. Oznacza to, że realizacja opowieści oszacowanej na jeden idealny dzień pracy zajmuje pięć dni kalendarzowych. Firma Systematic Software Engineering stosująca piąty stopień CMM (ang. *Capability Maturity Model*) posiada szerokie dane pokazujące, że zespoły, które uzyskają ponadpięćdziesięcioprocentową wydajność przetwarzania opowieści użytkownika, będą systematycznie podwajać swoją prędkość.

7.1.4. Kilometraż nie zawsze jest taki sam

W małym projekcie opowieści użytkownika wraz z przypadkami testowymi oraz mapą opowieści zdobiącą ścianę pomieszczenia zespołu mogą być wszystkim, czego potrzeba (Patton 2009). W tym rozdziale omówimy wiele sposobów realizacji części architektury odpowiadającej na pytanie, co system robi: opowieści użytkowników, przypadki użycia, funkcje, a nawet dobre, staromodne wymagania. Tak jak wcześniej, przy okazji będziemy trochę kwestionować konwencjonalną mądrość metodyki Agile. Ale nie damy jednoznacznej recepty.

Zanim przejdziemy dalej, powinniście zapamiętać dwie rzeczy. Po pierwsze, głównym celem każdego formalizmu związanego z wymaganiami jest zapewnienie centralnej platformy do bezpośredniego dialogu, który zachodzi wokół niej. Taki dialog daje największe szanse budowania wspólnej perspektywy niezbędnej do spełnienia oczekiwań użytkowników końcowych. Po drugie, trzeba pamiętać, że my nie udzielamy ostatecznych odpowiedzi. Zachęcamy do eksperymentowania w celu znalezienia tego, co sprawdza się w konkretnym przypadku. To może oznaczać stawianie sobie wyzwań oraz próbowanie rzeczy, w które nie do końca wierzymy, że działają. Być może warto, ale nikt z nas nie może wiedzieć z góry.

Spróbujmy zatem powrócić po kolei do odpowiedzi na pytania: *kto?*, *co?* i *dlaczego?* Naszym celem jest wyjście poza postulat Manifestu Agile „działającego oprogramowania". Chcemy, aby nasze oprogramowanie było *użyteczne*. Do podejścia metodyki Agile dodamy praktyki Lean, aby zminimalizować marnotrawstwo wynikające z przeróbek. *Czym-system-jest* to stabilna część

architektury, natomiast *co-system-robi* to jego część dynamiczna. Ale nawet w obrębie części *co-system-robi* jest element, który wcześnie należy ustabilizować — funkcja także ma formę — a bardziej dynamiczna część mieści się w tym fragmencie Manifestu Agile, który jest poświęcony „przygotowaniu się na zmiany". Potrzebujemy zarówno praktyk Lean, *jak i* metodyki Agile.

7.2. Kto będzie korzystać z naszego oprogramowania?

Początkowo w opowieściach użytkowników metodyki Agile rzadko występują prawdziwi użytkownicy (Jeffries, Anderson i Hendrickson 2001, s. 25 – 28). Na przykład:

> Dla każdego rachunku oblicz saldo poprzez zsumowanie wszystkich wpłat i odjęcie wszystkich wypłat. (Jeffries, Anderson i Hendrickson 2001).

W wielu fragmentach nazywanych *opowieściami użytkowników* w ogóle nie ma użytkowników. Wiele firm stosuje formę opowieści użytkownika „Jako użytkownik...", ale użytkownik nigdy nie jest nikim innym... jak tylko *użytkownikiem*. Podczas tej translacji zagubiono *rolę* użytkownika. Czy użytkownik jest w moim wieku? W wieku mojego dziecka? Mojej 85-letniej ciotki? Dobre historie użytkowników wyjaśniają role użytkowników. Role, które odzwierciedlają przemyślenia analityków dotyczące tożsamości użytkownika. W ten sposób do opowieści użytkowników trafia *użytkownik* w klasycznym znaczenia tego słowa. W opowieściach użytkowników Agile nie ma poza tym żadnej opowieści, ale to już inna opowieść...

Zanim przejdziemy do omawiania ról użytkowników, zwróćmy uwagę, że należy odróżnić *role użytkowników* od ról w architekturze DCI. W DCI są to obiekty, które mogą odgrywać różne role, więc by uniknąć nieporozumień, możemy nazwać je *rolami obiektów*.

7.2.1. Profile użytkowników

Jak definiujemy rolę użytkownika? Najpierw szukamy wskazówek w użyteczności. Eksperci w dziedzinie użyteczności wiedzą, jak badać perspektywę użytkownika końcowego oraz jak skonsolidować te ustalenia w *profilach użytkowników*. Model przypadków użycia reprezentuje interesy użytkowników końcowych za pomocą aktorów. Aktor jest rolą użytkownika, dlatego *profil użytkownika* można zmapować bezpośrednio do przypadku użycia *aktor*. Profil użytkownika staje się opisem aktora. Profile użytkowników mogą oczywiście być również używane jako role użytkowników w opowieściach użytkowników.

Jeśli organizacja nie posiada specjalistów w dziedzinie użyteczności, zespół może samodzielnie przeprowadzić cenne modelowanie ról użytkowników. Mike Cohn (Cohn 2004, rozdział 3.) przedstawia ciekawe wytyczne w formie procedury składającej się z czterech kroków.

7.2.2. Osoby

W przypadku aplikacji internetowych popularne stało się tworzenie *osób*. Osoba jest wymyśloną postacią o określonej płci, wieku, rodzinie, zawodzie, hobby oraz wszystkim innym, co można wiedzieć o postaci. Osoba jest „kimś, kogo znasz tak dobrze, jakbyś z nim spał" — to popularne powiedzenie w środowisku ludzi posługujących się tym pojęciem. Osoba może odgrywać wiele ról użytkowników w powiązaniu z naszym oprogramowaniem — tak jak prawdziwa osoba. Profil użytkownika nie jest konkretną osobą, ale rolą użytkownika bazującą na faktach dotyczących

rzeczywistych użytkowników. Osoby mają imiona, na przykład „Maria" lub „Jan", podczas gdy profil użytkownika charakteryzuje się nazwą roli użytkownika, na przykład „administrator bazy danych" lub „nastoletni klient banku".

7.2.3. Profile użytkowników czy osoby?

Osoby mogą dodać wartość do organizacji, w których oprogramowanie jest używane przez takie szerokie grono użytkowników, że określenie ograniczonego, cennego zbioru profili użytkowników wydaje się niemożliwe, lub tam, gdzie koncentracja na użytkownikach jest dla organizacji czymś nowym. Zamiast sytuacji, w której wszyscy interesariusze w czasie tworzenia oprogramowania mają na myśli własny obraz „osoby", mogą teraz współdzielić uspołeczniony, konkretny opis osoby. Istnieje ryzyko, że stworzymy oprogramowanie dla tej konkretnej osoby, a nie dla roli użytkownika, którą może odgrywać wiele różnych osób.

W bardziej dojrzałych organizacjach, w których członkowie zespołu mają zdyscyplinowany kontakt z użytkownikami końcowymi, osoby sprawiają wrażenie nierealistycznych. Będą one postrzegane jako powierzchowne i zbyt różniące się od prawdziwych użytkowników, z którymi mają styczność członkowie zespołu. W tego rodzaju organizacjach lepiej posługiwać się *profilami użytkowników* niż osobami. Profile użytkowników są bazującymi na doświadczeniu charakteryzacjami segmentów rynku, a w szczególności tych segmentów, do których ma dotrzeć produkt (rysunek 7.2).

Profil użytkownika: nastoletni użytkownik

Wiek: od 13 do 17 lat

Region geograficzny: Stany Zjednoczone i Australia

Wykorzystanie internetu: zadania szkolne, hobby, rozrywka, wiadomości, kwestie zdrowotne, które są dla niego zbyt wstydliwe, aby o nich rozmawiać, e-handel

Współczynnik wypełnienia: 85% wykorzystania witryny. Typowy czas w internecie wynosi 5–10 godzin tygodniowo

Umiejętności badawcze: przeciętne do słabych

Umiejętności czytania: przeciętne do słabych

Cierpliwość: niska, krótki czas koncentracji

Względy projektowe: przyciąga uwagę dobra grafika oraz dobry, czytelny projekt strony

Rysunek 7.2. Profil użytkownika (za Nielsen 2005)

Czasami wizerunek osoby jest zbyt ograniczający, a rynek zbyt odległy. Możemy skorzystać z wzorca projektowego *Surrogate Customer* (Coplien i Harrison 2004, s. 116 – 117), aby uzyskać jak najszerszą wiedzę o rynku w przypadkach, gdy mamy ograniczony kontakt z prawdziwymi klientami. Constantine i Lockwood (Constantine i Lockwood 1999) mówią również o tym, jak opracować rozsądne wymagania w takich warunkach. Należy jednak dążyć do zachowania postawy Agile i ciągłego uzyskiwania informacji od społeczności użytkowników końcowych.

7.2.4. Role użytkowników i terminologia

Słowa oznaczają rzeczy. Dobre, opisowe nazwy mogą pomóc w zbliżeniu się do oczekiwań użytkowników końcowych. „Maria" i „Jan" jako imiona typowych klientów nie wnoszą zbyt wielu informacji. „Nastoletni użytkownik" (rysunek 7.2) przekazuje znacznie więcej informacji i może być podstawą do stworzenia roli „nastoletni klient banku". Takie terminy są ważne nie tylko w rozmowach między zainteresowanymi stronami, ale także w zagwarantowaniu czytelności i łatwości utrzymania kodu. Tak, *można* używać tych uzgodnionych nazw w kodzie! Tradycyjne

możliwości śledzenia, które są sformalizowane przez narzędzia, są martwe w metodyce Agile (pamiętajmy o zasadzie „osoby i interakcje ponad procesy i narzędzia"). W podejściu Lean możliwości śledzenia zastępujemy uważnie dobraną terminologią. Dobra nazwa roli użytkownika może przyczynić się do właściwego rozumienia roli. Wystarczy obserwować swoich użytkowników — nie trzeba z nimi spać.

7.3. Do czego użytkownicy chcą wykorzystać nasze oprogramowanie?

To jest zasadnicza część odpowiedzi na pytanie, *co system robi*. Części *kto* i *dlaczego* pomagają właściwie odpowiedzieć na to pytanie — dostarczają ważnego *kontekstu*[1]. Odpowiedzi na pytanie, *co system robi*, można udzielić na wiele sposobów: w postaci wymagań, diagramów przepływu danych, własności, aktorów i przypadków użycia, opowieści użytkowników i testów akceptacyjnych, osób i scenariuszy, w postaci narracji, scenorysów (ang. *story boards*), prototypów i prawdopodobnie wielu innych. Kiedy system coś *robi*, oznacza to, że ma jakieś *zachowanie*. Powyższe techniki opisują zachowanie mniej lub bardziej wyraźnie. Tradycyjne sformułowanie wymagania może brzmieć następująco: „System powinien przelewać pieniądze z jednego rachunku na inny". Brzmi znajomo? Nie różni się zbytnio od naszej opowieści użytkownika: „Jako posiadacz rachunku chcę przelewać pieniądze pomiędzy moimi rachunkami, by mieć pewność, że na żadnym z rachunków nie powstanie debet".

7.3.1. Lista własności

Listy własności (ang. *feature lists*) to jeszcze inne podejście do opisywania funkcjonalności. Opis własności dla wymagania wymienionego powyżej może brzmieć następująco: „Przelew pieniędzy z jednego rachunku na inny". Słowo „przelew" oznacza zachowanie, ale nie jest ono wyraźnie opisane, co prowadzi do stawiania pytań, których przykłady zaprezentowaliśmy w pierwszej części tego rozdziału.

Różnica pomiędzy własnością a opowieścią użytkownika jest subtelna. Na początku, kiedy Kent Beck wprowadził opowieści użytkowników (Beck 2005), jedyną różnicą pomiędzy opisem własności a opowieściami użytkownika, było to, że opowieści użytkowników były krótkie i pisane ręcznie na „karcie opowieści" (ang. *story card*). W dzisiejszym świecie Agile większość opowieści użytkownika jest zapisywanych w komputerze za pomocą takich narzędzi jak Excel lub narzędzi bardziej dostosowanych do tego celu. Trudno jest zauważyć różnicę między listami własności a listą opowieści użytkowników z wyjątkiem tego, że termin *opowieść użytkownika* jest społecznie bardziej akceptowalny wśród firm programistycznych, które postrzegają siebie jako Agile.

7.3.2. Diagramy przepływu danych

Diagram przepływu danych (ang. *Data Flow Diagram* — DFD) — kolejna alternatywa — zawierałby wysokopoziomowy proces o nazwie „Przelew pieniędzy". Do i z tego procesu przekazywane byłyby dane (tzn. kwota i saldo). Proces ten można zdekomponować na następujące podprocesy:

[1] Tutaj, w rozdziale 7. używamy słowa „kontekst" w przybliżeniu w takim znaczeniu, w jakim używa się go w języku naturalnym. Pojęcie w znaczeniu bardziej ścisłym pojawia się w rozdziale 9.

„Zdefiniuj rachunek źródłowy i docelowy", „Przelej pieniądze" oraz „Zrealizuj zapisy księgowe". Dekompozycja zatrzymuje się, gdy proces najniższego poziomu można nazwać „atomowym" (nie można go dalej dekomponować). Diagramy przepływu danych mogą się sprawdzać w przypadku indywidualnego analityka, ale nie gwarantują skutecznego mechanizmu przekazywania wiedzy pomiędzy członkami zespołu projektowego. Bardzo łatwo pogubić się na wielu poziomach dekompozycji.

7.3.3. Osoby i scenariusze

Osoby często są wykorzystywane razem ze scenariuszami. Scenariusz opisuje konkretny schemat działania dla wskazanej osoby. Jeśli osobą jest „Maria", 32-letnia samotna matka 2-letniego Benjamina i 4-letniej Laury, która pracuje na pełny etat jako pielęgniarka, to scenariusz może opisywać, jak Maria w drodze do domu z pracy korzysta z bankomatu, aby przelać pieniądze z konta oszczędnościowego, by mogła zapłacić rachunek za naprawę samochodu. Scenariusz dla osoby koncentruje się na tym, jakie przyciski ma wciskać Maria oraz jakie elementy menu ma wybierać.

7.3.4. Narracje

Narracja[2] może korzystać z tego samego podejścia jak w przypadku osób, ale unika tworzenia konkretnego stereotypu. Zamiast tego tworzone jest krótkie opowiadanie opisujące funkcjonalność systemu. Na przykład możemy wymyślić hipotetyczną osobę o imieniu Maria „w locie". Niech Maria będzie treserem psów policyjnych. Jej samochód jest w warsztacie. Maria trzyma w jednej ręce smycz z dwoma psami policyjnymi, a na drugim ręku ma Benjamina. Pada deszcz, a za 10 minut zamykają przedszkole Laury. W tym przypadku narracja opisywałaby wizję, jak bankomat może pomóc Marii w tej konkretnej sytuacji, bez skupiania się na szczegółach interfejsu użytkownika bankomatu. Ale historia może nam pomóc w uzyskaniu świadomości, że ten przypadek wiąże się z innymi wymaganiami w odniesieniu do interfejsu użytkownika niż w przypadku osoby, która wykonuje transakcje bankowe z komputera w swoim domu. Narracja koncentruje się na *kontekście* (Cockburn 2001, s. 18).

Narracja może się składać z dwóch części: części *przed* i części *po*. Opisują one kłopoty Marii, zanim bank dał jej łatwy sposób przelania pieniędzy, oraz po tym fakcie. Możemy zatem mówić o *narracji problemu* oraz *narracji wizji*, które pomagają zrozumieć *motywację* użytkownika lub firmy. Wykorzystanie kombinacji osób i scenariuszy wyjaśnia motywację użytkownika, ale opisuje interfejs użytkownika w pewien pośredni, zagmatwany sposób.

7.3.5. Projektowanie aplikacji sterowane zachowaniami

W opowieściach użytkowników brakuje wyraźnego opisu zachowania. Popularnym sposobem na to, by ten brak zrekompensować, jest dodanie testów akceptacyjnych na odwrocie karty opowieści użytkownika. Oznacza to, że odpowiedzi na niektóre z pytań postawionych na początku tego

[2] Używamy terminu „narracja" po to, by nie mylić tego rodzaju historii z opowieściami użytkowników. Alistair Cockburn używa terminu „narracje użycia" w odniesieniu do historii, które przypominają scenariusze w podejściu wykorzystującym osoby i scenariusze. Unika terminu „opowieść" w tym kontekście z tego samego powodu: aby nie mylić ich z pojęciem opowieści użytkowników w programowaniu ekstremalnym.

rozdziału zostaną udzielone w postaci kryteriów testów akceptacyjnych. Na przykład „Test: jeśli konto źródłowe po transakcji stanie się ujemne, przelew nie będzie dozwolony". Prawdopodobnie będzie więcej kryteriów testów akceptacyjnych. Tyle, na ile pozwoli miejsce na karcie. Kryteria te zostaną opisane w innym miejscu.

To skrócone podejście sugeruje, że przemieszczamy wiedzę z obszaru *co-system-robi* do opisów testów. Właśnie takie podejście jest stosowane w metodyce BDD (*Behavior-Driven Development*) (North 2006). W BDD opowieści użytkowników są danymi wejściowymi do scenariuszy testowych opisujących, co system robi. Programista koduje scenariusze testowe bezpośrednio za pomocą narzędzia testowego — i voilà! Od tej chwili programista jest odpowiedzialny za wymagania. Być może jest to o jeden krok dalej, niż chcielibyśmy pójść teraz. Nie chcemy stracić naszych ekspertów z dziedziny biznesu ani specjalistów od interakcji z użytkownikami z powodu dążenia do kodowania. Kod jest bardzo czytelny dla koderów, ale być może nie jest tak bardzo czytelny dla reszty świata.

7.3.6. Teraz, gdy jesteśmy rozgrzani...

Spróbujmy cofnąć się o parę kroków i zobaczmy, w jakim miejscu jesteśmy. Mamy techniki do opisania *co-system-robi*, w których brakuje jawnej specyfikacji zachowania: tradycyjnych wymagań, list własności i opowieści użytkownika (wykorzystywanych bez testów akceptacyjnych). Mamy również mechanizm osób i scenariuszy, które opisują zachowania, ale które ograniczają się tylko do kilku specyficznych przykładów. Pozostałą część zachowania pozostawiono wyobraźni programisty. Narracje są po prostu opowieściami, które mogą nam dać dobry, wspólny obraz motywacji. Ale narracji nie można testować, zatem oferują one bardzo niepełny obraz tego, co system robi.

Prototypy

Być może powinniśmy spróbować zasymulować sytuacje, w których powstają te wymagania. Prototypy są świetnym sposobem na odkrywanie i testowanie części tego, co robi system. Prototyp można wykorzystać do przetestowania koncepcji interfejsu użytkownika, części funkcjonalności, a nawet wydajności lub technicznej wykonalności. Prototypy architektury również mogą być bardzo przydatne. Scenorysy to bardzo uniwersalna technika użyteczności. Pierwotnie używano ich do wspierania projektu, ale równie dobrze służyły sprawdzaniu poprawności założeń zespołu dotyczących przepływu sterowania dla poszczególnych funkcji systemu. Zespół zaczyna od scenorysów, których zadaniem jest uchwycenie wizji przepływu pracy użytkownika, aby później przystąpić do sprawdzania założeń w rzeczywistym środowisku.

