

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#

Autor: Robert C. Martin, Micah Martin

Tłumaczenie: Mikołaj Szczepaniak

ISBN: 978-83-246-1177-5

Tytuł oryginału: [Agile Principles, Patterns,
and Practices in C#](#)

Format: B5, stron: 848



Poznaj nowoczesną metodykę wytwarzania oprogramowania w C#

- Jak stosować w praktyce zasady zwinnego wytwarzania oprogramowania?
- W jaki sposób wykorzystywać w projekcie diagramy UML?
- Jak korzystać z wzorców projektowych?

W związku ze stale rosnącymi oczekiwaniami użytkowników oprogramowania produkcja systemów informatycznych wymaga dziś korzystania z usystematyzowanych metod zarządzania. Projekt informatyczny, przy którym nie używa się sensownej metodologii wytwarzania, jest skazany na porażkę – przekroczenie terminu, budżetu i niespełnienie wymagań funkcjonalnych. Kierowanie projektem zgodnie z określonymi zasadami również nie gwarantuje sukcesu, lecz znacznie ułatwia jego osiągnięcie. Na początku roku 2001 grupa ekspertów zawiązała zespół o nazwie Agile Alliance. Efektem prac tego zespołu jest metodologia zwinnego wytwarzania oprogramowania – Agile.

Książka „Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#” to podręcznik metodologii Agile przeznaczony dla twórców oprogramowania korzystających z technologii .NET. Dzięki niemu poznasz podstawowe założenia i postulaty twórców Agile i nauczysz się stosować je w praktyce. Dowiesz się, jak szacować terminy i koszty, dzielić proces wytwarzania na iteracje i testować produkt. Zdobędziesz wiedzę na temat refaktoryzacji, diagramów UML, testów jednostkowych i wzorców projektowych. Przeczytasz także o publikowaniu kolejnych wersji oprogramowania.

- Techniki programowania ekstremalnego
- Planowanie projektu
- Testowanie i refaktoryzacja
- Zasady zwinnego programowania
- Modelowanie oprogramowania za pomocą diagramów UML
- Stosowanie wzorców projektowych
- Projektowanie pakietów i komponentów

**Przekonaj się, ile czasu i pracy zaoszczędzisz,
stosując w projektach metodologię Agile**



Spis treści

Słowo wstępne	17
Przedmowa	21
Podziękowania	31
O autorach	33
Część I Wytwarzanie zwinne	35
Rozdział 1. Praktyki programowania zwinnego	37
Agile Alliance	38
Programiści i ich harmonijna współpraca jest ważniejsza od procesów i narzędzi	39
Działające oprogramowanie jest ważniejsze od wyczerpującej dokumentacji	40
Faktyczna współpraca z klientem jest ważniejsza od negocjacji zasad kontraktu	41
Reagowanie na zmiany jest ważniejsze od konsekwentnego realizowania planu	42
Podstawowe zasady	43
Konkluzja	46
Bibliografia	47
Rozdział 2. Przegląd technik programowania ekstremalnego	49
Praktyki programowania ekstremalnego	50
Cały zespół	50
Opowieści użytkownika	50

Krótkie cykle	51
Testy akceptacyjne	52
Programowanie w parach	53
Wytwarzanie sterowane testami (TDD)	54
Wspólna własność	54
Ciągła integracja	55
Równe tempo	56
Otwarta przestrzeń pracy	56
Gra planistyczna	57
Prosty projekt	57
Refaktoryzacja	59
Metafora	59
Konkluzja	61
Bibliografia	61
Rozdział 3. Planowanie	63
Wstępne poznawanie wymagań	64
Dzielenie i scalanie opowieści użytkownika	65
Planowanie wydań	66
Planowanie iteracji	66
Definiowanie warunków zakończenia projektu	67
Planowanie zadań	67
Iteracje	69
Śledzenie postępu	69
Konkluzja	70
Bibliografia	71
Rozdział 4. Testowanie	73
Wytwarzanie sterowane testami	74
Przykład projektu poprzedzonego testami	75
Izolacja testów	76
Eliminowanie powiązań	78
Testy akceptacyjne	79
Wpływ testów akceptacyjnych na architekturę oprogramowania	81