W kierunku podstaw do podejmowania decyzji

Wszystko to są dobre techniki rozgrzewki: pozwalają rozpocząć konwersację, burzę mózgów i wizjonerstwo. Należy wybrać swoje ulubione i zacząć je stosować — aż będziemy gotowi do konsolidacji. Cel, jakim jest architektura, wymaga skonsolidowanych danych wejściowych. Konsolidacja daje pewność, że odpowiednie osoby podejmą właściwe decyzje we właściwym czasie. Burza mózgów i wizjonerstwo są zarówno pomocne, jak i konieczne, ale są to *dywergentne* techniki myślenia, podczas gdy decyzje powinny ostatecznie wyłaniać się z technik *konwergentnych*. Są to decyzje dotyczące zarówno biznesu, jak i architektury — te dwa obszary idą „ręka w rękę".

Znane i nieznane niewiadome

Zatem jaki jest nasz poziom konsolidacji, jeśli chcemy być zarówno Lean, jak i Agile? Nie chcemy robić wszystkiego „z góry". To nie byłoby Agile. Nie chcemy odkładać decyzji „do ostatniego odpowiedzialnego momentu" — to nie byłoby Lean. Więc przede wszystkim powinniśmy wiedzieć, jakie decyzje musimy podjąć, aby uzyskać właściwą architekturę, oraz jakie decyzje powinniśmy odroczyć. Dobrze byłoby mieć framework do zorganizowania tych decyzji. Pierwszym krokiem jest oddzielenie znanych wiadomych od znanych niewiadomych, tak aby znane niewiadome stały się jak najbardziej widoczne. Z uwagi na nowe wymagania nie możemy przewidzieć, kiedy niewiadome staną się wiadomymi, ale chcemy pozostawić miejsce dla tych odkryć.

Musimy też, z tyłu głowy, schować fakt, że gdzieś pojawią się nieznane niewiadome. Tworzenie prototypów może pomóc w ucieleśnieniu nieznanych niewidomych poprzez przekształcenie ich w znane niewiadome, a czasami przez całkowite wyeliminowanie niepewności w wyniku dialogu z interesariuszami — cechy typowej dla mentalności Lean „wszystkie ręce na pokład". Zdrowa architektura dziedziny może przyczynić się do obniżenia kosztów zmiany wymagań *co-system-robi* poprzez ograniczenie długoterminowych przeróbek ogólnej formy systemu. Zanim jednak zajmiemy się tymi niespodziankami, warto stworzyć framework, który organizuje to, co *wiemy* do tej pory.

Przypadki użycia jako framework podejmowania decyzji

Jednym z przykładów takiego frameworka są przypadki użycia. Przypadki użycia mają sens tylko wtedy, gdy oprogramowanie obsługuje przepływ pracy użytkownika. Jako kontrprzykład można przytoczyć prosty edytor tekstu — trudno stworzyć dla niego przypadki użycia, ponieważ nie ma w nim spójnych przepływów pracy. To samo odnosi się do systemów bez większych interakcji z użytkownikami, jak przemysłowe systemy sterowania czy też oprogramowanie zainstalowane w samochodzie lub suszarce do ubrań. Być może macie doświadczenie z technikami, które mogą pomóc w konsolidacji tego rodzaju systemów — może za pomocą maszyny stanów, tabel decyzyjnych lub wykresów przyczynowo-skutkowych. Więc które z nich można wykorzystać zamiast przypadków użycia? Odpowiedź brzmi: to zależy. Do tego zagadnienia powrócimy w podrozdziale 7.7. Na razie chcemy podzielić się naszymi doświadczeniami z przypadkami użycia jako technikami Lean konsolidacji tej części architektury, która opisuje, co robi system.

Przypadki użycia mają złą reputację. Mimo że mają opinię technik „ciężkich", przypadki użycia obsługują metodykę Agile bardzo dobrze. Są skutecznym sposobem opisywania specyfikacji stwarzających możliwości. Wystarczy wziąć pod uwagę aktualizacje przyrostowe zamiast specyfikacji „z góry" oraz *gry kooperacyjne* (Cockburn 1999) zamiast rozbudowanej analizy wymagań „z góry".

Zanim przejdziemy do omawiania konsolidacji zastanówmy się, dlaczego użytkownik chce korzystać z naszego oprogramowania.

7.4. Dlaczego użytkownicy chcą korzystać z naszego oprogramowania?

Zanim odpowiemy na to pytanie, dobrze jest wcześniej zastanowić się nad innym.

Dlaczego przedstawiciele biznesu uważają, że użytkownicy chcą korzystać z naszego oprogramowania?

Przedstawiciele biznesu mają do czynienia z ludźmi pochodzącymi z szerokich i różnorodnych kręgów. W technologii Scrum mamy właściciela produktu, który równoważy interesy wszystkich

zainteresowanych stron w oprogramowaniu. Użytkownicy końcowi są interesariuszami, ale dla wielu przedstawicieli biznesu to klient — ten, który płaci — jest często ważniejszy niż końcowy użytkownik. Jeszcze ważniejsze od klienta może być samo przedsiębiorstwo — jak nasza firma będzie się rozwijać w długim okresie? Pierwszy krok polega na rozpoznaniu tych różnic. To niesamowite, jak wielu firmom brakuje wyraźnego obrazu różnicy pomiędzy klientem a użytkownikiem końcowym (ponadto niezwykłe jest, jak wiele firm jest całkowicie pozbawionych kontaktu z użytkownikami końcowymi).

Narracje mogą być dobrym sposobem na zbadanie interesów tych, o których dbamy. W powyższym przykładzie z Marią, samotną matką, która chce przesłać pieniądze, najpierw powinniśmy skupić się na tym, by jej pomóc. Ale być może ona wolałaby, aby bank był otwarty poza jej godzinami pracy, tak aby mogła wejść i porozmawiać z prawdziwą osobą, która zrealizowałaby za nią przelew (oraz skorzystać z pomocy innej osoby, która zajęłaby się dzieckiem i psami) — oczywiście bez potrzeby czekania w kolejce. Tak więc motywacją dla banku być może jest nie tyle pomoc Marii, ile uniknięcie wydłużenia godzin otwarcia czy też zatrudnienia większej liczby pracowników do obsługi klientów. Bank chce świadczyć usługi na rzecz swoich klientów w sposób, który oszczędza ich pieniądze i pozwala im wykonywać pracę. Bank chce działać w taki sposób, aby nie tracić klientów na rzecz konkurencji. Aby zmotywować klientów banku do lojalności wobec niego, dobrym pomysłem jest patrzenie na nich w kontekście, jak na użytkowników końcowych. Kiedy zamawiamy bilety lotnicze, wybieramy dostawcę, który może zarówno zaoferować tanie bilety, jak i dostarczyć łatwą w obsłudze stronę internetową.

W przypadku frameworka Scrum dobry właściciel produktu przekazuje członkom zespołu motywację różnych interesariuszy. Jednym ze sposobów, aby to zrobić, jest wypowiedzenie lub napisanie narracji. Innym jest poruszająca rozmowa na temat nowego produktu lub usługi. Jeden z właścicieli produktów powiedział nam, że po 20 latach eksperymentowania doszedł do przekonania, że rozmowa jest najskuteczniejszym sposobem na przekazanie motywacji interesariuszy.

Przypadki użycia mogą być zwięzłym, zorganizowanym sposobem komunikowania tych motywacji. Każdy przypadek użycia opisuje we wprowadzeniu motywację biznesową oraz motywację lub zamiary użytkowników. Stara, dobra analiza interesariuszy także się sprawdza. Wiele nieporozumień i brak motywacji w zespołach deweloperskich wynika z tego, że zespół nie wie, *dlaczego* ma robić to, co robi. Członkom zespołu dużą trudność sprawia także podejmowanie świadomych decyzji. I nie da się wszystkiego napisać. To jest jeden z powodów, dla którego metodyka Agile stała się tak popularna. Trzeba pozwolić ludziom (i kodowi) mówić samym za siebie.

7.5. Konsolidacja tego, co system robi

Jakie zatem są kryteria konsolidacji tego, *co system robi*? Kluczowe znaczenie ma użyteczne oprogramowanie, a żeby je zbudować, musimy znać naszych użytkowników. Zbieramy dane o naszych użytkownikach i tworzymy ich profile — lub opisy ról, aktorów czy osób — w zależności od organizacji i kultury. Wszystko to mówi nam, *kim* jest użytkownik. Ale chcemy też wiedzieć, co użytkownik *robi* — interesują nas jego zachowania. Scenorysy są sposobem na walidację naszych założeń na temat zachowania użytkownika. Scenariusze w technice „osoby i scenariusze" dają nam przykład, jak wyobrażamy sobie korzystanie z naszego systemu przez użytkowników. Narracje dostarczają wizji, dlaczego chcemy określonej funkcjonalności. Diagramy przepływu danych pomagają nam zrozumieć... przepływ danych.

Częścią, na której chcemy się skoncentrować w związku z zachowaniem użytkownika, jest jego *przepływ pracy* oraz to, jak chcemy wspierać ten przepływ pracy za pomocą naszego oprogramowania: co-system-robi. W celu przygotowania się na zmiany musimy rozwijać nasze oprogramowanie

w małych kawałkach — w postaci iteracji lub sprintów. Stopniowo budujemy nasze oprogramowanie w sposób, który pozwala nam dostosować się i zareagować na zmiany pomiędzy iteracjami. Kiedy reagujemy na zmiany, chcemy wiedzieć, gdzie jesteśmy. W przeciwnym razie tylko *reagujemy*, a nie *odpowiadamy*. Potrzebujemy bazy. Baza jest zakodowanym i przetestowanym oprogramowaniem, które jest gotowe do wydania wewnętrznego lub zewnętrznego. Kluczowe jest tutaj słowo *test*. Zespoły powinny zbadać przepływ pracy użytkownika, który chcą wspierać w każdej iteracji. Chcemy, aby testy były tak blisko prac rozwojowych, jak to możliwe.

Pojawiają się nowe wymagania — zarówno zewnętrzne, jak i wewnętrzne. Świat i biznes zmieniają się. A czasami zmiany wymagań wynikają z tego, że ktoś po prostu zmienił zdanie. Uczymy się więcej, gdy pracujemy z dziedziną, a system oraz wymagania zmieniają się nieco bardziej. Pracujemy w sposób przyrostowy i odnajdujemy nowe „zakamarki" systemu. Wszystko to oznacza, że decyzje o funkcjonalności podejmujemy cały czas. Aby uniknąć omawiania tych samych kwestii w kółko, musimy udokumentować nasze decyzje na temat tego, co robi system. Możemy oczywiście zmienić nasze decyzje, ale wtedy wiemy, z jakiej decyzji do jakiej nastąpiły zmiany.

W ostatecznym efekcie wiele decyzji będzie udokumentowanych w kodzie. To jest OK. Precyzyjniej byłoby powiedzieć, że te decyzje będą *zakodowane* w kodzie, a zdekodowanie ich często nie jest zabawne. Jednym z celów technik zaprezentowanych w tej książce — w szczególności projektowania opartego na dziedzinie i architekturze DCI — jest wyeliminowanie kodu jako sposobu komunikowania zamiarów programisty. Ważne jest, aby posługiwać się wspólnym językiem dokumentowania decyzji. Ten język rzadko może być językiem programowania. W końcu nie dla wszystkich interesariuszy język programowania jest podstawowym językiem. Spotkania i sprawozdania z tych spotkań również nie zawsze się sprawdzają (w końcu kto to czyta?). Zatem potrzebujemy frameworka, który pomoże nam opisać ważne decyzje dotyczące funkcjonalności.

Na razie nie mówimy o decyzjach, które może podejmować programista. Te są najlepiej dokumentowane za pomocą kodu. Mówimy o decyzjach, które dotyczą innych zainteresowanych — szczególnie przedstawicieli biznesu, na przykład właściciela produktu w metodyce Scrum.

Nowe wymagania również stwarzają potrzebę podejmowania decyzji. Dla nich również potrzebujemy miejsca. W tradycyjnej specyfikacji wymagań w rzeczywistości nie ma miejsca na nowe wymagania — zbiór wymagań jest zamknięty. Opowieści użytkowników i zestawy funkcji w gruncie rzeczy mogą się tylko rozrastać wraz z każdym nowym wymaganiem. Strategia ta jednak powoduje, że nie jesteśmy w stanie zobaczyć lasu z powodu drzew. Kiedy opowieść użytkownika się powtarza? Kiedy jest to tylko zbieżność z inną opowieścią użytkownika? Jak można zweryfikować, czego brakuje? To podejście sprawdza się w przypadku bardzo małych i prostych systemów — na przykład stron internetowych, kiedy można wydać nową wersję codziennie i natychmiast uzyskać informacje zwrotne. Nasze decyzje są udokumentowane na stronie internetowej, a użytkownicy je testują. To tworzy zwięzły, prosty kontekst.

Ostatnim elementem w części *co-system-robi* jest zwrócenie uwagi na to, aby fragmenty funkcjonalności były wystarczająco małe, tak by planowanie wydania było łatwe. Co planowanie wydania ma wspólnego z architekturą? Chcemy wiedzieć, które funkcjonalności będą wyzwaniem dla architektury, a które są tylko rozszerzeniem już istniejących funkcjonalności. Jedną z opcji, jakie mamy do dyspozycji, jest zajęcie się funkcjonalnością ryzykowną dla architektury we wczesnej iteracji, aby uniknąć przykrych niespodzianek w późniejszym iteracjach. Jest to jeden z przykładów sytuacji, gdy architekt powinien wspomóc właściciela produktu w szeregowaniu rejestru wymagań produktu. To architekt jest tym, który wie, czy konieczna jest redukcja ryzyka. Jest to sprzeczne z zasadą naiwnych zwolenników Agile, że najpierw należy zająć się najprostszymi rzeczami. To jednak stosunkowo niewielki problem, jeśli wziąć pod uwagę pogląd Agile na temat czegokolwiek, co jest ukierunkowane na architekturę.

Dlatego kryteria konsolidacji części *co-system-robi* zawierają następujące elementy:

1. Wsparcie przepływu pracy użytkowników (scenariusze użycia).
2. Wsparcie dla testów, które powinny być blisko prac rozwojowych (scenariusze testów).
3. Wsparcie dla skutecznego procesu podejmowania decyzji o funkcjonalnościach (scenariusz słonecznego dnia — znany również jako *scenariusz najważniejszego sukcesu* — a odchylenia).
4. Wsparcie dla nowo powstających wymagań (odchylenia).
5. Wsparcie dla planowania wydań (przypadki użycia i odchylenia).
6. Uzyskanie danych wejściowych do opracowania architektury (terminy i pojęcia ze scenariuszy).
7. Budowanie w zespole zrozumienia tego, nad czym zespół pracuje.

Pokazaliśmy wcześniej, że jednym z rozwiązań są przypadki użycia. Nie są one jednak jedynym rozwiązaniem. Warto wziąć pod uwagę kryteria wymienione powyżej. Czy pasują do naszego kontekstu rozwoju? Jakie inne techniki z powodzeniem stosowaliśmy wcześniej? Należy kierować się zdrowym rozsądkiem!

Mogliśmy wybrać dla tej techniki jakąś inną nazwę niż przypadek użycia — być może zastosować jakieś fantazyjne japońskie słowo i powiedzieć, że jest to nowa technika formułowania wymagań zgodna z Lean. Prawdopodobnie zajęłoby nam to trochę czasu, zanim rozpoznalibyśmy tę technikę jako przypadek użycia. Jesteśmy jednak zwolennikami przejrzystości. Zilustrujemy to za pomocą przykładu. Naszym celem nie jest nauczenie Was posługiwania się przypadkami użycia. (Literatura dostarcza mnóstwo dobrych źródeł: Adolph et al. 1998; Constantine i Lucy 1999; Cockburn 2001; Wirfs-Brock 1993). Naszym celem jest pokazanie, w jaki sposób technika przypadków użycia może pomóc w rozwijaniu oprogramowania zgodnie z Agile i Lean, bez konieczności odkrywania nowych technik tylko po to, by móc je nazwać *Agile* i *Lean*. Nie stawiamy sobie celu, aby pokazać doskonały przykład zastosowania przypadków użycia. Proces wokół przypadków użycia jest ważniejszy niż same przypadki użycia.

7.5.1. Widok helikoptera

Po rozgrzewce jesteśmy gotowi do konsolidacji. Być może mamy wiele historii użytkowników oraz kilka prototypów. Przedstawiciel biznesu wygłosił motywacyjną mowę lub cały zespół podzielił się żywymi opowieściami. Mamy mnóstwo danych wejściowych — przede wszystkim chcemy je zorganizować i dokonać ich przeglądu. Czego nauczyła nas sesja rozgrzewkowa na temat kontekstu?

Przejdźmy od bankomatu znajdującego się na zewnątrz i wejdźmy do środka, gdzie nie pada. Spróbujmy pomyśleć o sobie jako o obywatelach, którzy chcą przeprowadzać transakcje bankowe, siedząc w domach przy swoich komputerach. Bank nie jest już diabelskim twórcą pieniędzy, ale naszym partnerem, który próbuje ułatwić nam życie. Jakich zatem najważniejszych usług oczekiwalibyśmy od banku? Co chcielibyśmy realizować w domu przez internet? Chcielibyśmy móc przelewać pieniądze, przeglądać ostatnie transakcje, dodawać nowe stałe płatności, drukować wyciągi z konta i opłacać rachunki. Podczas burzy mózgów może przyjść nam na myśl wiele innych rzeczy, ale to są podstawowe usługi, które menedżer produktu uważa za ważne na początek. Moglibyśmy nazwać je „usługami" lub „epiką" lub „ogólnymi wymaganiami" albo „funkcjami", ale wolimy nazywać je przypadkami użycia.

Abyśmy mogli używać nazwy *przypadki użycia*, każdy z nich powinien wspierać cel dla użytkownika. Najpierw zatem powinniśmy zdefiniować użytkownika: w języku przypadków użycia to jest *aktor*. W trakcie sesji burzy mózgów zbadaliśmy granice: czy osoba w wieku 11 lat jest zbyt młoda? Czy osoba w wieku 85 lat jest zbyt stara? Czy mamy na myśli tylko osoby prywatne, czy również instytucje? Czy udzielamy dostępu klientowi, który ma kredyt, ale nie ma rachunku w naszym banku? (Czy takie osoby istnieją?) Zdecydowaliśmy się nazwać aktora „prywatnym posiadaczem rachunku". Jeśli nie mamy dobrego opisu tego aktora, priorytetem powinno być jego opracowanie. Dysponujemy teraz prostym przeglądem.

Prywatny posiadacz rachunku
- Przeglądanie ostatnich transakcji
- Przelew pieniędzy
- Drukowanie wyciągu z rachunku
- Dodawanie stałych płatności
- Opłacanie rachunków

Chcemy wykonać pierwszą konsolidację widoku helikoptera. Będzie to kontener, w którym znajdzie się większość naszych decyzji, *co system robi*, dlatego chcemy jak najszybciej go ustabilizować. Jedną z rzeczy potrzebnych na początku projektu jest nazwa systemu. Nazwijmy go Internetowy Bank Zielony Ląd. Wraz z opisem aktora i najważniejszymi przypadkami użycia powinno dać nam to dobry, ogólny obraz kontekstu systemu.

Podczas przeglądania listy przypadków użycia, a może podczas myślenia o tym, jak powinien się zacząć pierwszy przypadek użycia, zauważamy że potrzebujemy usługi logowania. To jest klasyczna dyskusja dotycząca obszaru przypadków użycia: Czy „Logowanie" jest przypadkiem użycia? W większości przypadków odpowiedź brzmi: nie. Wcześniej zauważyliśmy, że przypadek użycia ma cel w kontekście. Użytkownik nie uzyskuje niczego przez samo zalogowanie. Nie byłby zadowolony z siebie, gdyby opowiadając rodzinie o tym, jak spędził dzień, powiedział: „Dzisiaj naprawdę coś zrobiłem — zalogowałem się do mojego banku dziesięć razy".

Z jednej strony logowanie jest złem koniecznym, niezbędnym warunkiem wykonywania czegoś innego. Z drugiej — jest ono ważne dla transakcji bankowych. Bezpieczeństwo dostępu leży nie tylko w interesie banku, ale także w interesie posiadacza rachunku. Procedura logowania w bankach może obejmować cztery lub pięć kroków, zatem system i aktor naprawdę mają coś do zrobienia. A nie chcemy przecież powtarzać tych czynności na początku każdego przypadku użycia. Czy możemy odroczyć decyzję dotyczącą tego, jak to powinno działać? Możemy poczekać i podjąć decyzję później. W takim przypadku możemy pominąć ten temat w naszej dyskusji i przejść dalej. Ale po co odkładać decyzję? Prawdopodobnie nie uzyskamy więcej informacji, które mogłyby nam pomóc. Jeśli dodajemy usługi do istniejącego systemu, w którym funkcjonalność logowania była dostępna już wcześniej, nie musimy robić nic więcej. W takim przypadku wystarczy, jeśli stwierdzimy: „posiadacz rachunku jest zalogowany" jako warunek wstępny we wszystkich przypadkach użycia. Jeśli musimy zmienić istniejące procedury logowania, dodać nowy sposób logowania lub jeszcze nie mamy tej funkcjonalności, to trzeba coś zrobić. To jest ważne wymaganie. Należy kierować się zdrowym rozsądkiem.

Mogą się zdarzyć sytuacje, w których zdrowy rozsądek nakaże nam zrealizować usługę logowania jako odrębny przypadek użycia. To jest niezgodne z dobrymi praktykami przypadków użycia, ponieważ operacji logowania brakuje celu. Bezpieczeństwo musi być zasadniczą kompetencją biznesową Internetowego Banku Zielony Ląd. W celu zapewnienia odpowiedniego poziomu bezpieczeństwa systemu musimy stworzyć pomost pomiędzy aspektami biznesowymi a aspektami programistycznymi. Komunikowanie tych aspektów za pomocą przypadków użycia może

okazać się pomocne. To, czy logowanie zostanie zaprezentowane jako przypadek użycia, czy nie, nie jest takie ważne. Ważne jest to, że temat ten został omówiony i uzyskaliśmy konsensus w sprawie decyzji, więc wszyscy wiemy, *dlaczego* mamy przypadek użycia logowania lub dlaczego go nie mamy.

Przyzwyczajenia: punkt widzenia programisty a punkt widzenia użytkownika

Deweloperzy starają się unikać redundancji, ponieważ nadmiarowy kod sprawia trudności w utrzymaniu. W przypadku analizy sprawa przedstawia się inaczej. Przypadki użycia koncentrują się na *kontekście*. Nie ma problemu, jeśli powtórzymy wyjaśniającą funkcjonalność w wielu przypadkach użycia. Kiedy powtarzanie sprawia, że zbyt trudno utrzymać spójność aktualizacji (ponieważ musimy edytować tę samą treść wiele razy), to musimy coś zrobić. Ale nie usuwamy redundancji, jeśli nie jest to konieczne. Eliminowanie redundancji jest zadaniem projektu, a projekt lepiej wyraża się za pomocą innych narzędzi niż przypadki użycia.

Wszyscy pracujemy razem, biorąc pod uwagę perspektywę biznesową z jednej strony oraz interesy programisty z drugiej. Programiści i przedstawiciele biznesu patrzą na przypadki użycia inaczej. Kiedy urzędnik bankowy przegląda przypadek użycia, może chcieć, aby wszystkie kroki logowania były tam uwzględnione. Ważne jest, aby użytkownik uwierzytelnił się za pomocą hasła. Ważne jest, aby zobaczyć, że w tym przepływie użytkownik może uzyskać podpowiedź hasła. Ważne jest, aby programiście zostały przedstawione reguły biznesowe dotyczące zapewnienia odpowiednio trudnego hasła. Nie ma znaczenia, czy te kroki są w spójny sposób zdublowane w innym miejscu: bezpieczeństwo niesie ze sobą powtórzenia. Poza tym byłoby trudne, a nawet mylące rozwijanie przypadku użycia uwierzytelniania w izolacji, ponieważ dobry przypadek użycia zawsze rozgrywa się w kontekście jego celu. Mówienie o logowaniu w obliczu braku jakiegokolwiek celu sprawia, że czynnościom biznesowym brakuje kontekstu.

Z drugiej strony takie powtórzenie, w najgorszym wypadku, doprowadzi do tego, że programista będzie niepotrzebnie powielać kod. Informatyka poświęciła wiele lat na opracowanie formalizmów zwanych procedurami tylko na tę okazję, aby pojedyncza, zamknięta kopia kodu mogła być ponownie wykorzystana w wielu punktach wywołania. Ale teraz na programiście spoczywa ciężar porównania każdej nowej sekwencji logowania ze starymi — linia po linii — aby zapewnić, że ten sam zamknięty egzemplarz procedury logowania obowiązuje w nowej odmianie w taki sam sposób, w jaki był stosowany, gdy kod został napisany. Konteksty muszą być ze sobą zgodne, a pytania o kontekst, gdy się pojawią, wymagają subtelnych wyjaśnień od przedstawicieli biznesu. To jest marnotrawstwo czasu.

Przydałoby się zastosować tu kompromis i traktować uwierzytelnianie jako element podobny do przypadku użycia, ale takiego, który przedstawia czysty punkt widzenia programisty, a nie przedstawicieli biznesu. Taki pseudoprzypadek użycia nazywamy *zwyczajem* (ang. *habit*). Nazwy tej użyliśmy dla odróżnienia od przypadku użycia. Jest to przeznaczone dla programistów narzędzie, które wspiera projekt i które istnieje głównie w ich obszarze. Zwyczaje są strukturami, które pomagają zespołowi zadawać właściwe pytania, aby się upewnić, że wszyscy tak samo rozumieją określone elementy — w ten sposób spełniamy cel Lean spójności. Należy pamiętać, że przypadki użycia (a także zwyczaje) *reprezentują* wymagania, ale odpowiedź na pytanie, czy rzeczywiście *są* one wymaganiami, musi poczekać do opinii użytkownika w działającym systemie.

Zwyczaje pomagają rozwiązywać długotrwałe problemy, dla których przypadki użycia są często zbyt szczegółowe dla przedstawicieli biznesu i niewystarczająco szczegółowe dla programistów. Z historycznego punktu widzenia doprowadziło to przedstawicieli biznesu do posługiwania się opowieściami użytkowników, a programistów — do wykorzystania skryptów testowych. Poprawki te, zamiast rozwiązać problem, tylko pogłębiły przepaść. Zwyczaje eliminują powta-

rzające się fragmenty szczegółów przypadków użycia. W ten sposób uzyskujemy bardziej spójne przypadki użycia dla przedstawicieli biznesu. Programiści mogą skorzystać z tych wielokrotnie wywoływanych fragmentów w celu uzyskania wystarczającej wiedzy o scenariuszach. Dzięki temu mogą opracować algorytmy i je zaimplementować (punkt 9.4.2). Algorytmy nie wymagają pośredniej reprezentacji: kod programu dobrze się do tego nadaje. Przedstawiciele biznesu mogą dalej pracować na wyższym poziomie.

Zwyczaj nie jest tym samym co przypadek użycia, ponieważ ten pierwszy *uwzględnia* relacje. Nie chcemy bowiem spowodować, by przypadek użycia stał się niezrozumiały. Głównym założeniem zwyczajów nie jest wyeliminowanie informacji z przypadków użycia w celu usunięcia redundancji. Zamiast tego dodają szczegóły w taki sam sposób jak reguły biznesowe. Powszechnie stosowaną praktyką jest oddzielenie reguł biznesowych i innych szczegółów uzupełniających z opisów przypadków użycia. Przypadek użycia jest „izbą rozliczeniową" informacji, która wskazuje inne potrzebne szczegóły, a zwyczaj jest jednym z tych, których potrzebujecie; zazwyczaj jest jednym z tych szczegółów. Zwyczaje różnią się od przydomków (punkt 5.4.1), ponieważ koncentrują się bardziej na działaniu niż na danych. Zwyczaje nie są algorytmami, ponieważ w pewnym stopniu są niedeterministyczne — wyszczególniają decyzje, które nie mają znaczenia i jako takie są pozostawione programiście lub technologii (podobna zasada dotyczy przypadków użycia). Zwyczaje są zwykle częściowo uporządkowaną listą etapów. Mogą reprezentować reguły biznesowe, algorytmy lub kroki w przypadkach użycia.

Na rysunku 7.3 pokazaliśmy scenariusze dla operacji Przesuń pieniądze i Zaksięguj, wykorzystując zmodyfikowaną formę przypadku użycia, którą nazwaliśmy „zwyczajem". Nazwa „Przesuń pieniądze" odzwierciedla pewien poziom szczegółów. Porównajmy ją z nazwą „Przelej pieniądze", która implikuje transakcję biznesową, podczas gdy operacja „Przesuń pieniądze" jest bardziej mechaniczna. W zwyczaju brakuje zamiaru i motywacji. Dane te możemy uzyskać z każdego okalającego zwyczaj przypadku użycia. (Zamiar i motywacja mogą być różne dla różnych przypadków użycia wywołujących operacje Przesuń pieniądze i Zaksięguj!)

Nazwa zwyczaju: Przesuń pieniądze i Zaksięguj

Warunki wstępne: Prawidłowy rachunek źródłowy i docelowy został zidentyfikowany i jest znana kwota, która ma być przelana

Sekwencja:

1. Internetowy Bank Zielony Ląd sprawdza dostępne środki
2. Internetowy Bank Zielony Ląd aktualizuje rachunki
3. Internetowy Bank Zielony Ląd aktualizuje informacje na wyciągach

Warunki końcowe:

✓ Okresowe wyciągi odzwierciedlają dokładny zamiar transakcji (przelew jest przelewem, a nie parą operacji wypłata-depozyt)

Rysunek 7.3. Zwyczaj Przesuń pieniądze i Zaksięguj

Chociaż rysunek 7.3 komunikuje ważne wymagania, sam w sobie nie spełnia żadnego celu biznesowego. Ma znaczenie dla dewelopera, ponieważ jest to cykliczny, powtarzający się blok logiki. W związku z tym stanowi dobrą podstawę dla konstrukcji DCI, które implementują co-system-robi. Choć zwyczaj nie ma celu biznesowego, jest ważnym zbiorem szczegółów, które mogą powtarzać się w wielu przypadkach użycia. Ujęcie tego fragmentu w postaci zwyczaju spełnia oczekiwania obu kręgów osób (przedstawicieli biznesu i programistów), które potrzebują pojedynczej, zamkniętej kopii.

Zwyczaje nie powinny mieć odmian. Po pierwsze, trudno jest pokazać, w jaki sposób odmiana na tym poziomie uogólnia się we wszystkich przypadkach użycia wykorzystujących wybrany zwyczaj. Po drugie, odmiana może zbyt łatwo doprowadzić do replikacji i potencjalnie sprzecznych opisów odmian w przypadku użycia i zwyczaju. Odmiany należy uwzględniać na poziomie przypadków użycia.

Warunki wstępne i końcowe dla zwyczaju zwykle są zapisywane na niskim poziomie. Mogą one implikować podejście projektowania kontraktowego (punkt 6.1.2) w kodzie. Są specyficzne dla kodu i mogą wyznaczać testy jednostkowe dla tego kodu. Należy pamiętać, aby nie dodawać warunków wstępnych ani warunków końcowych, które byłyby nieprawidłowe dla dowolnego przypadku użycia wykorzystującego zwyczaj. Należy również pamiętać o tym, aby warunki wstępne i końcowe dotyczące biznesu były umieszczone na poziomie przypadków użycia: powielenie ich może doprowadzić do redundancji i niespójności.

Przycinanie zakresu

A co z drukowaniem wyciągów? Czy posiadacz rachunku naprawdę powinien mieć prawo drukowania wyciągów z rachunku? To jest decyzja należąca do przedstawicieli biznesu. Przedstawiciele zespołu projektowego być może mogą przekonać przedstawicielu biznesu, że korzystne jest opóźnienie tej usługi do czasu, aż Internetowy Bank Zielony Ląd zajmie się szerszą kwestią obsługi dokumentów. Mamy teraz zaktualizowaną listę:

Prywatny posiadacz rachunku
- Logowanie
- Przeglądanie ostatnich transakcji
- Przelew pieniędzy
- Dodawanie stałych płatności
- Opłacanie rachunków

Być może pomyślicie: „To jest zbyt proste. Przegląd ma zawierać tylko pięć przypadków użycia? System, nad którym pracuję, jest znacznie bardziej skomplikowany". Ale ten obraz w gruncie rzeczy jest realistyczny. Złożoność polega na liczbie wariantów w obrębie przypadków użycia, zatem nie jest odzwierciedlony w ogólnej liczbie przypadków użycia. Jeśli zespół ma w widoku helikoptera więcej niż 15 przypadków użycia, to coś jest nie tak! Na kłopoty wskazuje również sytuacja, kiedy w jednym produkcie musimy zarządzać ponad 240 przypadkami użycia na raz (Cockburn 2008).

Więcej przypadków użycia wskazuje na to, że być może mamy nierealistycznie długi plan wydań (na przykład rok do pierwszej wersji). Mogło się również zdarzyć, że zdefiniowaliśmy przypadki użycia na zbyt szczegółowym poziomie. A może uwzględniliśmy przypadki użycia, w których aktorami są tylko inne systemy, a nie role użytkowników reprezentujące prawdziwych ludzi (do takiego celu są lepsze notacje niż przypadki użycia). W większości projektów występuje około ośmiu przypadków użycia. Należy wybrać dobre nazwy aktorów i przypadków użycia. Będziemy posługiwali się nimi przez wiele miesięcy lub lat.

W proces tworzenia i dostrajania przypadków użycia powinni być zaangażowani wszyscy interesariusze. Należy pamiętać o sekrecie Lean. Trzeba zaangażować cały zespół — w tym i programistów, i testerów, ponieważ muszą oni zrozumieć kontekst i szczegóły na tyle dobrze, aby móc je zaimplementować i przetestować. Potrzebujemy przedstawicieli biznesu reprezentowanych przez właściciela produktu, menedżera produktu i im podobnych. Potrzebujemy architekta i specjalisty od interakcji z użytkownikami (jeśli mamy ich w zespole). Potrzebujemy również ekspertów dziedzinowych, bez względu na to, jak nazywamy ich w naszej firmie. Konsolidację widoku helikoptera można wykonać w kilka godzin — jeśli już jesteśmy rozgrzani. To jest Lean.

7.5.2. Ustawianie sceny

Mogą także istnieć inni aktorzy, których chcielibyśmy dodać do sceny. Może to być istniejące oprogramowanie bankowe wykorzystywane przez Internetowy Bank Zielony Ląd. Należy trochę o tym pomyśleć. Jeśli jeszcze tego nie zrobiliśmy, powinniśmy zidentyfikować *ludzi* reprezentujących te systemy, z którymi mamy pracować. Należy dodać tych *systemowych* aktorów do widoku helikoptera, jeśli to poprawia przegląd sytuacji.

To jest właściwy czas na konsolidację motywacji biznesowej i intencji użytkownika. Robi się to na poziomie pojedynczych przypadków użycia. Weźmy za przykład przypadek użycia *Przelew pieniędzy*.

> Motywacja biznesowa: „W ramach naszej strategii *Umożliwić klientom wykonywanie operacji bankowych w domu* postrzegamy przelew pieniędzy jako ważną usługę. Posiadacz rachunku może obserwować swoje rachunki i przelewać pieniądze na rachunek, którego saldo staje się coraz niższe albo o którym wie, że wkrótce osiągnie niskie saldo. Może to oszczędzić nam (bankowi) czasu na wysyłanie listów, gdy nastąpi przekroczenie salda oraz wyeliminuje konieczność zamykania (i późniejszego ponownego otwierania) rachunku. W rozszerzonej usłudze chcielibyśmy także, by posiadacz rachunku mógł przelewać pieniądze na rachunki innych posiadaczy rachunków — zarówno w naszym banku, jak i w innych bankach. Nasi konkurenci już udostępnili tę usługę, dlatego nie możemy zbyt długo czekać, aby także ją zaoferować".

Być może naszemu zespołowi i pozostałym interesariuszom wystarczy zapisanie pierwszego zdania, ponieważ znamy już motywację biznesową i potrzebujemy tylko podpowiedzi. Sposób dokumentowania nie ma zbyt wielkiego znaczenia. Ważne jest, by wszyscy interesariusze znali motywację biznesową. Należy kierować się zdrowym rozsądkiem.

> Intencja użytkownika: „Jako posiadacz rachunku chcę przelewać pieniądze pomiędzy moimi rachunkami, by mieć pewność, że na żadnym z rachunków nie powstanie debet i bank nie zamknie mi karty debetowej".

Tak, można tu umieścić opowieść użytkownika (albo kilka opowieści użytkowników) lub coś innego. Należy używać wyobraźni. Dobra sytuacja jest wtedy, gdy motywacja biznesowa i intencje użytkownika wskazują w tym samym kierunku i nie ma pomiędzy nimi konfliktu.

By ustawić scenę, ważne jest określenie *zakresu* przypadków użycia w relacji względem siebie. Chcemy wiedzieć, jakie warunki otaczają scenariusz przypadku użycia podczas startu, a jakich warunków możemy oczekiwać na końcu. Często się zdarza, że warunek końcowy jednego przypadku użycia jest warunkiem wstępnym innego. W związku z tym definiowanie zakresów także pomaga zweryfikować poprawność widoku helikoptera. Zapewnia też podstawy dla pierwszego zdefiniowania zakresu w biznesie — niezbędnego do zgrubnego planu wydań. Robimy to poprzez dodanie warunku wstępnego i warunku końcowego do każdego przypadku użycia. Kontynuujmy przykład przelewu pieniędzy.

> Warunek wstępny: „Prywatny posiadacz rachunku jest zalogowany w Internetowym Banku Zielony Ląd. Na ekranie wyświetla mu się lista rachunków, których jest właścicielem".
>
> Warunek końcowy: „Kwota, którą wprowadził prywatny posiadacz rachunku, jest przesuwana z rachunku źródłowego na rachunek docelowy. Następuje aktualizacja sald obu rachunków oraz dzienników transakcji".