Konkluzja	82
Bibliografia	82
Rozdział 5. Refaktoryzacja	83
Prosty przykład refaktoryzacji	
— generowanie liczb pierwszych	84
Testy jednostkowe	86
Refaktoryzacja	87
Ostatnie udoskonalenia	93
Konkluzja	97
Bibliografia	98
Rozdział 6. Epizod z życia programistów	99
Gra w kręgle	100
Konkluzja	146
Przegląd reguł gry w kręgle	147
Część II Projektowanie zwinne	149
Rozdział 7. Czym jest projektowanie zwinne?	153
Symptomy złego projektu	154
Symptomy złego projektu, czyli potencjalne źródła porażek	154
Szywność	155
Wrażliwość	155
Nieelastyczność	156
Niedostosowanie do rzeczywistości	156
Nadmierna złożoność	156
Niepotrzebne powtórzenia	157
Nieprzejrzystość	157
Dlaczego oprogramowanie ulega degradacji	158
Program Copy	159
Przykład typowego scenariusza	159
Przykład budowy programu Copy w ramach projektu zwinnego	163
Konkluzja	166
Bibliografia	166

Rozdział 8. Zasada pojedynczej odpowiedzialności	167
Definiowanie odpowiedzialności	170
Oddzielanie wzajemnie powiązanych odpowiedzialności	171
Trwałość	171
Konkluzja	172
Bibliografia	172
Rozdział 9. Zasada otwarte-zamknięte	173
Omówienie zasady otwarte-zamknięte	174
Aplikacja Shape	177
Przykład naruszenia zasady OCP	177
Przykład pełnej zgodności z zasadą otwarte-zamknięte	180
Przewidywanie zmian i „naturalna” struktura	181
Przygotowywanie punktów zaczepienia	182
Stosowanie abstrakcji do jawnego zamykania oprogramowania dla zmian	184
Zapewnianie zamknięcia z wykorzystaniem techniki sterowania przez dane	185
Konkluzja	187
Bibliografia	187
Rozdział 10. Zasada podstawiania Liskov	189
Naruszenia zasady podstawiania Liskov	190
Prosty przykład	190
Przykład mniej jaskrawego naruszenia zasady LSP	192
Przykład zaczerpnięty z rzeczywistości	199
Wyodrębnianie zamiast dziedziczenia	205
Heurystyki i konwencje	208
Konkluzja	209
Bibliografia	209
Rozdział 11. Zasada odwracania zależności	211
Podział na warstwy	212
Odwracanie relacji własności	213
Zależność od abstrakcji	215

Prosty przykład praktycznego znaczenia zasady DIP	216
Odkrywanie niezbędnych abstrakcji	217
Przykład aplikacji Furnace	219
Konkluzja	221
Bibliografia	221
Rozdział 12. Zasada segregacji interfejsów	223
Zanieczyszczanie interfejsów	223
Odrębne klasy klienckie oznaczają odrębne interfejsy	225
Interfejsy klas kontra interfejsy obiektów	227
Separacja przez delegację	227
Separacja przez wielokrotne dziedziczenie	229
Przykład interfejsu użytkownika bankomatu	230
Konkluzja	237
Bibliografia	237
Rozdział 13. Przegląd języka UML	
dla programistów C#	239
Diagramy klas	243
Diagramy obiektów	244
Diagramy sekwencji	245
Diagramy współpracy	246
Diagramy stanów	246
Konkluzja	247
Bibliografia	247
Rozdział 14. Praca z diagramami	249
Po co modelować oprogramowanie?	249
Po co budować modele oprogramowania?	250
Czy powinniśmy pracować nad rozbudowanymi projektami przed przystąpieniem do kodowania?	251
Efektywne korzystanie z diagramów języka UML	251
Komunikacja z innymi programistami	252
Mapy drogowe	253
Dokumentacja wewnętrzna	255
Co powinniśmy zachowywać, a co można wyrzucać do kosza?	255

Iteracyjne udoskonalanie	257
Najpierw zachowania	257
Weryfikacja struktury	259
Wyobrażenie o kodzie	261
Ewolucja diagramów	262
Kiedy i jak rysować diagramy	264
Kiedy przystępować do tworzenia diagramów, a kiedy rezygnować z dalszego rysowania ich	264
Narzędzia CASE	265
A co z dokumentacją?	266
Konkluzja	267
Rozdział 15. Diagramy stanów	269
Wprowadzenie	270
Zdarzenia specjalne	271
Superstany	272
Pseudostan początkowy i końcowy	274
Stosowanie diagramów skończonych maszyn stanów	274
Konkluzja	276
Rozdział 16. Diagramy obiektów	277
Migawka	278
Obiekty aktywne	279
Konkluzja	283
Rozdział 17. Przypadki użycia	285
Pisanie przypadków użycia	286
Przebiegi alternatywne	287
Co jeszcze?	288
Prezentowanie przypadków użycia na diagramach	288
Konkluzja	290
Bibliografia	290
Rozdział 18. Diagramy sekwencji	291
Wprowadzenie	292
Obiekty, linie życia, komunikaty i inne konstrukcje	292
Tworzenie i niszczenie obiektów	293