W pierwszej części etapu konsolidacji jesteśmy trybie „scenariusza słonecznego dnia". Koncentrujemy się na tym, jak rozwinie się sytuacja, kiedy wszystko przebiegnie pomyślnie. Później możemy przenieść te warunki wstępne i końcowe do kodu zgodnie z tym, co powiedzieliśmy w punkcie 6.1.2.

Właściciel produktu jest teraz gotowy na pierwszy, przybliżony plan wydania.

Wydanie 1.: Logowanie i przeglądanie ostatnich transakcji

Wydanie 2.: Przelew pieniędzy

Wydanie 3.: Dodawanie stałych płatności

Wydanie 4.: Opłacanie rachunków

Nie możemy oczywiście jeszcze określić dat, ale to dobry początek, który nakreśla kolejność wykonywanych prac. Scena jest gotowa, ale nic się jeszcze na niej nie dzieje. Pozwólmy grać aktorom.

7.5.3. Odtwarzanie scenariusza słonecznego dnia

Pamiętacie? Dobry przypadek użycia dotyczy *przyrostowych uaktualnień* i *gry zespołowej*. Nie chcemy definiować wszystkiego na raz, ale zastosować zasadę Lean pracy dokładnie na czas. Nie chcemy, by analitycy pisali przypadki użycia, siedząc przy swoich biurkach, ale chcemy, aby stały się one centralnym punktem rozmowy na konkretny temat. Należy rozpocząć od scenariusza słonecznego dnia dla przypadków użycia pierwszego wydania. Pierwsze wydanie w naszym przykładzie zawiera przypadki użycia „Logowanie" i „Przeglądanie ostatnich transakcji". Zarówno przypadek użycia „Logowanie", jak i „Przeglądanie" mogą stwarzać problemy w świecie idealnych przypadków użycia, ponieważ cele użytkownika nie są dla nich oczywiste. Czasami jednak pozwalamy na takie wyjątki z powodów, o których pisaliśmy wcześniej. Nie będziemy tu szczegółowo opisywać przypadków użycia pierwszego wydania. Zamiast tego będziemy kontynuować przykład ze scenariuszem słonecznego dnia przypadku użycia „Przelew pieniędzy" z wydania 2.

Jeśli chodzi o określenie etapów scenariusza przypadku użycia słonecznego dnia, wiele osób posługujących się metodyką Agile poddaje się, uznawszy przypadki użycia za zbyt ciężkie, ale to jest ważna inwestycja i opłaca się ją wykonać. Zapewnienie postępu wymaga myślenia, dyscypliny i decyzji. Myślenie, rozmowa i dyscyplina stanowią jedynie podstawę dla decyzji. Scenariusze przypadku użycia pomagają nam w ustaleniu, jakie decyzje musimy podjąć. Jeśli nie podejmujemy decyzji od razu, powinniśmy pomyśleć, kto je podejmie i kiedy.

Jeśli jesteśmy skłonni odłożyć decyzję, powinniśmy zadać sobie pytanie: czy to jest decyzja, którą programista może podjąć podczas kodowania? Jeśli odpowiedź brzmi „tak", to w porządku — możemy pozostawić decyzję programiście. Jeśli odpowiedź brzmi „nie", należy zadać sobie pytanie, czy można podjąć decyzję teraz, czy też najpierw trzeba zdobyć więcej informacji. Jeśli potrzebujemy więcej informacji, powinniśmy się upewnić, czy zdołamy zdobyć je w odpowiednim czasie. Decyzje podejmowane wcześnie są charakterystyczne dla podejścia Lean, natomiast późne decyzje są typowe dla podejścia Agile. Strukturalne opisy przypadków użycia opracowane przyrostowo mogą nam pomóc w ustaleniu właściwego momentu na podjęcie decyzji.

Strukturalny opis przypadku użycia rozpoczyna się od:

- tabeli[3] zawierającej kolumny z numerami kroków, aktorami przypadku użycia oraz odpowiedzialnością systemu:

Krok	**Intencje aktora** (Constantine i Lucy 1999)	**Odpowiedzialność systemu** (Wirfs-Brock 1993)
1.		

- nazwy aktora lub nazwy systemu na początku każdego kroku:

Krok	**Intencje aktora**	**Odpowiedzialność systemu**
1.	*Posiadacz rachunku* wybiera rachunek źródłowy i decyduje się na przelanie pieniędzy	*Internetowy Bank Zielony Ląd* pokazuje rachunek źródłowy, dostarcza listę rachunków docelowych i pole do wpisania kwoty

- terminologii, która jest starannie przemyślana i spisana:

Krok	**Intencje aktora**	**Odpowiedzialność systemu**
1.	Posiadacz rachunku wybiera *rachunek źródłowy* i decyduje się na przelanie pieniędzy	Internetowy Bank Zielony Ląd pokazuje rachunek źródłowy, dostarcza listę rachunków docelowych i pole do wpisania kwoty
2.	Posiadacz rachunku wybiera *rachunek docelowy*, wpisuje kwotę i zatwierdza ją	Internetowy Bank Zielony Ląd przelewa pieniądze, realizuje *księgowanie* i wyświetla posiadaczowi rachunku *potwierdzenie* wykonanej *transakcji*

Stop! To jest ważne:

Terminy i pojęcia, które wykorzystamy w przypadkach użycia, są danymi wejściowymi dla architektury.

Powyższa czynność nie jest tylko ćwiczeniem „znajdź rzeczowniki", które wykonywaliśmy w starych czasach, kiedy projektowaliśmy klasy. Zapamiętajmy, że to jest wejście do części architektury opisującej, *co system robi* (zobacz, jak to działa, w rozdziałach 8. i 9.). Dobre nazwy utrzymują połączenie z użytkownikami końcowymi i innymi żywymi ludźmi.

[3] Dwukolumnowy format tabeli zaproponowała Rebecca Wirfs-Brock. Wprowadziła ona ideę rozmowy do przypadków użycia poprzez umieszczenie działań użytkownika w lewej kolumnie, a reakcji systemu w prawej kolumnie. Nazwy kolumn zostały opisane w (Constantine i Lockwood 1999). W tej pozycji opisano też pomysł istotnych przypadków użycia, w których szczegóły projektu interfejsu użytkownika (UI) pozostawia się projektantowi interfejsu UI, natomiast szczegóły projektu systemu — projektantowi systemu. Projekt interfejsu UI oraz projekt systemu znacznie lepiej wyraża się za pomocą narzędzi niż za pomocą przypadków użycia.

Krok nie jest krokiem, zanim system nie zareaguje na działania aktora[4]. Powinniśmy myśleć jak użytkownik: czy czujemy, że system coś zrobił, jeśli nie dostajemy z systemu żadnych oznak działania? Możemy też myśleć jak tester: czy jest sens testować wybór docelowego rachunku przez aktora, zanim aktor wprowadzi kwotę i zaakceptuje transakcję? Być może tester chce sprawdzić, czy system pokazuje sensowną listę rachunków docelowych, a to można zrobić po wykonaniu kroku 1. To, czy jest sens tworzyć oddzielny scenariusz testowy dla kroku 1., czy wykonać test dla całego scenariusza słonecznego dnia, zależy od uznania testera. Jednak struktura jest dostępna już teraz, a tester może zacząć pisanie scenariuszy testów w tym samym czasie, kiedy programista zaczyna projektowanie i kodowanie. Ta równoległość zmniejsza opóźnienia w procesie rozwoju.

Wystarczyły dwa kroki, aby opisać scenariusz słonecznego dnia dla jednego przypadku użycia. Czy to trudne? Trudna część nie jest związana z formatem — zespół może się go nauczyć. Trudniejsza jest dyscyplina myślenia i podejmowania decyzji. Musimy zdecydować, jakiej terminologii będziemy używać. W celu udokumentowania tych decyzji może nam się przydać słowniczek. Dziedzina może wiele zyskać nie tylko z powodu posiadania słowniczka przez każdy zespół, ale także dzięki dokumentowi online zawierającemu wspólną terminologię. Słowniczek na tu i teraz może być postrzegany jako podejście Agile, natomiast dokument online z terminologią reprezentuje podejście Lean — nie będzie trzeba w kółko omawiać tych samych pojęć.

Inne kluczowe decyzje pozostają do rozstrzygnięcia. Czy posiadacz rachunku powinien wybrać działanie przelewu przed wyborem konta źródłowego, czy na odwrót? Kto powinien podjąć tę decyzję i kiedy? To powinna być decyzja projektanta interfejsu użytkownika. W grze kooperacyjnej mamy do dyspozycji tę osobę. Może ona podjąć tego rodzaju decyzje. Wszystko w porządku? Może jednak nie... Sformułowania powinny być jak najbardziej elastyczne — zwłaszcza jeśli chodzi o kolejność, w jakiej są wykonywane działania.

Na przykład w kroku 1. nie mówimy, czy faktyczna implementacja będzie typu rzeczownik-czasownik, czy czasownik-rzeczownik. Działanie i rachunek występują w tym samym kroku, a określony porządek nie ma znaczenia z punktu widzenia przypadków użycia, ponieważ dla przedstawicieli *biznesu* kolejność nie jest istotna. W przypadku użycia koncentrujemy się na ogólnym przepływie pracy, przejściach w tę i z powrotem od działania aktora do reakcji systemu z perspektywy biznesu. Jeśli aktor musi wprowadzić i (lub) zaznaczyć wiele informacji w tym samym oknie, nadal uważamy to za jeden krok przypadku użycia. Projektant interfejsu użytkownika musi zaprojektować okno w sposób, który zapewni aktorowi najwygodniejszą obsługę. A interfejs użytkownika powinien jak najbardziej pasować do modelu mentalnego użytkownika końcowego. Aby to zapewnić, budujemy i testujemy prototypy interfejsu użytkownika. Prototypy te dobrze się sprawdzają jako równoległe działania w celu tworzenia definicji przypadków użycia.

Jeśli jeszcze nie zgubiliśmy się w szczegółach interfejsu użytkownika, być może zastanawiają nas inne kwestie. Czy ten scenariusz słonecznego dnia obejmuje przelew na rachunki, które nie należą do posiadacza rachunku? Czy „inne rachunki" oznaczają inne rachunki w tym banku, czy we wszystkich bankach? Czy „wszystkie banki" oznaczają banki w tym kraju, czy we wszystkich krajach? Trzeba zawołać właściciela produktu! Właściciel produktu już tam jest? To dobrze! Zatem możemy iść dalej. Po pierwsze, należy śledzić status podejmowanych decyzji. W tym celu dodajemy kolumnę z komentarzem lub pytaniem do kroku przypadku użycia (tabela 7.1).

[4] Zdarzają się wyjątki, kiedy możemy posługiwać się pustymi krokami dla opisu intencji użytkownika: na przykład aby pokazać, kiedy użytkownik musi poczekać na system, żeby wykonać swoją część.

Tabela 7.1. Scenariusz z komentarzami

Krok	Intencje aktora	Odpowiedzialność systemu	Komentarz
1.	Posiadacz rachunku wybiera *rachunek źródłowy* i decyduje się na przelanie pieniędzy	Internetowy Bank Zielony Ląd pokazuje *rachunek źródłowy*, dostarcza listę *rachunków docelowych* i pole do wpisania kwoty	Czy posiadacz rachunku powinien wybrać działanie przelewu przed wyborem konta źródłowego, czy na odwrót?
2.	Posiadacz rachunku wybiera *rachunek docelowy*, wpisuje kwotę i zatwierdza ją	Internetowy Bank Zielony Ląd przelewa pieniądze, realizuje księgowanie i wyświetla posiadaczowi rachunku *potwierdzenie* wykonanej *transakcji*	Jakie są reguły rządzące rachunkiem docelowym? (Własny rachunek, inne rachunki, inne banki?) Czy potwierdzenie jest właściwym terminem? Czy potrzebujemy potwierdzenia, jeśli przelew jest realizowany pomiędzy własnymi rachunkami posiadacza rachunku? Czy posiadacz rachunku powinien mieć możliwość wydrukowania potwierdzenia?

Zwróćmy uwagę, że przypadki użycia są instrumentem wspierania gry w grę kooperacyjną. Napisanie specyfikacji nie jest spektaklem jednego aktora. Chcemy, aby wszyscy zadawali pytania i pomagali podejmować decyzje. Wtedy wszyscy będziemy znać uzasadnienie decyzji i będziemy mogli poruszać się znacznie szybciej w trakcie implementacji. Wszyscy, razem, od wczesnej fazy.

Istotne pytania, na przykład o inny rachunek, nie powinny być wyłącznie ukryte w kolumnie takiej jak pokazana powyżej. Przedstawiciele biznesu są właścicielami tej decyzji i jest to ściśle związane z planowaniem wydań — oraz architekturą. Co pociąga za sobą wydanie 2. — przelew pieniędzy? Czy chodzi tylko o przelewanie pieniędzy na własne rachunki i z własnych rachunków posiadacza rachunku? A jeśli rozszerzymy to działanie na inne konta, to czy nadal będzie to ten sam przypadek użycia? Tutaj musimy wrócić do analizy modelu mentalnego użytkownika końcowego. Czy posiadacza rachunku obchodzą różnice pomiędzy „innymi" kontami?

System bankowy autorów tej książki nie uważa, że posiadacze rachunków mają inny model dla własnych rachunków, innych rachunków w tym samym banku oraz rachunków w innych bankach. Wszystkie one wyświetlają się na tej samej liście rachunków „docelowych" natychmiast po wprowadzeniu numeru rachunku i nadaniu mu nazwy. Zatem to jest model, którego użytkownicy oczekują. Trudno powiedzieć, czy to nasz bank nauczył nas, aby mieć ten model w pamięci, czy też mieliśmy go wcześniej. Ale wydawało się naturalne, aby myśleć o tym w ten sposób, więc może był to właściwy model od samego początku.

Sprawy mogą się skomplikować tylko wtedy, gdy chcemy przelać pieniądze do banku w innym kraju. Nie jest to tak różne działanie dla nas jako posiadaczy rachunku, ale z oczywistych względów bardzo różne dla banku jako firmy. Potrzebujemy zarówno specjalistów od interakcji z użytkownikami, jak i ekspertów z dziedziny biznesu, aby określić *oczekiwania*. Z punktu

widzenia użyteczności możemy zbadać model mentalny użytkownika końcowego w odniesieniu do rachunków. Czy rachunek „docelowy" jest po prostu innym rachunkiem niezależnie od tego, kto jest jego właścicielem i w jakim kraju? Być może taki jest wniosek. W takim przypadku eksperci dziedzinowi mogą powiedzieć, że procesy i reguły biznesowe leżące u podstaw operacji przelewu międzynarodowego różnią się tak radykalnie od przypadku krajowego, że włączenie go do tego samego przypadku użycia może stwarzać przeszkody w projektowaniu i implementacji. Należy kierować się zdrowym rozsądkiem. Prawdopodobny wynik może być taki, że wszystkie przelewy krajowe na rachunki osób prywatnych będą uwzględnione w przypadku użycia Przelew pieniędzy, natomiast do opisania transakcji na rachunki w innych krajach będzie służył inny przypadek użycia: Przelew na rachunek w banku zagranicznym. Jeśli pominiemy kwestię użyteczności w tej decyzji, narazimy interfejs użytkownika — tak jak w naszym banku, w którym nie jest oczywiste, w jaki sposób należy przelać pieniądze do innego kraju.

Reguły biznesowe

Czy słyszeliśmy, żeby ktoś pytał o reguły biznesowe? To jest duża część działalności bankowej, jak również wielu innych rodzajów działalności. Czy reguły biznesowe wyrażamy za pomocą przypadków użycia? Banki klasyfikują klientów według kryteriów biznesowych, takich jak historia kredytowa, historia debetowa i wartość netto. Dla różnych klas klientów stosowane są różne zasady. O ile możemy przekroczyć saldo, zanim bank zareaguje? Jak długo bank będzie wysyłał wezwania, zanim zamknie naszą kartę kredytową? Czy bank będzie automatycznie przelewał pieniądze z innego naszego rachunku, jeśli będą na nim pieniądze?

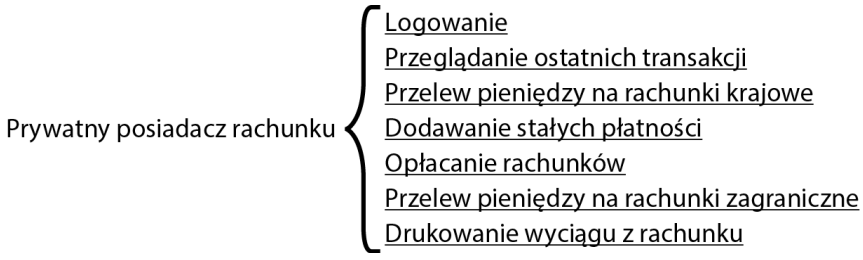
Wszystko to są reguły biznesowe. Przypadek użycia nie powinien wywoływać kategorii klienta za każdym razem, gdy napotkamy usługę, która różni się w zależności od typu klienta. Możemy zacząć od zdefiniowania wspólnych pojęć (takich, które — miejmy nadzieję — już wykorzystujemy w biznesie), na przykład typu klienta. Następnie możemy stworzyć miejsce, gdzie można sprawdzić, jakie typy klientów mamy w biznesie. Czy jest to częścią słowniczka, słownika danych lub opisu reguł biznesowych, zależy od tego, co oznacza (zdrowy) rozsądek w naszej firmie.

Do nazw reguł biznesowych można się odwoływać z przypadków użycia. Na przykład można regule biznesowej nadać nazwę *Przekroczony*. Następnie w słowniku reguł biznesowych możemy ustalić, co oznacza *Przekroczony* dla każdego typu klienta. Zmiany w regułach biznesowych są częstym źródłem nowych wymagań, a reguły biznesowe są kolejnym przykładem wiedzy, która zazwyczaj jest udokumentowana poza językiem programowania.

Być może nasza dziedzina ma charakter techniczny i reguły biznesowe nie stanowią wielkiego problemu. Być może posługujemy się skomplikowanymi algorytmami. Obowiązują tu te same reguły: należy nadać algorytmom nazwy i udokumentować je w innym miejscu — gdzieś, gdzie można je znaleźć i łatwo zaktualizować.

Algorytmy, reguły biznesowe, słownik danych, projekt interfejsu użytkownika... to nie brzmi jak Agile? Cóż, jeśli możemy obyć się bez tych elementów i w dalszym ciągu tworzyć użyteczne oprogramowanie, które stawi czoło konkurencji — to dobrze dla nas. Z naszego doświadczenia jednak wynika, że istnieje ryzyko omawiania tych samych rzeczy w kółko i (lub) pomijania ważnych decyzji biznesowych oraz pozostawiania decyzji dotyczących użyteczności deweloperom przy swoich klawiaturach. Pozostawienie decyzji biznesowych programistom pracującym w izolacji jest ryzykowne — nie wspominając o liczbie przeróbek, które spowoduje takie podejście. Być może takie przeróbki spełniają cechy Agile, ale z całą pewnością nie są Lean.

Nie zapomnieliśmy: podczas pracy ze scenariuszem słonecznego dnia zaktualizowaliśmy widok helikoptera.



Dodaliśmy nowy przypadek użycia do zagranicznego przelewu pieniędzy. A przypadek użycia Drukowanie wyciągu z rachunku powrócił? Dlaczego? Podczas dyskusji na temat przypadku użycia przelewu pieniędzy stało się jasne, że tego rodzaju decyzje są ważne dla architektury (punkt 5.2.2). Wcześniej nie zdawaliśmy sobie z tego sprawy. Zatem chociaż te przypadki użycia nie będą wykorzystane w pierwszych czterech wydaniach, *nadal* mogą mieć wpływ na architekturę, więc nie chcemy ich ukrywać. Najważniejsze przypadki użycia mogą dostarczać informacji o formie nawet w części *czym-system-jest*.

Zaktualizowany plan wydań właściciela produktu może teraz wyglądać następująco:

Wydanie 1.: Logowanie i przeglądanie ostatnich transakcji

Wydanie 2.: Przelew pieniędzy na rachunki w naszym banku

Wydanie 3.: Dodawanie stałych płatności

Wydanie 4.: Opłacanie rachunków

Wydanie 5.: Przelew pieniędzy na rachunki w innych krajowych bankach

Wydanie *n*: Drukowanie wyciągu z rachunku

Wydanie *n+1*: Przelew pieniędzy na rachunki w bankach zagranicznych

Teraz wydanie 2. jest łatwiejsze do przewidzenia i realizacji, niż było wcześniej. Ale wciąż mamy sporo do zrobienia, zanim będziemy mogli umieścić daty na planie wydań. I dlaczego mówimy tak dużo o przepływie pracy użytkownika? Dwa kroki — dla nas to nie jest dużo na temat przepływu pracy! Poczekajmy. Na razie opisaliśmy tylko proste podstawy. Naprawdę interesującymi rzeczami zajmiemy się w następnym punkcie.

7.5.4. Dodawanie interesujących rzeczy

Głównym powodem konsolidacji scenariusza słonecznego dnia tak wcześnie jest stworzenie struktury dla odchyleń. Wszystko, co odbiega od scenariusza słonecznego dnia, jest nowym scenariuszem w ramach przypadku użycia. Gdy dodaliśmy dwa nowe odchylenia, dołożyliśmy co najmniej dwa nowe scenariusze. Kiedy zaczniemy łączyć kilka odchyleń ze scenariuszem słonecznego dnia, mogą one przeniknąć do wielu scenariuszy. Ale co to jest odchylenie? To jest wszystko to, co omówiliśmy i postanowiliśmy, a co nie jest częścią scenariusza słonecznego dnia. Obejmuje to wszystko, o czym jeszcze nie pomyśleliśmy — nieznane niewiadome. W dużej części dotyczy nowych wymagań.

Zatem powróćmy na chwilę do trybu burzy mózgów. Nasz przypadek użycia nazywa się teraz Przelew pieniędzy na rachunki krajowe. Chcemy wymienić wszystko, co przychodzi nam na myśl, a co może odbiegać od scenariusza słonecznego dnia. To oznacza, że musimy zdecydować, jakie są nasze domyślne założenia w odniesieniu do listy rachunków docelowych w scenariuszu

słonecznego dnia. Z jakim rodzajem przelewów będziemy mieć najczęściej do czynienia lub jakie przelewy bank będzie realizował najczęściej? Załóżmy, że priorytetem jest transfer pieniędzy pomiędzy własnymi rachunkami posiadacza rachunku. Przedstawiciele biznesu, uwzględniając informacje od deweloperów, podjęli pewne decyzje. Możemy teraz zaktualizować nasz scenariusz słonecznego dnia (tabela 7.2).

Tabela 7.2. Przelew pieniędzy: scenariusz słonecznego dnia

Krok	Intencje aktora	Odpowiedzialność systemu	Komentarz
1.	Posiadacz rachunku wybiera *rachunek źródłowy* i decyduje się na przelanie pieniędzy	Internetowy Bank Zielony Ląd pokazuje *rachunek źródłowy*, dostarcza listę rachunków docelowych i pole do wpisania kwoty	Czy posiadacz rachunku powinien wybrać działanie przelewu przed wyborem konta źródłowego, czy na odwrót? *Domyślnie Internetowy Bank Zielony Ląd wyświetla listę własnych rachunków posiadacza rachunku (z wyjątkiem rachunku źródłowego)*
2.	Posiadacz rachunku wybiera *rachunek docelowy*, wpisuje kwotę i zatwierdza ją	Internetowy Bank Zielony Ląd przelewa pieniądze, realizuje księgowanie i wyświetla posiadaczowi rachunku potwierdzenie wykonanej transakcji	Czy *potwierdzenie* jest właściwym terminem? Czy potrzebujemy potwierdzenia, jeśli przelew jest realizowany pomiędzy własnymi rachunkami posiadacza rachunku? Czy posiadacz rachunku powinien mieć możliwość wydrukowania potwierdzenia?

Jedyne zmiany nastąpiły w kolumnie komentarza, ponieważ wydaje się ona oczywistym miejscem dokumentowania naszej decyzji lub tej części decyzji. Decyzja o utworzeniu nowego przypadku użycia dla zagranicznego przelewu pieniędzy jest udokumentowana przez sam fakt istnienia nowego przypadku użycia. Pozostała część decyzji — rachunki w innych bankach krajowych — jest częściowo udokumentowana w rozszerzonej nazwie przypadku użycia i będzie dodatkowo udokumentowana w odchyleniu przypadku użycia Przelew pieniędzy na rachunek w banku krajowym. Spróbujmy przyjrzeć się niektórym odchyleniom:

- Przelew pieniędzy na konto krajowe inne niż własny rachunek (czy ma znaczenie, w jakim banku?).
- Zapisanie informacji o rachunku w innym banku i dodanie rachunku do listy rachunków docelowych.
- Sprawdzenie, czy jest wystarczająco dużo pieniędzy, aby zrealizować przelew (jaki komunikat, jeśli nie ma wystarczających środków?).
- Wydrukowanie potwierdzenia (otwarta kwestia).
- Zapytanie o hasło przed transakcją (obowiązkowe?).
- Posiadacz rachunku ma tylko jeden rachunek (komunikat?).

- Kwota jest za niska (jakaś reguła?).
- Kwota jest za wysoka (jakaś reguła?).
- Zaplanowanie przelewu na późniejszą datę.
- Usunięcie dodanych rachunków docelowych.
- Czy transakcja trwa dłużej niż 15 sekund?
- Czy transakcja się nie powiodła? (Z jakich powodów?)

Jeśli przy stole znajdują się przedstawiciele biznesu, testerzy, programiści, specjaliści w dziedzinie interakcji z użytkownikami, ta lista może szybko się rozrastać (tabela 7.3)! Zadawane są coraz to nowe pytania. Jeśli potrzebujemy hasła do zaakceptowania transakcji — czy to jest kolejny krok w scenariuszu słonecznego dnia? Jakie są reguły prawidłowych kwot? Istnieje niezliczenie wiele rzeczy do wyjaśnienia i decyzji do podjęcia. Pamiętajmy również o konsolidacji. Niektórzy uwielbiają tabele (tabela 7.4).

Tabela 7.3. Odchylenia przypadku użycia Przelew pieniędzy

Dotyczy numeru kroku	Działanie powodujące odchylenie	Komentarz
1a	Posiadacz rachunku chce przelać pieniądze na inny własny rachunek	Posiadacz rachunku musi wprowadzić numer rejestracyjny i numer rachunku
1b	Posiadacz rachunku chce dodać inny własny rachunek do listy rachunków docelowych	Posiadacz rachunku może nadać nazwę rachunkowi (obowiązkowo?)
2a	Na rachunku źródłowym nie ma wystarczających środków, aby można było zrealizować przelew	Wyświetlenie komunikatu o błędzie i rezygnacja z transakcji (kto jest odpowiedzialny za komunikaty dla posiadacza rachunku?)
2b	Posiadacz rachunku zatwierdza przelew na inny własny rachunek	Internetowy Bank Zielony Ląd pyta o hasło, zanim zaakceptuje przelew
2c	Kwota nie spełnia reguł poprawności	Reguły poprawności?
2d	Posiadacz rachunku wprowadza późniejszą datę przelewu	Internetowy Bank Zielony Ląd dostarcza opcji umożliwiającej wprowadzenie późniejszej daty. Przelew zostanie zrealizowany w tym dniu, zgodnie z *dniami bankowymi* (jak daleko w przyszłość można planować realizację przelewu?)
2e	Czy transakcja trwa dłużej niż akceptowany minimalny czas?	Powody, dla których transakcja trwa dłużej? Działania podejmowane w takiej sytuacji? Akceptowalny minimalny czas?
2f	Transakcja się nie powiodła?	Powody niepowodzenia? Działania naprawcze? Odpowiednie komunikaty?

Tabela 7.4. Dodanie miejsca na nowo powstające wymagania

Dotyczy numeru kroku	Działanie powodujące odchylenie	Komentarz
1a	Posiadacz rachunku chce przelać pieniądze na rachunek innego klienta	Posiadacz rachunku musi wprowadzić numer rejestracyjny i numer rachunku
1b	Posiadacz rachunku chce dodać rachunek innego posiadacza rachunku do listy rachunków docelowych	Posiadacz rachunku może nadać nazwę rachunkowi (obowiązkowo?)
1c	*Nowe wymaganie...*	
1d	*Nowe wymaganie...*	
2a	Na rachunku źródłowym nie ma wystarczających środków, aby można było zrealizować przelew	Wyświetlenie komunikatu o błędzie i rezygnacja z transakcji (kto jest odpowiedzialny za komunikaty dla posiadacza rachunku?)
2b	Posiadacz rachunku zatwierdza przelew na inny własny rachunek	Internetowy Bank Zielony Ląd pyta o hasło, zanim zaakceptuje przelew
2c	Kwota nie spełnia reguł poprawności	Reguły poprawności?
2d	Posiadacz rachunku wprowadza późniejszą datę przelewu	Internetowy Bank Zielony Ląd dostarcza opcji umożliwiającej zaplanowanie przelewu za pewną liczbę *dni bankowych* w przyszłości (jak daleko w przyszłość można planować realizację przelewu?)
2e	Czy transakcja trwa dłużej niż akceptowany minimalny czas?	Powody, dla których transakcja trwa dłużej? Działania podejmowane w takiej sytuacji? Akceptowalny minimalny czas?
2f	Transakcja się nie powiodła?	Powody niepowodzenia? Działania naprawcze? Odpowiednie komunikaty?
2g	*Nowe wymaganie...*	
2h	*Nowe wymaganie...*	

Podczas konsolidacji za każdym razem odpowiadamy na pewne pytania i podejmujemy pewne decyzje. Zadajemy nowe pytania i uzyskujemy więcej informacji umożliwiających podejmowanie decyzji. Na tym polega natura postępów w rozwoju oprogramowania. Większość odchyleń należy do określonego kroku w scenariuszu słonecznego dnia. Do tego kroku odnosimy

się w pierwszej kolumnie. Jaka jest różnica między tą listą odchyleń a listą wymagań lub funkcji czy opowieści użytkownika? Niezbyt wielka, jeśli spojrzymy na nie jedna po drugiej. Moglibyśmy sformułować każde z odchyleń w postaci opowieści użytkowników, gdybyśmy dzięki temu poczuli się bardziej Agile. „Jako posiadacz rachunku chcę, by system mnie zatrzymał przy próbie przelania większej kwoty, niż mam na rachunku, tak by nie powstał debet".

Cóż, może to nie jest najlepsza opowieść użytkownika na świecie. Forma opowieści użytkownika może pomóc nam zobaczyć, kto jest prawdziwym interesariuszem: „Jako bank chcę zatrzymać klientów przed przelewaniem pieniędzy, których nie mają, tak bym mógł pozostać w biznesie". Nie, taka forma w rzeczywistości także się nie sprawdza. Należy kierować się zdrowym rozsądkiem. Można również zapisać odchylenia na karteczce i umieścić na ścianie (lub na szafie). Następnie możemy nazwać je opowieściami użytkownika w pierwotnym sensie.

Na razie będziemy nadal nazywać takie sytuacje *odchyleniami* od scenariusza słonecznego dnia. A tym, co uzyskujemy z połączenia odchyleń ze scenariuszem słonecznego dnia, jest *struktura*. Możemy teraz omówić krok w scenariuszu słonecznego dnia. Możemy nawet skoncentrować się na systemowej części kroku w scenariuszu słonecznego dnia — tam gdzie spodziewamy się wyzwań architektonicznych. Albo możemy omówić odchylenie i jego konsekwencje. Zapewni nam to ostrość dyskusji zmierzającej do podejmowania dobrych *decyzji*. A przedstawiciel biznesu uzyska strukturalne dane wejściowe do opracowania planu wydań. Aby bardziej szczegółowo zaplanować wydanie 2., przedstawiciel biznesu może teraz przeanalizować listę odchyleń i zdecydować, które z nich muszą się znaleźć w wydaniu 2., a które mogą poczekać.

Kiedy ustabilizujemy scenariusz słonecznego dnia, zespół może go oszacować. A potem, kiedy odchylenia będą ustabilizowane, można je oszacować. Przypadki użycia będą się rozrastać. Nie chodzi o to, że dodajemy więcej przypadków użycia — to są wyjątki. Nie chodzi także o to, że rozbudowujemy scenariusz słonecznego dnia — to także jest rzadkość. Przypadki użycia rozrastają się, ponieważ dodajemy odchylenia.

Oto istotne spostrzeżenie: *odchylenia można dodawać przyrostowo*! Nasz zakres tworzą następujące odchylenia: rozszerzenia, wyjątki, alternatywy, odmiany, obsługa błędów i wymagania niefunkcjonalne. To tu wykonujemy większość prac generujących dochody. Czy wszystkie one mogą być jednolinijkowe? Być może. Z naszego doświadczenia wynika, że tak może być w większości przypadków. Dlatego większość projektów czerpie informacje z opowieści użytkownika; poza tym, że dość często tracą relacje pomiędzy tymi opowieściami. Ponadto, dzięki dbaniu o to, by elementy były niewielkie i proste, utrzymujemy projekt w duchu Agile. Należy kierować się zdrowym rozsądkiem.

Jeśli możemy dodawać przyrostowo przypadki użycia, to możemy również przyrostowo wprowadzać je do wydań. Możemy też przyrostowo je kodować i dostarczać *dokładnie na czas*. To przyrostowe podejście wspiera kluczowy dogmat Lean. Jest rzeczą oczywistą, że wspiera postulat Agile reagowanie na zmiany.

A co z nowo powstającymi wymaganiami? Zajrzyjmy do tabeli 7.4. Puste wiersze czekają na nowo powstające wymagania. Większość nowo powstających wymagań przychodzi „w przebraniu" odchyleń od przepływu słonecznego dnia. „Od wielkiego dzwonu" uzyskujemy zupełnie nowy przypadek użycia. Znajomość zakresu zmian pomaga nam poznać rząd wielkości, z jakim mamy do czynienia. W większości przypadków, kiedy dodajemy lub usuwamy cały przypadek użycia, jest to spowodowane rosnącą wiedzą na temat charakteru naszych przypadków użycia lub nawet dziedziny. Czasami nowy przypadek użycia powstaje z istniejącego przypadku użycia. Przykład widzieliśmy, kiedy przypadek użycia Przelew pieniędzy dał początek nowemu: Przelew pieniędzy na rachunek zagraniczny. Czasami scalamy dwa przypadki użycia, ponieważ zdajemy sobie sprawę, że nie różnią się pomiędzy sobą — być może jeden z nich jest tylko rozszerzeniem (odchyleniem) innego.

W tym momencie warto sformułować ostrzeżenie.

> **Ostrzeżenie!** Jeśli masz zamiar podzielić przypadek użycia taki jak „Przelew pieniędzy" na trzy przypadki użycia: „Wybierz rachunki", „Przelej pieniądze" i „Zaksięguj", to wykonujesz dekompozycję i lepiej będzie, jeśli posłużysz się narzędziami dekompozycji, na przykład diagramami przepływu danych.

Chwileczkę! Nasz specjalista od bezpieczeństwa spogląda mi przez ramię: „Posiadacz rachunku musi wprowadzić hasło tylko wtedy, gdy zatwierdza przelew na inny rachunek? Czy to nie jest część scenariusza słonecznego dnia? Powinna być!". Dlaczego? Czy jest sens prosić o hasło podczas przelewania pieniędzy pomiędzy własnymi rachunkami? „Taką mamy strategię. Powinniśmy zawsze pytać o hasło przed transakcją".

W jaki sposób wpływa to na model mentalny użytkownika? Zadamy pytanie specjaliście w dziedzinie interakcji z użytkownikami.

„Jest jeszcze jedna zasada, która tu obowiązuje: spójność" — mówi specjalista w dziedzinie interakcji z użytkownikami — „więc prawdopodobnie możemy wprowadzić hasła dla wszystkich przelewów bez zakłócania modelu mentalnego użytkownika. Zwłaszcza jeśli robimy to od pierwszego wydania, zanim użytkownicy wypracowali nawyki". W scenariuszu słonecznego dnia znalazł się dodatkowy krok (tabela 7.5).

Tabela 7.5. Dodanie kroku potwierdzania hasłem

Krok	Intencje aktora	Odpowiedzialność systemu	Komentarz
1.	Posiadacz rachunku wybiera *rachunek źródłowy* i decyduje się na przelanie pieniędzy	Internetowy Bank Zielony Ląd wyświetla *rachunek źródłowy*, dostarcza listę rachunków docelowych i pole do wpisania kwoty	Czy posiadacz rachunku powinien wybrać działanie przelewu przed wyborem konta źródłowego, czy na odwrót? *Domyślnie Internetowy Bank Zielony Ląd wyświetla listę własnych rachunków posiadacza rachunku (z wyjątkiem rachunku źródłowego). Kiedy posiadacz rachunku doda inne rachunki, wyświetlą się one na domyślnej liście dla tego posiadacza rachunku*
2.	Posiadacz rachunku wybiera *rachunek docelowy*, wpisuje kwotę i zatwierdza ją	Internetowy Bank Zielony Ląd wyświetla informacje o przelewie (*rachunek źródłowy*, *rachunek docelowy*, datę, kwotę) i żąda hasła w celu zatwierdzenia przelewu	Domyślną datą jest data bieżąca

Tabela 7.5. Dodanie kroku potwierdzania hasłem — ciąg dalszy

Krok	Intencje aktora	Odpowiedzialność systemu	Komentarz
3.	Posiadacz rachunku wprowadza hasło i zatwierdza przelew	Internetowy Bank Zielony Ląd przelewa pieniądze, realizuje księgowanie i wyświetla posiadaczowi informacje z wyciągu oraz dziennik transakcji. Internetowy Bank Zielony Ląd wydaje potwierdzenie transakcji	Zwyczaj: Przelanie pieniędzy i księgowanie. Internetowy Bank Zielony Ląd sprawdza dostępne środki, aktualizuje rachunki i aktualizuje informacje do wyciągu. (Definiujemy *przelew* i *księgowanie*). Czy potwierdzenie jest właściwym terminem? Czy potrzebujemy potwierdzenia, jeśli przelew jest realizowany pomiędzy własnymi rachunkami posiadacza rachunku? Czy posiadacz rachunku powinien mieć możliwość wydrukowania potwierdzenia?