Proste pętle	295
Przypadki i scenariusze	295
Pojęcia zaawansowane	298
Pętle i warunki	298
Komunikaty, których przesyłanie wymaga czasu	300
Komunikaty asynchroniczne	302
Wiele wątków	307
Obiekty aktywne	308
Wysyłanie komunikatów do interfejsów	309
Konkluzja	310
Rozdział 19. Diagramy klas	311
Wprowadzenie	312
Klasy	312
Asocjacje	313
Relacje dziedziczenia	314
Przykładowy diagram klas	315
Omówienie szczegółowe	318
Stereotypy klas	318
Klasy abstrakcyjne	319
Właściwości	320
Agregacja	321
Kompozycja	322
Liczebność	323
Stereotypy asocjacji	324
Klasy zagnieżdżone	326
Klasy asocjacji	326
Kwalifikatory asocjacji	327
Konkluzja	328
Bibliografia	328
Rozdział 20. Heurystyki i kawa	329
Ekspres do kawy Mark IV Special	330
Specyfikacja	330
Popularne, ale niewłaściwe rozwiązanie	333
Nieprzemysłana abstrakcja	336
Poprawione rozwiązanie	337

Implementacja modelu abstrakcyjnego	343
Zalety projektu w tej formie	358
Implementacja projektu obiektowego	366
Bibliografia	366
Część III Studium przypadku listy płac	367
Uproszczona specyfikacja systemu listy płac	368
Ćwiczenie	369
Przypadek użycia nr 1 — dodanie danych nowego pracownika	369
Przypadek użycia nr 2 — usunięcie danych pracownika	370
Przypadek użycia nr 3 — wysłanie karty czasu pracy	370
Przypadek użycia nr 4 — wysłanie raportu o sprzedaży	370
Przypadek użycia nr 5 — wysłanie informacji o opłacie na rzecz związku zawodowego	371
Przypadek użycia nr 6 — zmiana szczegółowych danych pracownika	371
Przypadek użycia nr 7 — wygenerowanie listy płatności na dany dzień ..	372
Rozdział 21. Wzorce projektowe Command	
i Active Object — uniwersalność	
i wielozadaniowość	373
Proste polecenia	374
Transakcje	377
Fizyczny podział kodu	378
Czasowy podział kodu	379
Metoda Undo	379
Wzorzec projektowy Active Object	380
Konkluzja	386
Bibliografia	386
Rozdział 22. Wzorce projektowe Template Method	
i Strategy — dziedziczenie	
kontra delegacja	387
Wzorzec projektowy Template Method	388
Błędne zastosowanie wzorca Template Method	392
Sortowanie bąbelkowe	392

Wzorzec projektowy Strategy	396
Konkluzja	402
Bibliografia	402
Rozdział 23. Wzorce projektowe Facade i Mediator	403
Wzorzec projektowy Facade	404
Wzorzec projektowy Mediator	405
Konkluzja	407
Bibliografia	408
Rozdział 24. Wzorce projektowe	
Singleton i Monostate	409
Wzorzec projektowy Singleton	410
Zalety	412
Wady	412
Wzorzec Singleton w praktyce	413
Wzorzec projektowy Monostate	415
Zalety	417
Wady	417
Wzorzec Monostate w praktyce	417
Konkluzja	423
Bibliografia	423
Rozdział 25. Wzorzec projektowy Null Object	425
Omówienie	425
Konkluzja	429
Bibliografia	429
Rozdział 26. Przypadek użycia listy płac	
— pierwsza iteracja	431
Uproszczona specyfikacja	432
Analiza przez omówienie przypadku użycia	433
Dodanie danych nowego pracownika	434
Usunięcie danych pracownika	436
Wysłanie karty czasu pracy	436
Wysłanie raportu o sprzedaży	437