To dobrze, że szybko zdaliśmy sobie sprawę, że potrzebujemy tego dodatkowego kroku! Chcemy, aby scenariusz słonecznego dnia był tak stabilny, jak to możliwe. W przeciwnym razie trudno będzie obsłużyć odchylenia. W świetle tej ostatniej zmiany musimy teraz do nich powrócić. Nasze odchylenie (2b), gdy system prosi o podanie hasła przed przelaniem pieniędzy na inny rachunek, znikło. Czy usuniemy wiersz i zmienimy numerację pozostałych odchyleń? Czy zaznaczymy odchylenie jako usunięte lub przeniesione do scenariusza słonecznego dnia, ale zachowamy wiersz? Należy kierować się zdrowym rozsądkiem. Decyzja i tak jest udokumentowana. Dokumentację uzupełniającą tworzymy tylko wtedy, gdy chcemy mieć zapis, dlaczego została podjęta decyzja, przez kogo i być może kiedy. Odchylenia i tak trzeba przenumerować. Miłej zabawy podczas identyfikacji liczby zmian wprowadzonych na naszej liście odchyleń (tabela 7.6).

Tabela 7.6. Zaktualizowana lista odchyleń

Dotyczy numeru kroku	Działanie powodujące odchylenie	Komentarz
1a	Posiadacz rachunku dodaje tekst do transakcji po stronie rachunku źródłowego	Czy to nie jest część scenariusza słonecznego dnia? Jaki powinien być tekst domyślny, gdyby posiadacz rachunku nie dodał tekstu?
1b	Posiadacz rachunku chce przelać pieniądze na inny własny rachunek	Posiadacz rachunku musi wprowadzić numer rejestracyjny i numer rachunku
1c	Posiadacz rachunku chce dodać inny własny rachunek do listy rachunków docelowych	Posiadacz rachunku może nadać nazwę rachunkowi (obowiązkowo?)

Tabela 7.6. Zaktualizowana lista odchyleń — ciąg dalszy

Dotyczy numeru kroku	Działanie powodujące odchylenie	Komentarz
2a	Na rachunku źródłowym nie ma wystarczających środków, aby można było zrealizować przelew	Wyświetlenie komunikatu o błędzie i rezygnacja z transakcji (kto jest odpowiedzialny za komunikaty dla posiadacza rachunku? Definiujemy *minimalne saldo rachunku*?)
2b	Posiadacz rachunku dodaje tekst do transakcji po stronie rachunku docelowego	Czy to nie jest część scenariusza słonecznego dnia? Jaki powinien być tekst domyślny, gdyby posiadacz rachunku nie dodał tekstu?
2c	Kwota nie spełnia reguł poprawności	Reguły poprawności?
2d	Posiadacz rachunku wprowadza późniejszą datę przelewu	Internetowy Bank Zielony Ląd dostarcza opcji umożliwiającej wprowadzenie późniejszej daty. Przelew zostanie zrealizowany w tym dniu, zgodnie z *dniami bankowymi* (jak daleko w przyszłość można planować realizację przelewu?)
3a	Nieprawidłowe hasło	Standardowa procedura?
3b	Czy transakcja trwa dłużej niż akceptowany minimalny czas?	Powody, dla których transakcja trwa dłużej? Działania podejmowane w takiej sytuacji? Akceptowalny minimalny czas?
3c	Transakcja się nie powiodła?	Powody niepowodzenia? Działania naprawcze? Odpowiednie komunikaty?
Wszystkie	Posiadacz rachunku chce uzyskać pomoc online	Kto jest odpowiedzialny za pomoc online?

Pamiętajmy: bogactwo przypadków użycia objawia się w odchyleniach, a one powstają swobodnie i często. Są tam, gdzie powstaje zróżnicowanie. Skonsolidowaliśmy scenariusz słonecznego dnia wcześnie, aby ustalić strukturę dla odchyleń: to jest fundament. Główny scenariusz sukcesu przypadku użycia to jego stabilna charakterystyka biznesowa, która rzadko się zmienia, jeśli w ogóle. Dostarcza kotwicy dla odchyleń, a odchylenia gwarantują nam wzrost funkcjonalności.

Możemy też zabawić się w znalezienie pięciu błędów (jak w dziecięcej zabawie polegającej na znajdowaniu różnic pomiędzy dwoma obrazkami). Utrzymanie spójności opisu przypadków użycia wymaga wielu par oczu i co najmniej jednego testera.

Ale raczej poświęćmy czas na inne ćwiczenie.

Spróbujmy zidentyfikować wszystkie terminy i pojęcia ze scenariusza słonecznego dnia oraz odchylenia, które naszym zdaniem będą miały wpływ na architekturę.

A co z rejestrem wymagań, jeśli używamy Scrum? Odchylenie zazwyczaj dobrze pasuje jako element rejestru wymagań. Scenariusz słonecznego dnia sam w sobie powinien być jednym elementem rejestru wymagań. Niepełny scenariusz słonecznego dnia nie daje zbyt wielkiej wartości biznesowej. Jeśli scenariusz słonecznego dnia obejmuje więcej zadań, niż może zrealizować jedna osoba w ciągu połowy sprintu, można podzielić scenariusz słonecznego dnia na etapy, ale nie dostarczymy żadnych wartości biznesowych, zanim nie zrealizujemy wszystkich etapów.

7.5.5. Od przypadków użycia do ról

Przedstawiciele biznesu, specjaliści w dziedzinie interakcji z użytkownikiem, testerzy i programiści razem pracują nad przypadkami użycia. Programiści nie są w stanie opracować żadnej implementacji tylko poprzez przekazywanie dokumentów. W szczególności ważne jest, aby dać deweloperom możliwość uzyskania informacji zwrotnych zarówno w celu zrozumienia motywacji i potrzeb użytkownika końcowego, jak i zestawienia ich z własną wiedzą na temat ograniczeń implementacji. Wspólnie oczekiwania obu stron mogą ewoluować.

Często to, co dla przedstawicieli biznesu jest wyjaśniającą definicją, dla dewelopera jest artefaktem programowym. Chcemy ustanowić ścieżkę od jednego do drugiego. Przypadki użycia zachęcają do używania dobrej terminologii, która zapewnia możliwości śledzenia, natomiast architektura DCI dostarcza miejsca do wyrażania tej terminologii w kodzie.

Wyznaczanie ról na podstawie przypadków użycia

Spójrzmy jeszcze raz na tabelę 7.5. Można w niej znaleźć opis następującego kroku:

1.	Internetowy Bank Zielony Ląd wyświetla *rachunek źródłowy*, dostarcza listę *rachunków docelowych* i pole do wpisania *kwoty*	Czy posiadacz rachunku powinien wybrać działanie przelewu przed wyborem konta źródłowego, czy na odwrót?

Zwróćmy uwagę, że wyróżniliśmy terminy: *rachunek źródłowy*, *rachunek docelowy* i *kwota*. Z perspektywy samego przypadku użycia wyróżnienie może oznaczać definicję w słowniku dziedziny. To pomaga przedstawicielom biznesu w wyjaśnieniu swoich potrzeb. Ale deweloperowi terminy te dostarczają silnych wskazówek implementacji. Architektura DCI dąży do tego, aby tego rodzaju pojęcia były uchwycone bezpośrednio w kodzie. W implementacji nazywamy je *rolami obiektowymi*, aby odróżnić je od aktorów z przypadków użycia.

Przypadki użycia pomagają nam zrozumieć odpowiedzialność tych ról. Na przykład z powyższego możemy wywnioskować, że *rachunek źródłowy* powinien dostarczać informacje, które pomagają zidentyfikować go przez posiadacza rachunku. Staje się to obowiązkiem roli obiektowej *rachunku źródłowego*. Wchodząc głębiej w przypadki użycia oraz tematykę przelewu pieniędzy, znajdujemy więcej obowiązków dla *rachunku źródłowego*, *rachunku docelowego* oraz innych *ról obiektowych*. Rozpoznanie tych obowiązków może spowodować konieczność rozwinięcia lub udoskonalenia przypadku użycia.

Opisywany przykład jest prosty. Występuje w nim pojedynczy system o nazwie „Internetowy Bank Zielony Ląd". Częściej będziemy mieli do czynienia z większą liczbą systemów, których nazwy będą się pojawiały w kolumnie systemu (oczywiście może być również więcej aktorów). Każdy z tych systemów będzie miał określone obowiązki w odniesieniu do przypadku użycia. W wielu przypadkach te nazwy systemów staną się jednocześnie rolami obiektowymi.

Dla przykładu rozważmy nowy serwis pozwalający posiadaczowi rachunku na wzięcie pożyczki z banku przez internet albo w bankomacie. Przypadek użycia opisuje wydział kredytów jako system, który musi zatwierdzić wniosek o pożyczkę (być może automatycznie, w czasie rzeczywistym). Z punktu widzenia głównego przypadku użycia, który w większości opisuje dyskusje pomiędzy posiadaczem rachunku a Internetowym Bankiem Zielony Ląd, wydział kredytów to po prostu kolejna rola. Członkowie zespołu projektowego chcą to widzieć w taki sam sposób. W architekturze DCI stanie się on rolą obiektową.

Czy zespół definiuje te role w ramach analizy, czy w ramach projektu? Czy tę dyskusję powinniśmy prowadzić tutaj, w rozdziale 7., czy w rozdziale 9.? To właśnie są ważne pytania, z jakimi spotykamy się w życiu. Właściwie jest nam wszystko jedno. Pamiętajmy: cały zespół pracuje jako całość. Dzięki temu przedstawiciele biznesu mogą dostarczać informacje programistom, a programiści mogą dostarczać informacje przedstawicielom biznesu. Wszyscy, razem, od wczesnej fazy. Doświadczenie nauczy nas szczegółów tego procesu.

Wypełnianie luki pomiędzy przedstawicielami biznesu a programistami

Na rysunku 7.3 wprowadziliśmy pojęcie zwyczaju, który obejmuje powtarzający się fragment przypadku użycia w zamkniętej formie. W klasycznym stylu przypadków użycia odpowiedzialność po stronie oprogramowania jest oznaczana jako odpowiedzialność „systemu" albo, jeśli jesteśmy bardziej selektywni w naszej terminologii, „banku". Terminy te przekazują bardzo mało nowych informacji zarówno przedstawicielom biznesu, jak i programistom. Zamiast uciekania się do używania ogólnego języka branżowego oraz w celu uniknięcia posługiwania się taką terminologią, jak *abstrakcyjne klasy bazowe* lub *dzienniki transakcji bazy danych*, możemy skorzystać z lepszych metafor do opisania tych pojęć. W szczególności możemy zaprezentować wiele z tych metafor w postaci ról obiektowych. Role obiektowe są konstrukcjami programowymi w architekturze DCI, które odzwierciedlają elementy modelu mentalnego użytkownika końcowego.

Na rysunku 7.3 używamy raczej sterylnego terminu „Internetowy Bank Zielony Ląd" („system") do zaznaczenia wszystkiego, co robi oprogramowanie. To pozostawia użytkownikowi końcowemu niewielki wgląd w to, co się dzieje, a programiście daje nadmiernie szerokie i mgliste pojęcie kontekstu. Rozważmy alternatywną reprezentację sekwencji:

1. **Rachunek źródłowy** weryfikuje dostępne środki.
2. **Rachunek źródłowy** i **rachunek docelowy** aktualizują salda.
3. **Rachunek źródłowy** aktualizuje informacje na wyciągu.

Powyższy model daje użytkownikowi końcowemu metaforyczny sens. Co więcej, pomaga programiście zamknąć pętlę sprzężenia zwrotnego, aby upewnić się, że współdzieli on model mentalny użytkownika końcowego. W tym przypadku nowe terminy: **rachunek źródłowy** i **rachunek docelowy** nie są aktorami przypadku użycia, ale są podobnymi do aktorów agentami wewnątrz systemu Internetowego Banku Zielony Ląd — rolami obiektowymi. W implementacji staną się one pojęciami programowymi reprezentowanymi przez abstrakcyjne klasy bazowe lub inne interfejsy. Do czego dostarczają interfejs? Do obiektów. Tak jak aktor przypadku użycia jest perspektywą roli dla pewnej istoty ludzkiej, tak rola obiektowa jest behawioralną częścią obiektu programowego. Role obiektowe opiszemy szczegółowo w rozdziale 9.

Opisywane podejście należy stosować selektywnie, pamiętając, że wszystkie analogie (i metafory) gdzieś zostaną zastąpione. Nie należy wymuszać metafory, jeśli zespół nie potrzebuje określonego pojęcia lub wizji, które poprawią zrozumienie i posuną prace do przodu.

7.6. Podsumowanie

Czy spełniliśmy nasze kryteria konsolidacji? Spróbujmy podsumować. Naszym celem było:

1. Wsparcie przepływu pracy użytkowników (scenariusze użycia).
2. Wsparcie testów, które powinny być blisko prac rozwojowych (scenariusze testów).
3. Wsparcie dla skutecznego procesu podejmowania decyzji o funkcjonalnościach (scenariusz słonecznego dnia a odchylenia).
4. Wsparcie dla nowo powstających wymagań (odchyleń).
5. Wsparcie dla planowania wydań (przypadki użycia i odchylenia).
6. Uzyskanie danych wejściowych do opracowania architektury (terminy i pojęcia ze scenariuszy).
7. Budowanie w zespole zrozumienia przedmiotu pracy.

7.6.1. Wsparcie przepływu pracy użytkowników

Ustabilizowaliśmy scenariusz słonecznego dnia. Zdecydowaliśmy się na trzy kroki (i ustaliliśmy ich kolejność), które użytkownik i system mają do wykonania, aby otrzymać pożądany rezultat. Odchylenia, które zdefiniowaliśmy do tej pory, należą do konkretnego kroku w scenariuszu słonecznego dnia. Kiedy przypadek użycia rozrasta się przyrostowo za pośrednictwem odchyleń, musimy zdecydować, gdzie umieścić każde z tych odchyleń w przepływie pracy. Dzięki temu utrzymujemy świadomość wspieranego przepływu pracy. Czy jesteśmy przekonani o tym, że system będzie wspierać najlepszy przepływ pracy dla użytkownika? Należy zapytać specjalistę w dziedzinie interakcji z użytkownikiem. Można zrobić to, co zrobiłaby ta osoba — przeprowadzić test użyteczności.

7.6.2. Wsparcie dla testów blisko prac rozwojowych

Po ustabilizowaniu scenariusza słonecznego dnia tester (lub inżynier systemowy — punkt 3.2.1) może zacząć definiowanie scenariuszy testowych. Jeśli tester czuje się na tyle dobrze, że może przystąpić do pisania scenariuszy testowych, wiemy, że nasz przypadek użycia jest w bardzo dobrej formie. Działania testera dodadzą wiele cennych odchyleń do przypadku użycia. Struktura scenariuszy testowych jest już określona przez relacje pomiędzy scenariuszem słonecznego dnia a odchyleniami. Teraz tylko do testera należy zdefiniowanie scenariuszy, a zwłaszcza ich kombinacji, dla których testowanie ma sens. Ta decyzja jest zasadniczą kompetencją testów. Nie ryzykujemy tego, że wymagania będą obsługiwać jedną strukturę, projekt i kod — kolejną, a test będzie obsługiwał jeszcze inną. Z perspektywy historycznej rozwiązania tego dylematu można podzielić na dwie kategorie:

1. Klasyczne rozwiązanie polega na wstrzymaniu wykonywania scenariuszy testowych do czasu, aż scenariusze użycia zostaną zakodowane. Efekt: opóźnienia w pracach rozwojowych i testy w roli „musztardy po obiedzie” (myślę, że większość z nas tego doświadczyła).
2. Podejście Agile, w którym testy są pisane przed kodem. Efekt:
   a) Wymagania są wyrażone za pomocą skryptów testowych, przez co tracimy przedstawicieli biznesu, specjalistów od interakcji z użytkownikami i innych interesariuszy, którzy nie rozumieją kodu; b) zbyt wcześnie zagłębiamy się

w szczegóły; c) tracimy możliwość określenia tego, co jest ważne do przetestowania, i stosujemy podejście „testowania wszystkiego", ryzykując uzyskanie fałszywego poczucia bezpieczeństwa.

Scenariusze przypadków użycia pomagają zyskać pewność, że wszyscy jesteśmy „na tej samej stronie", zanim zakodujemy i zaczniemy testować scenariusze. Projekt systemu i projekt testów mogą odbywać się równolegle. Jesteśmy gotowi do testowania natychmiast po zakodowaniu scenariusza.

7.6.3. Wsparcie dla wydajnego podejmowania decyzji na temat funkcjonalności

Mamy nadzieję, że przykłady zaprezentowane w tekście przekonują Was o tym, że mamy potrzebne wsparcie dla procesu podejmowania decyzji. Bieżący status scenariusza słonecznego dnia, odchylenia, opis każdego scenariusza oraz komentarze i pytania — wszystko to odzwierciedla aktualny stan decyzji dotyczących tego, co robi system. Kiedy zajdzie potrzeba podjęcia nowej decyzji, będziemy znali dokładny kontekst: odpowiedni przypadek użycia, czy jest to scenariusz słonecznego dnia, czy odchylenie, który to krok w scenariuszu słonecznego dnia lub które odchylenie. Wszystko to bardzo wspiera proces podejmowania decyzji!

7.6.4. Wsparcie dla nowo powstających wymagań (odchyleń)

Mogą powstać nowe przypadki użycia lub jeden przypadek użycia może być połączony z innym. Przypadki użycia łączymy wtedy, kiedy zdamy sobie sprawę, że dwa przypadki użycia są w zasadzie takie same. Jeśli taka sytuacja występuje często, powinna nam się zapalić czerwona lampka: nasze przypadki użycia od początku są zbyt szczegółowe. Bardziej organicznym sposobem pracy z przypadkami użycia jest umożliwienie powstawania nowych przypadków użycia — jeden przypadek użycia powstaje na bazie innego przypadku użycia. Należy uważać, aby taka sytuacja nie zdarzyła się przedwcześnie — wtedy ryzykujemy, że później wystąpi problem scalania. Zyskamy poczucie, że jest zbyt wcześnie, ponieważ powstający przypadek użycia będzie sprawiał wrażenie niedojrzałego. Ale najczęściej nowe przypadki użycia będą powstawały na poziomie odchyleń. Po drodze dodajemy, a czasami usuwamy odchylenia. Odchylenia są implementowane w różnych iteracjach w zależności od kolejności rejestru wymagań wyznaczonej przez właściciela produktu (o ile pracujemy w Scrumlandzie). Inaczej mówiąc: odchylenia są dobrze podzielonymi jednostkami, których możemy użyć, aby plan wydań stał się bardziej realistyczny.

7.6.5. Wsparcie dla planowania wydań

Po zidentyfikowaniu mniej więcej 15 przypadków użycia możemy przystąpić do opracowania pierwszego, surowego planu wydań. Po skonsolidowaniu scenariusza słonecznego dnia i omówieniu odchyleń możemy stworzyć nieco bardziej szczegółowy plan wydań. Plan wydań jest gotowy do wprowadzenia dat w chwili, gdy zostaną oszacowane scenariusze słonecznego dnia oraz najważniejsze odchylenia.

Po ustaleniu dat wydań pozostaje problem opracowania najważniejszych elementów dla każdego wydania. Z natury scenariusz słonecznego dnia jest najważniejszy dla każdego przypadku użycia. Implementacja odchylenia przed scenariuszem słonecznego dnia nie jest rozsądnym podejściem. Ponadto decyzja o tym, jakie odchylenia powinny się znaleźć w wydaniu, należy do przedstawicieli biznesu. *To, co sprawia, że przypadki użycia są obszerne, to długa lista odchyleń*,

a nie scenariusz słonecznego dnia — nie powinien zawierać więcej niż siedem kroków, tymczasem w większości przypadków zawiera ich mniej. Natomiast jeden przypadek użycia może z łatwością obejmować od 30 do 50 odchyleń (pomyślmy tylko o obsłudze błędów), a liczba scenariuszy może rosnąć wykładniczo z każdym dodanym odchyleniem. Więc prawdziwa złożoność i zakres są ukryte w odchyleniach. Możemy sprostać temu zadaniu, korzystając z naszej elastyczności i zdolności do bycia Agile. Odchylenia są jednostkami, które możemy wykorzystać podczas dokonywania zmian w priorytetach i kolejności, kiedy obsługujemy nowo powstające wymagania oraz gdy dbamy o konsekwencje planowania wydań.

7.6.6. Uzyskanie danych wejściowych do opracowania architektury

Opisy przypadków użycia pomagają nam uświadomić sobie pojęcia i terminologię. Świadomość pojęć i terminów wymaga ich *zrozumienia*. Aby opracować dobrą architekturę, musimy rozumieć system. Należy wyodrębnić terminy i pojęcia z przypadków użycia i *konsekwentnie* używać słów w naszej architekturze i kodzie. Pomaga nam to uzyskać *czytelny* kod. O tym, że przypadki użycia wpływają na formę kodu, przekonamy się podczas korzystania z architektury DCI (rozdział 9.). Czytelny kod zapewnia *łatwość utrzymania*. Kod łatwy w utrzymaniu pomaga architekturze przetrwać dłużej. Dobra, trwała architektura daje większe zadowolenie programistom, a przedsięwzięcie czyni bardziej wartościowym. Uwielbiamy happy endy.

7.6.7. Budowanie w zespole zrozumienia przedmiotu pracy

Do zapewnienia szczęśliwego zakończenia potrzebujemy kilku dodatkowych szczegółów. Aby zespół rozumiał produkt, musi być zaangażowany w pracę nad produktem. Można mieć najlepsze przypadki użycia w świecie, napisane przez najlepszych na świecie analityków. Jeśli jednak zespół nie jest częścią procesu tworzenia produktu, przypadki użycia są bezużyteczne. Aby zrozumieć produkt, zespół musi zacząć wszystko od początku. To nie dlatego, że członkowie zespołu są głupi. To dlatego, że 90% wiedzy jest teraz w głowie analityka, a reszta być może w przypadkach użycia. Jeśli opowieści użytkowników są „obietnicą przyszłej rozmowy", to przypadki użycia są „przypomnieniem o dyskusji i decyzjach" dotyczących tego, co system robi. Dobre jest to, że są to *strukturalne* przypomnienia!

Kiedy więc Kent Beck (Beck 1999, s. 90) pisze: „Biznes pisze historię", a Scrum podkreśla, że właściciel produktu jest odpowiedzialny za rejestr wymagań, nie odczytujcie tego tak, że przedstawiciele biznesu piszą opowieści użytkowników w izolacji, a właściciel produktu tworzy rejestr wymagań bez udziału zespołu. Praca bez udziału zespołu powoduje wprowadzenie specyfikacji typu „przerzuć przez mur" do wytwarzania oprogramowania z wykorzystaniem technik Agile i Lean. Specyfikacje typu „przerzuć przez mur" to główny powód, dla którego projektowanie kaskadowe ma dziś tak złą reputację. Lean ma na celu wyeliminowanie kaskad, czy też murów, przez które przerzuca się efekty pracy. Agile dotyczy indywidualnych osób i interakcji. Zachowajmy ducha Agile i Lean: *wszyscy*, *razem*, *od wczesnej fazy*.

7.7. „To zależy": kiedy przypadki użycia nie są dobre?

Przypadek użycia rejestruje sekwencję zdarzeń — kolekcję powiązanych ze sobą scenariuszy — które pozwalają przejść użytkownikowi końcowemu w kierunku jakiegoś celu w pewnym kontekście. Co zrobić, jeśli mamy trudności w wyrażaniu interakcji użytkownika w takiej formie? Czasami przypadki użycia nie są dobrym wyborem. Jeśli spojrzymy na scenariusz i nie możemy odpowiedzieć na pytania o kontekst: „Dlaczego użytkownik to robi?" oraz „Jaka jest jego (lub jej) motywacja w tym kontekście?", to przypadki użycia prawdopodobnie są złym środkiem dopasowanym do dziedziny lub nie mamy wystarczająco dobrej łączności z użytkownikami. Jeśli zachodzi ta pierwsza sytuacja, to należy poważnie zbadać alternatywne formalizmy uchwycenia wymagań.

Powstaje pytanie: *jakie alternatywy?* Odpowiedź brzmi: to zależy. Niektóre projekty w rzeczywistości są zdominowane przez takie operacje. Wyobraźmy sobie edytor figur graficznych, który oferuje takie „przypadki użycia" jak obracanie, kolorowanie, zmiana rozmiaru, tworzenie i usuwanie obiektów. Z perspektywy przypadku użycia są to operacje atomowe. Pomimo to każda z nich może wymagać wielu ruchów użytkownika końcowego. Każdy z nich jest nadal jest pojedynczym krokiem w przypadku użycia. Liczba ruchów lub kliknięć myszą potrzebnych do osiągnięcia celu jest kwestią projektu interfejsu użytkownika, a nie kwestią wymagań. Przesunięcie obiektu lub zmiana jego koloru nie jest algorytmem w matematycznym, a nawet potocznym tego słowa znaczeniu — to jest atomowa operacja. Jeśli nie ma biznesowej potrzeby, by grupować takie operacje, to każda z nich jest osobną komendą i w rzeczywistości nie potrzebujemy przypadków użycia. Użytkownik nie wykonuje *sekwencji zadań w celu osiągnięcia jakiegoś celu w kontekście*. Każde polecenie sprowadza się do prymitywnej operacji na samym obiekcie dziedziny i sam wzorzec MVC ze swoją metaforą bezpośredniej manipulacji wystarczy do wykonania pracy. Jeśli zdecydujemy się skorzystać z przypadków użycia w celu uchwycenia wejścia do systemu zdominowanego przez atomowe operacje, możemy uzyskać setki przypadków użycia opisujących oczywiste scenariusze.

7.7.1. Klasyczne programowanie obiektowe: architektury atomowych zdarzeń

Jednym ze sposobów na wyjaśnienie sukcesu programowania obiektowego jest to, że umożliwiło ono wykorzystanie graficznych interfejsów użytkownika i myszy. Te style interakcji człowieka z komputerem zdobyły naszą wyobraźnię i doprowadziły do tego, że użytkownicy zaczęli uważać komputery za towarzyszy podczas rozwiązywania problemów, a nie za odległych podwykonawców. Ludzkie zadowolenie z natychmiastowej informacji zwrotnej, szybkość (lub przynajmniej złudzenie szybkości) uzyskania rozwiązania i przypominające „ludzkie" zachowania interfejsów w Xerox PARC utorowały drogę dla wpływów Smalltalka na świat.

Związek między zaangażowaniem użytkowników i Smalltalkiem w szczególności lub programowaniem obiektowym w ogóle jest więcej niż przypadkowy. Istnieje ciągłość reprezentacji prowadząca od modelu mentalnego świata użytkownika końcowego, poprzez interfejs graficzny, do struktury obiektów należących do interfejsu. Koncepcja jest taka, że użytkownik manipuluje elementami interfejsu użytkownika w sposób, który skłania go do myślenia, że bezpośrednio manipuluje obiektami w programie — lub, jeszcze lepiej, obiektami świata rzeczywistego reprezentowanymi przez te obiekty programowe. Umieszczenie towaru w koszyku w księgarni internetowej daje użytkownikowi końcowemu, przynajmniej metaforycznie, poczucie prawdziwych zakupów z koszykiem. Zjawisko to czasami jest nazywane metaforą bezpośredniej manipulacji (Laurel 1993).

W dalszej części książki omówimy architekturę *Model-View-Controller-User* (podrozdział 8.1) jako najpopularniejszy sposób tworzenia iluzji obsługi tej metafory.

Aby coś zrobić na interfejsie GUI z bezpośrednią manipulacją, najpierw wybieramy obiekt (na przykład książkę), a następnie manipulujemy nim (umieszczamy w koszyku). Wydaje się niezręczne, aby zamiast tego najpierw powiedzieć: „Chcę kupić książkę", a następnie ją wybrać. Jest tak dlatego, że przed faktycznym podjęciem decyzji o *realizacji* zakupu najpierw *myślimy* o książce. Prawdopodobnie nie odwiedzilibyśmy strony, gdybyśmy mieli rozstrzygać o tym, czy chcemy kupić książkę. Ten styl interakcji określa się jako *rzeczownik-czasownik*: najpierw wybieramy rzeczownik (książkę), a następnie akcję (zakup).

Zwróćmy uwagę na ścisłe powiązanie z naszą architekturą, która zawiera część czym-system-jest (książki) oraz co-system-robi (takie przypadki użycia jak: zakup książki, wyszukiwanie wcześniejszych wydań książki oraz odczytywanie recenzji książki).

Metafora rzeczownik-czasownik prowadzi do interfejsów, które przeprowadzają użytkownika przez sekwencję ekranów lub kontekstów. Kiedy wybieramy książkę, znajdujemy się w kontekście myślenia o tej książce. Książka ma fizyczną reprezentację na ekranie (zdjęcie okładki, ISBN lub tytuł i autor), która wypełnia nasz umysł. Procesy świadomości w naszym umyśle koncentrują się na tym obiekcie. Jest wiele czynności, które możemy zrobić z tym obiektem, a proces myślenia prowadzi nas do miejsca, w którym chcemy być. Dopiero wtedy „wykonujemy komendę". Ta „komenda" jest poleceniem w rodzaju „włóż tę książkę do koszyka z zakupami". Na większości interfejsów graficznych będziemy odtwarzać ten scenariusz przez naciśnięcie przycisku oznaczonego etykietą „Dodaj do koszyka".

Oznacza to, że obiekt na ekranie (książka) jest jak obiekt w programie, a na ekranie widzimy, jak możemy manipulować tym obiektem w sposób, który ma sens biznesowy (umieścić go w koszyku, pobrać recenzje itp.). Są to w istocie funkcje członkowskie wykonywane na tym obiekcie. Klient pracuje i wywołuje polecenia — w niezwykle skonkretyzowanym kontekście: tzn. na tym obiekcie. Obiekt staje się interpreterem poleceń — jest zdolny do wykonania jednej prymitywnej operacji na raz.

Z kolei ten styl interakcji prowadzi do stylu architektury, którą w tej książce nazywamy *architekturą zdarzeń atomowych*. Taki styl koncentruje się na obiektach oraz tym, co możemy z nimi zrobić, a nie na „scenariuszach", które istnieją w retrospekcji jako zbiór tych zdarzeń i komend. Kontekst możemy przyjąć za pewnik. Dokładniej skupiamy się na rolach, jakie obiekty mogą spełniać w bieżącej sytuacji (choć książka może służyć za stoper do drzwi, kupując książkę online, koncentrujemy się na innej roli). Dajemy użytkownikom wybór, pozwalając im na interakcję z programami w sposób Agile: w danym momencie mogą dokonać jednego wyboru. Gdybyśmy zamiast tego skupili się na scenariuszu, mielibyśmy główny plan, który wykluczałby wybieranie różnych książek w różnej kolejności oraz eliminowałby opcję użytkownika do zmiany kursu w dowolnej chwili (żeby zapomnieć bieżącą książkę i przejść do innej albo „wyjąć" książkę z koszyka na zakupy).

7.8. Testowanie użyteczności

Testy użyteczności skupiają się na interakcji pomiędzy użytkownikiem końcowym a systemem. Ich celem jest zapewnienie, aby typowi użytkownicy końcowi, którzy mieszczą się w typowych profilach użytkowników mogli osiągnąć to, co chcą osiągnąć. Przeprowadzenie dobrych testów użyteczności wymaga umiejętności specjalnie wyszkolonych i doświadczonych testerów. Specjaliści w dziedzinie interakcji z użytkownikami wykonują takie testy, dając użytkownikom końcowym

przykładowe ćwiczenia lub zadania do wykonania, i monitorują interakcje pomiędzy użytkownikiem końcowym a planowanym systemem. Dokładnie notują, które ekrany są wykorzystywane w przypadku użycia oraz jakie interakcje zachodzą między użytkownikiem a systemem. Tego rodzaju test w kontekście architektury sprawdza, czy uchwyciliśmy model mentalny użytkownika końcowego.

Firmy zbyt często odkładają testy użyteczności do czasu ukończenia prac rozwojowych nad oprogramowaniem, traktując je jak testy akceptacyjne. W tym momencie jest już za późno. Jeśli jest potrzebna fundamentalna zmiana w interfejsie, to sygnalizuje ona niepowodzenie uchwycenia modelu mentalnego użytkownika końcowego. Może to oznaczać, że architektura jest oparta na błędnych pojęciach. Jeśli koncepcyjne podwaliny są złe, to nie wystarczy refaktoryzacja, aby poprawić sytuację. Może to oznaczać opóźnienie aktualnego wydania lub odłożenie poprawki do kolejnej wersji. Zbyt często konieczne zmiany wpływają na podstawową formę architektury tak bardzo, że koszty są zbyt duże, by można było je ponieść, a użytkownicy muszą w nieskończoność borykać się z problemami.

W związku z tym lepszym podejściem jest wykorzystywanie testów użyteczności jako techniki inżynierii postępowej. Należy skorzystać z informacji przekazanych z testów użyteczności do modeli interfejsu w celu dostrojenia architektury. To może być trudne w tradycyjnych organizacjach, ponieważ często występuje w nich ogromna luka pomiędzy architektami, implementatorami i specjalistami w dziedzinie interakcji z użytkownikami. Ale w zespołach Lean, stosujących zasadę: wszyscy, razem, od wczesnej fazy, istnieją podstawy organizacyjne do takiej pętli sprzężenia zwrotnego.

Możemy pójść o krok dalej poprzez połączenie testów użyteczności z dynamicznymi testami architektury przy użyciu kart CRC (ang. *Candidate object, Responsibilities, and Collaborators* — z ang. *obiekt-kandydat, odpowiedzialności i współpracownicy*) (Beck 1991). Należy zorganizować naradę zespołu i pozwolić każdemu jego członkowi odgrywać rolę jednego lub większej liczby obiektów w systemie. Każdy członek zespołu posiada kartę reprezentującą albo obiekt dziedzin systemu, albo rolę systemu, albo rolę obiektową, zgodnie z wprowadzeniem w punkcie 7.5.5. Prowadzący prosi użytkownika końcowego o wykonanie czynności ze scenariusza przypadku użycia. Kiedy użytkownik prosi o wykonanie określonej czynności (wciśnięcie przycisku na ekranie lub rozwinięcie menu), osoba reprezentująca rolę lub obiekt wstaje, dostarczając właściwą usługę. Odpowiedzi są obowiązkiem obiektu lub interfejsu publicznego roli. Członek zespołu zapisuje je w lewej kolumnie karty natychmiast po ich pojawieniu się. Jeśli rola lub obiekt wymaga współpracy innej roli lub obiektu do realizacji transakcji, odpowiedni członek zespołu zostaje poproszony o zrealizowanie zadania i zapisanie odpowiedniego obowiązku na swojej karcie. Kiedy pierwszy członek zespołu przekazuje sterowanie drugiemu, zapisuje rolę lub obiekt drugiego w prawej kolumnie karty. Warto użyć pałeczki lub żetonu albo innego „świętego artefaktu" w roli licznika programu w tym ćwiczeniu.

Po zakończeniu ćwiczenia warto porównać to, co jest na kartach, z tym, co zostało ujęte w architekturze.

7.9. Dokumentacja?

Przede wszystkim należy zapamiętać, że naszym głównym celem jest doprowadzenie do sytuacji, w której członkowie zespołu ze sobą rozmawiają. Każdy ma jakiś zbiór spostrzeżeń i każdy potrzebuje spostrzeżeń innych. Tradycyjne firmy utrudniają budowanie sieci powiązań umożliwiającej przekazywanie właściwych spostrzeżeń do ich miejsc przeznaczenia. Opowieści użyt-

kowników są rodzajem artefaktu kulturowego, który zespoły mogą wykorzystać jako punkty kontaktowe w dyskusjach prowadzących do wspólnej wizji pracy. Jak mówi Mike Cohn, opowieści użytkowników raczej *reprezentują* wymaganie, niż *są* wymaganiem. Przypadki użycia są do pewnego stopnia takie same, ale również dostarczają strukturę wymagań, rejestr tego, w jaki sposób powstała struktura, i wynikające z niej wymagania. Pisana literatura dotycząca kultury daje podstawy, których nie jest w stanie dostarczyć kultura przekazywana ustnie, a większość wartości przypadków użycia polega na skonkretyzowaniu decyzji, w szczególności tych dotyczących odpowiedzi na pytanie *dlaczego?* w projekcie.

Biorąc pod uwagę ten kontekst, trzeba zauważyć, że same przypadki użycia są ważnymi dokumentami Agile. Należy pamiętać, że dokumenty spełniają dwa cele: komunikację bieżącą oraz pamięć organizacji w przyszłości. Przypadki użycia są formą, która pomaga w kształtowaniu myślenia w teraźniejszości. Zapewniają miejsce pozwalające na śledzenie zależności pomiędzy elementami, które jeszcze są do zaimplementowania w rejestrze wymagań, oraz pracami w trakcie realizacji w rejestrze sprintu w przypadku metodyki Scrum. Na tym po części polega rola specyfikacji stwarzającej możliwości.

To, czy przypadki użycia powinny być zachowane jako dokumenty historyczne, jest sprawą decyzji w indywidualnych projektach. Przypadki użycia warto zachować wtedy, gdy kodyfikują ważną wiedzę, do której zespół może chcieć powrócić w przyszłości. Czasami chcemy zapamiętać, dlaczego zrobiliśmy coś w określony sposób — zwłaszcza gdy rozumowanie nie wynika w oczywisty sposób z samego kodu. Dobrym przykładem jest kod obsługujący efekt uboczny wynikający z interakcji pomiędzy dwoma przypadkami użycia. Być może zachowanie przypadków użycia wyeliminuje konieczność ponownej pracy zmierzającej do ustalenia rozumowania i mechanizmów, które były zrozumiałe, gdy w przeszłości podejmowano decyzje projektowe. Dlatego wyrzucanie przypadków użycia nie byłoby Lean. Jeśli więc mamy potrzebę posiadania takiej historii i znamy sposób łatwego odzyskania takich dokumentów na żądanie, możemy zastanowić się nad ich zachowaniem.

Warto zachować trochę danych na temat tego, jak często wracamy do starych przypadków użycia. Raz na jakiś czas każdy z nas, autorów tej książki, przegląda zawartość swojej szafy. Jesteśmy zdumieni, jak wiele z naszych ubrań nie zostało użytych w ciągu roku lub dwóch lat. To jest zawsze trudna decyzja, aby oddać je na cele charytatywne, ponieważ chcemy je zachować ze względów sentymentalnych. Ale dysponujemy ograniczoną ilością miejsca w szafie. Zachowanie przypadków użycia i utrzymanie ich aktualności wymaga kosztów. Jeśli okaże się, że zarchiwizowany przypadek użycia nie ujrzał światła dziennego w ciągu kilku miesięcy, nie warto dalej go utrzymywać. Zróbmy porządek w szafie. Jest to część zasady Lean utrzymania porządku na stanowisku pracy.