Wysłanie informacji o opłacie na rzecz związku zawodowego	438
Zmiana szczegółowych danych pracownika	439
Wygenerowanie listy płac na dany dzień	441
Refleksja — identyfikacja abstrakcji	443
Wynagrodzenia wypłacane pracownikom	444
Harmonogram wypłat	444
Formy wypłat	446
Przynależność związkowa	446
Konkluzja	447
Bibliografia	447

Rozdział 27. Przypadek użycia listy płac

— implementacja	449
Transakcje	450
Dodawanie danych pracowników	450
Usuwanie danych pracowników	456
Karty czasu pracy, raporty o sprzedaży i składki na związki zawodowe	458
Zmiana danych pracowników	466
Co ja najlepszego zrobiłem?	477
Wynagradzanie pracowników	480
Wynagradzanie pracowników etatowych	483
Wynagradzanie pracowników zatrudnionych w systemie godzinowym	486
Program główny	498
Baza danych	499
Konkluzja	500
O tym rozdziale	501
Bibliografia	502

Część IV Pakowanie systemu płacowego

Rozdział 28. Zasady projektowania pakietów

i komponentów	505
Pakiety i komponenty	506
Zasady spójności komponentów — ziarnistość	507

Zasada równoważności wielokrotnego użycia i wydawania (REP)	507
Zasada zbiorowego wielokrotnego stosowania (CRP)	509
Zasada zbiorowego zamykania (CCP)	510
Podsumowanie problemu spójności komponentów	511
Zasady spójności komponentów — stabilność	511
Zasada acyklicznych zależności (ADP)	511
Zasada stabilnych zależności (SDP)	519
Zasada stabilnych abstrakcji (SAP)	525
Konkluzja	530
Rozdział 29. Wzorzec projektowy Factory	531
Problem zależności	534
Statyczna kontra dynamiczna kontrola typów	536
Fabryki wymienne	537
Wykorzystywanie fabryk do celów testowych	538
Znaczenie fabryk	540
Konkluzja	540
Bibliografia	540
Rozdział 30. Studium przypadku systemu płacowego	
— analiza podziału na pakiety	541
Notacja i struktura komponentów	542
Stosowanie zasady zbiorowego zamykania (CCP)	544
Stosowanie zasady równoważności wielokrotnego użycia i wydawania (REP)	546
Wiązanie komponentów i hermetyzacja	549
Mierniki	551
Stosowanie mierników dla aplikacji płacowej	553
Fabryki obiektów	556
Przebudowa granic spójności	558
Ostateczna struktura pakietów	559
Konkluzja	561
Bibliografia	561

Rozdział 31. Wzorzec projektowy Composite	563
Polecenia kompozytowe	565
Liczność albo brak licznosci	566
Konkluzja	566
Rozdział 32. Wzorzec projektowy Observer	
— ewolucja kodu w kierunku wzorca	567
Zegar cyfrowy	568
Wzorzec projektowy Observer	589
Modele	590
Zarządzanie zasadami projektowania obiektowego	591
Konkluzja	592
Bibliografia	593
Rozdział 33. Wzorce projektowe Abstract Server,	
Adapter i Bridge	595
Wzorzec projektowy Abstract Server	596
Wzorzec projektowy Adapter	598
Forma klasowa wzorca Adapter	599
Problem modemu — adaptery i zasada LSP	599
Wzorzec projektowy Bridge	604
Konkluzja	607
Bibliografia	607
Rozdział 34. Wzorce projektowe Proxy i Gateway	
— zarządzanie cudzymi interfejsami API	609
Wzorzec projektowy Proxy	610
Implementacja wzorca Proxy	615
Podsumowanie	630
Bazy danych, oprogramowanie pośredniczące	
i inne gotowe interfejsy	631
Wzorzec projektowy Table Data Gateway	634
Testowanie konstrukcji TDG w pamięci	642
Test bram DB	643