Jedną z alternatyw dla utrzymania przypadków użycia w dłuższej perspektywie jest idea „analizy jednorazowej[5]", zgodnie z którą przypadek użycia pełni funkcję komunikacyjną bez narzutu związanego z jego rolą jako archiwalnego artefaktu. Jeśli tak jest, wyrzuć przypadek użycia po jego zakodowaniu. Utrzymywanie go nie jest zgodne z Lean.

Nawet dokumentacja użytkownika końcowego ma swoją wartość. Gdy ktoś po raz pierwszy zobaczy iPoda, nie ma jeszcze modelu mentalnego, czym on jest. To jest odtwarzacz MP3, to jest interfejs iTunes. Jeśli nowy klient przechodzi na iPoda z innego odtwarzacza MP3, ważne jest, aby określić interfejs iTunes jako miejsce, gdzie wykonujemy konfigurację i zarządzanie, oraz by podkreślić, że iPod może być znacznie prostszy od starego odtwarzacza MP3. To jest zmiana paradygmatu dla użytkownika końcowego — zmiana, której może nie być w stanie wykonać

[5] Dziękujmy Paulowi S.R. Chisholmowi za ten doskonały termin.

samodzielnie. Proste przewodniki i przykłady mogą pomóc w stworzeniu metafor wspierających dobre doświadczenia użytkowników. Oczywiście, głębsze modele interakcji (*up* oznacza głośniej, natomiast *down* oznacza bardziej miękko) powinny być wbudowane i nie powinny zależeć od dokumentacji, aby stały się przydatne.

7.10. Tło historyczne

Pionierem przypadków użycia był Ivar Jacobson, który opisał je w 1992 roku (Jacobson 1992) i próbował zaoferować branży za pośrednictwem opracowanej przez siebie metodologii — *Objectory*.

Rebecca Wirfs-Brock była jedną z pierwszych osób, które wyszły poza wysokopoziomową graficzną konceptualizację przypadków użycia w kierunku opisywania tego, co faktycznie dzieje się wewnątrz przypadku użycia. W późniejszym czasie stała się pionierem dwukolumnowego opisu rozmów w kontekście aktorów i akcji (Wirfs-Brock 1993). Format zaproponowany przez Rebeccę nie został zaadaptowany ani przez RUP, ani przez Alistaira Cockburna, ale został przyjęty w społeczności specjalistów w dziedzinie użyteczności (na przykład Constantine i Lockwood 1999). Jest on stosowany w praktyce w wielu firmach i jest naszą preferowaną formą w tej książce.

Przypadki użycia są częścią UML od około 1995 roku — od najwcześniejszych dni notacji UML.

Alistair Cockburn na nowo zinterpretował przypadki użycia i zaczął wnosić swoją wizję do świata oprogramowania za pośrednictwem książki *Writing Effective Use Cases* (Cockburn 2001). Wizja Cockburna jest w całkowitej harmonii z wizją Agile, którą promował jako jeden z organizatorów i sygnatariuszy Manifestu Agile. Interpretacja przypadków użycia Rebekki także jest całkowicie zgodna z Agile.

Społeczność XP (którego początki sięgają 1997 roku) uznała przypadki użycia bazujące na UML z lat dziewięćdziesiątych za coś, czego należy unikać, i zaproponowała w zamian stosowanie opowieści użytkownika. Opowieści użytkowników były stosowane wspólnie z innymi zasadami i praktykami XP, takimi jak „klient na miejscu" i rozwój sterowany testami, w celu wspierania szybkiego rozwoju małych systemów. Jednak opowieści użytkowników jako samodzielne, minimalistyczne jednostki miały problemy ze skalowaniem do myślenia systemowego. W złożonych systemach relacje pomiędzy opowieściami były równie ważne jak same historie. Metodycy, w celu uwzględnienia tych aspektów, stopniowo dodawali funkcje do opowieści użytkowników: na przykład Mike Cohn zalecał dodawanie informacji przeznaczonych dla testerów. W czasie kiedy powstawała ta książka, z powodu istnienia indywidualnych odmian definicja opowieści użytkownika była mocno rozmyta. Obecnie nie funkcjonuje pojedyncza, powszechnie akceptowana definicja tego terminu.

Skorowidz

A

ABC, Abstract Base Classes, 132
abstrakcja, 88
abstrakcyjne klasy bazowe, 132, 192
adaptacja cech, 28
Agile, 20
aktor, 45, 221
aktualizacja logiki dziedziny, 261
aktualizowanie odchyleń, 190
algorytm, 114, 211, 230
angażowanie interesariuszy, 68
AOP, Aspect-Oriented Programming, 103, 275
architektura, 20, 31, 51
  atomowych zdarzeń, 152, 196, 206–209, 213
  część dynamiczna, 164
  DCI, 47, 152, 207, 219–276
  forma, 88
  funkcjonalna, 223
  klasyczna, 22
  kompresja, 88
  Lean, 19–22, 87
  MVC, 31, 153
  obiektowa, 153
  oparta na możliwościach, 276
  pojęcia, 179
  problemy użytkowników, 89
  składnik dynamiczny, 89
  składnik statyczny, 89
  SOA, 36
  systemu, 27
  terminy, 179
  ukończona, 156
  up-front, 153
artefakty systemowe, 111
asercje, 136, 139
asercje statyczne, 140, 142
aspekt, 207
atomowe funkcje składowe, 214
autonomia, 97, 125

B

BDD, Behavior-Driven Development, 67, 168
bezmetodowe role obiektów, 232
biznes, 60, 72
budowanie projektu obiektowego, 151
BUFD, 154

C

cechy wspólne, 119, 121
cechy, traits, 225
cel, goal, 84
  sprintu, Sprint goal, 84
  tworzenia architektury, 45
    DCI, 34
    Lean, 51
czarter, charter, 85
czas, 51, 71
części wspólne, 108, 111, 113
czytelność kodu, 228

D

dane, data, 220
dane wejściowe, 229
dane-kontekst-interakcje, 152, 220
DCI, Data-Context-Interaction, 152, 219–276
decyzje, 187
decyzje zakodowane, 171
definicja problemu, 77
  dobra, 79, 82
  dokumentacja, 85
  kompletna, 82
  zła, 79
deklaracje
  procedur, 136
  stałych, 136
  szablonów, 136
deweloperzy, 54, 61, 66, 72
DFD, Data Flow Diagram, 166
DFT, Design For Testability, 67
diagramy przepływu danych, 166
dokumentacja, 32–34, 127
  architektury, 156, 274
  definicji problemu, 85
  komunikacji, 33
  ponadczasowa, 34
  użytkownika końcowego, 199
dokumenty UML, 36
dostarczanie kodu, 264
DSL, Domain-Specific Language, 121
dynamiczne testy architektury, 198
dyrektywa
  extend, 261
  include, 240
działanie obiektów kontekstu, 246
dziedzina, 99
dziedziny rozłączne, 112

E

eksperci dziedzinowi, 54, 64, 72

F

faktoring na poziomie systemu, 29
faktoryzacja, 212, 216
forma, 46, 88, 143, 208, 223
  behawioralna, 95
  dziedzinowa, 95
  skodyfikowana, 123
framework
  dla identyfikatorów, 232
  kontekstu, 241
  podejmowania decyzji, 169
  Qi4j, 297
funkcje, 51
  składowe, 264
  systemu, 27, 159
funkcjonalności, 171

G

generatory aplikacji, 120
gra zespołowa, 178
grupy interesariuszy, 54

I

identyfikatory, 224, 226
identyfikowanie przypadków użycia, 59
IEEE 1471, 20
implementacja cech, 235, 264
implementacja rachunków
  w C#, 287
  w Pythonie, 283
  w Ruby, 291
  w Scali, 279
  w Squeaku, 299
intencje aktora, 184
interakcje, interaction, 91, 220
interesariusze, 41, 49, 68
interesariusze najważniejsi, 54
interfejs, 208
interfejs użytkownika, 180
inżynieria postępowa, forward engineering, 147

J

JAD, Joint Application Design, 39
jednolitość, 125
jedność zespołu, 74
język
  C#, 273
  C++, 235
  Java, 273
  LISP, 65
  Python, 273
  Ruby, 237
  Scala, 272
  Smalltalk, 235, 274
  Squeak, 274
języki
  aplikacyjne, 121
  DSL, 121
  dziedzinowe, 120, 121
  programowania, 27
  wzorców, 123

K

karta CRC, 215, 216
kategorie podobieństw, 114
klasa, 27, 151
  BB1Context, 304
  BB1RoledTrait, 305
klasy
  ABC, 134
  bazowe, 132
  kontekstu, 226
klasyczna architektura oprogramowania, 23
klasyfikacja, 109
klient, 54, 61, 72
kodowanie, 46, 131, 201, 219
  Agile, 203
  C++, 237, 244
  Ruby, 239, 242
komentarz, 184
komponent Controller, 210
kompresja, 88
kompromisy projektowe, 227
komunikacja w projekcie, 136
koncentracja na efektach długoterminowych, 21
konsolidacja, 172, 193
konsolidacja co-system-robi, 172
kontekst, context, 152, 220, 240
konteksty zagnieżdżone, 254
kontroler, 205, 209
koordynacja, 202
koordynowanie obiektów, 212
koszty, 51, 154
kryteria konsolidacji, 193
kultura, 104

L

Lean, 19–22, 87
linia produkcyjna, 73
lista
  odchyleń, 189
  scenariuszy, 58
  własności, feature list, 166
logika dziedzinowa, 261. 263
lokowanie chipów, 98
LSP, Liskov Substitutability Principle, 142

M

makro SELF, 238
marnotrawstwo czasu, 71
menedżer, 60
metody, 27
metody projektowania, 36
metodyka
  Agile, 20–24, 28, 41
  BDD, 67
  Lean, 30
  Scrum, 30, 38
  The Toyota Way, 38
miejsce obiektu kontekstu, 242
minimalizowanie przeróbek, 154
mity Agile, 93
model, 205, 208
  danych, 152
  DCI, 32
  dziedziny, 143, 208
  kaskadowy, 41, 204
  obliczeniowy, 145
  użytkownika końcowego, 110
    mentalny, 44, 180, 221, 265
    poznawczy, 57
moduł, 109
moduły zorganizowane według
  dziedziny biznesu, 102
  dziedziny rozwiązania, 102
  rynków, 102
MVC, Model-View-Controller, 31, 117
MVC-U, 205
myślenie, 29, 35

N

nakłady na architekturę, 154
narracje, 167
narzędzia
  analizy, 215
  projektowania, 215

narzędzie
doxygen, 217
Javadoc, 217
niewiadome, 169
nowe wymagania, 186, 187

O

obiekt, 151
kontekstu, 220, 240, 246, 265
typu jeden do wielu, 212
obiekty
DCI, 226
dziedziny, 208
kontekstowo-dziedzinowe, 268
obsługa
odchyleń, 189
przypadków użycia, 135
oczekiwania docelowych użytkowników, 58
odchylenia, 187, 194
odchylenia przypadku użycia, 185
oddzielanie kodowania, 218
odmiana, variation, 107
odpowiedzialność systemu, 184, 192
odraczanie decyzji, 30
odwzorowanie
organizacji, 104
roli obiektu, 227
scenariusza, 232
określanie zakresu, 177
opieka, 35
opis strukturalny przypadku użycia, 178
opowieści użytkowników, 160
oprogramowanie dostawców zewnętrznych, 124
optymalizowanie przychodów, 53
organizacja, 102, 104
osoba, 164, 167
osoby związane z biznesem, 54

P

paradygmat, 124
Model-View-Controller, 203
obiektowy, 80
paradygmaty projektowania, 97
parametry zmienności, 120
perspektywa
danych, 301
interakcji, 303
kontekstu, 302
testowania, 300
wsparcia, 304
plan wydań, 183
planowanie z góry, 21
podejmowanie decyzji, 168, 169
podstawowe działania, 41
podsystem, 96
podział, 106
algorytmów, 234
drugi
prawo Conwaya, 96
na części, 94
na moduły, 110
pierwszy
forma, 95
systemu, 98
wyjątkowy, 99
pojęcia, 179
poka-yoke, 90
polimorfizm, 227
praktyki TPM, 39
prawo
ciągłości, 148
Conwaya, 91, 96, 105
problem, 77, 81
proces definiowania problemów, 82
produkcja Agile, 24, 41
profile użytkowników, 164
program Agile, 203
programowanie
aspektowe, 275
obiektowe, 116, 151, 196, 215
piśmienne, 33, 217
XP, 45
projekt, 46
projektant interfejsu użytkownika, 180
projektowanie, 77
greenfield, 28
JAD, 39
sterowane zachowaniami, 167
prototypy, 168
przekształcanie przypadków użycia, 47
przenoszenie architektury, 128
przepływ danych, 170
przerabianie, 29
przerost funkcjonalności, creeping featurism, 84
przeróbki, 53, 91
przydatność, 126
przydomek, nickname, 127
przydomki informacyjne, information nicknames, 127
przypadki użycia, 27, 45, 57, 161, 172–176, 183, 190, 196, 199, 230
przyrostowe
dodawanie odchyleń, 187
uaktualnienia, 178
punkt widzenia
programisty, 174
użytkownika, 174
punkty testowe, 146

Q

Qi4j, 297

R

refaktoryzacja, 212, 216
reguły biznesowe, 182
rejestr wymagań, product backlog, 53
relacje, 149
relacje pomiędzy MVC i DCI, 222
rodzaje komunikacji, 32
ROI, 50, 65
role, 27, 151, 152, 191
obiektów, 164, 191, 208, 212, 220, 233
obiektów bezmetodowe, 222
obiektów z metodami, 234, 254, 270
użytkowników, 165, 208
rozmiar architektury, 155
rozszerzenie języka programowania, 145
rozwiązanie, 81
rozwiązanie ograniczone, 80
rozwiązywanie problemów użytkowników, 89
rozwijanie metod, 212
rój, 73
różnice, 108, 111, 119, 121
różnice pomiędzy Lean i Agile, 29
rusztowanie oprogramowania, 144

S

samoorganizacja, 78
scena, 177
scenariusz, 58, 167
przypadków użycia, 27
słonecznego dnia, 184
z komentarzami, 181
Scrum, 29, 37
sekret Lean, 52, 67
sieć interesariuszy, 71
skalowanie algorytmów, 142
słownik dziedziny, 127, 128
SOA, Service-Oriented Architecture, 36

specjalizacja, 29
specyfikacja stwarzająca możliwości, 163
Sprint, 29
sprzężenie zwrotne, 51
stan języków DSL, 121
stopa zwrotu z inwestycji, 50
struktura, 88, 106, 187, 223
  biznesu, 100
  danych, 114
  organizacyjna, 100
  systemu, 44
strumień wartości, 49, 51, 63
stwarzanie możliwości, 163
styl, 118
styl projektu, 105
  części stałe, 107
  części zmienne, 107
style architektury, 262
system, 87, 131
  co robi, 42, 46, 159, 168
  czym jest, 43
szkic kodu, 131
szybkie rozwiązania, 75

T

TDD, Test Driven Development, 93
technologia CORBA, 147
terminy, 179
tester, 66, 72
testowanie
  architektury, 146
  relacji, 150
  systemu, 67
  użyteczności, 146, 197
  własności, 58
testy, 46, 47
TPM, Total Productive Maintenance, 39
tworzenie
  klas, 213
  przypadków użycia, 176
  ról, 47
  wzorców, 33
typy, 114
typy relacji, 149

U

ukrywanie informacji, 260
umiejscowienie algorytmu, 270
UML, 36
usługi, 143
uzyskanie danych wejściowych, 195
użyteczności, 31
użytkownik, 164
użytkownik końcowy, 50, 54, 59, 72
używanie klas, 270

W

warstwa obiektów kontekstu, 153
warstwy kontekstów, 259
warunki końcowe, 136
warunki wstępne, 136
widok, 205
  helikoptera, 182
  systemu, 41
wiedza dziedzinowa, 100
wizja, vision, 84
własności
  języka C#, 121
  języka C++, 119
własność problemu, 83
  Scrum, 83
właściwości
  dynamiczne, 142
  statyczne, 142
wnioskowanie, 228
wsparcie
  dla nowych wymagań, 194
  dla podejmowania decyzji, 194
  dla testów, 193
  dla wydań, 194
  przepływu pracy, 193
wstrzykiwanie
  algorytmów, 225
  ról obiektowych, 260
wybór stylu projektu, 105
wykonywanie, 29
wykorzystanie
  oprogramowania, 166, 169
  przypadków użycia, 161
  specyfikacji, 161
wymagania, 31, 162, 171
wymiary złożoności, 101
wyznaczanie ról, 191
wzorce
  niskopoziomowe, 145
  organizacyjne, 101
  projektowe, 201
wzorzec
  CRTP, 264
  MVC, 117, 201, 204
  MVC-U, 202

Y

YAGNI, 153

Z

zaangażowanie
  interesariuszy, 31, 41, 49
  klienta, 70
zachowania ról obiektowych, 238
zachowanie, 223
zakres, 111
  przypadków użycia, 177
  systemów Agile, 25
zamierzenie, objective, 84
zarządzanie wymaganiami, 31
zasady Lean, 29, 271
zastępowalność Liskova, 142
zdarzenia atomowe, 218
zdefiniowanie problemu, 43
zdrowy rozsądek, 35
zespół, 65, 69, 74
złoty środek, 155
złożoność podziału systemu, 98
zmniejszanie kosztów, 53
zrozumienie przedmiotu pracy, 195
zwyczaje, 175, 231, 254

ARCHITEKTURA LEAN W PROJEKTACH AGILE

Tempo rozwoju aplikacji wymusza stosowanie elastycznych sposobów wytwarzania oprogramowania. Książka ta została poświęcona architekturze Lean, która usprawni ten proces dzięki nowatorskiemu podejściu. Wykorzystaj ją i przygotuj swoją aplikację na zmiany funkcjonalne, by użytkownicy mogli w pełni wykorzystać jej potencjał!

W trakcie lektury zapoznasz się z duchem Agile i Lean oraz przydzielisz najważniejsze role członkom projektu. Po tym niezwykle interesującym wstępie rozpoczniesz pasjonującą podróż po świecie architektury Lean. Dowiesz się, czym jest system, jak podzielić projekt na części i wybrać jego styl. Na podstawie kolejnych rozdziałów zorganizujesz swój kod i przetestujesz zaprojektowaną architekturę. Znajdziesz tu wiele przykładów, które w najlepszy sposób przedstawiają założenia i intencje architektury Lean, z dużym naciskiem na sam kod. To obowiązkowa lektura dla wszystkich programistów i projektantów systemów informatycznych.

Dzięki tej książce:

- **poznasz filozofię Agile i Lean**
- **zbudujesz kod odporny na zmiany**
- **zrozumiesz paradygmat DCI**
- **poznasz współczesne metody wytwarzania oprogramowania!**

Twój przewodnik po architekturze Lean!

ISBN: 978-83-246-8672-8

9 788324 686728

Cena: 59,00 zł