Stosowanie pozostałych wzorców projektowych łącznie z bazami danych	646
Konkluzja	648
Bibliografia	648
Rozdział 35. Wzorzec projektowy Visitor	649
Wzorzec projektowy Visitor	650
Wzorzec projektowy Acyclic Visitor	654
Zastosowania wzorca Visitor	660
Wzorzec projektowy Decorator	668
Wzorzec projektowy Extension Object	674
Konkluzja	686
Bibliografia	686
Rozdział 36. Wzorzec projektowy State	687
Zagnieżdżone wyrażenia switch-case	688
Wewnętrzny zasięg zmiennej stanu	691
Testowanie akcji	692
Zalety i wady	692
Tabele przejść	693
Interpretacja tabeli przejść	694
Zalety i wady	695
Wzorzec projektowy State	696
Wzorzec State kontra wzorzec Strategy	699
Zalety i wady	699
Kompilator maszyny stanów (SMC)	700
Plik Turnstile.cs wygenerowany przez kompilator SMC i pozostałe pliki pomocnicze	703
Zastosowania skończonej maszyny stanów	709
Wysokopoziomowa polityka działania graficznych interfejsów użytkownika (GUI)	709
Sterowanie interakcją z interfejsem GUI	711
Przetwarzanie rozproszone	712
Konkluzja	713
Bibliografia	714

Rozdział 37. Studium przypadku systemu płacowego	
— baza danych	715
Budowa bazy danych	716
Słaby punkt dotychczasowego projektu	716
Dodawanie danych nowych pracowników	719
Transakcje	732
Odczytywanie danych o pracownikach	738
Co jeszcze zostało do zrobienia?	753
Rozdział 38. Interfejs użytkownika systemu płacowego	
— wzorzec Model View Presenter	755
Interfejs	758
Implementacja	759
Budowa okna	770
Okno główne systemu płacowego	778
Pierwsza odsłona	791
Konkluzja	792
Bibliografia	792
Dodatek A Satyra na dwa przedsiębiorstwa	793
Rufus Inc. — Project Kickoff	793
Rupert Industries — Project Alpha	793
Dodatek B Czym jest oprogramowanie?	811
Skorowidz	827

Praktyki programowania zwinnego



Kogut na wieży kościelnej, choć wykuty z żelaza, szybko zostałby strącony, gdyby choć na moment zapomniał o powinności ulegania najmniejszemu powiewom.

— Heinrich Heine

Wielu z nas przeżyło koszmar pracy nad projektem, dla którego nie istniał zbiór sprawdzonych praktyk, jakimi można by się kierować. Brak efektywnych praktyk prowadzi do nieprzewidywalności, wielokrotnych błędów i niepotrzebnie podejmowanych działań.

Klienci są niezadowoleni z niedotrzymywanych terminów, rosnących kosztów i kiepskiej jakości. Sami programiści są zniechęceni ciągłymi nadgodzinami i świadomością niskiej jakości budowanego oprogramowania.

Po doświadczeniu podobnej sytuacji stajemy się ostrożniejsi, aby podobne fiasko nigdy nie stało się ponownie naszym udziałem. Tego rodzaju obawy często okazują się dobrą motywacją do definiowania procesów determinujących kolejne działania (włącznie z ich wynikami i produktami pośrednimi). Ograniczenia i oczekiwania względem poszczególnych etapów tych procesów definiujemy na podstawie dotychczasowych doświadczeń — z reguły decydujemy się na rozwiązania, które zdawały egzamin w przeszłości. Każdy taki wybór wiąże się z przeświadczeniem, że odpowiednie rozwiązania okażą się skuteczne raz jeszcze.

Projekty nie są jednak na tyle proste, by zaledwie kilka ograniczeń i gotowych komponentów mogło nas uchronić przed błędami. Musimy się liczyć z dalszym występowaniem niedociągnięć, których diagnozowanie skłania nas do definiowania kolejnych ograniczeń i oczekiwanych produktów końcowych, jakie w założeniu mają nas uchronić przed podobnymi błędami w przyszłości. Po wielu realizowanych w ten sposób projektach może się okazać, że nasze metody działania są skomplikowane, niewygodne i w praktyce uniemożliwiają efektywną realizację projektów.

Skomplikowany i nieusprawniający pracy proces może prowadzić do wielu problemów, które za pomocą tych metod chcieliśmy wyeliminować. Skutki stosowania takich procesów mogą obejmować opóźnienia czasowe i przekroczenia zakładanych wydatków budżetowych. Rozmaite niejasności w ramach tych procesów mogą rozmywać odpowiedzialność zespołu i — tym samym — prowadzić do powstawania niedopracowanych produktów. Co ciekawe, opisywane zjawiska skłaniają wiele zespołów do przekonania, że źródłem problemów jest niedostateczna liczba procesów. W takim przypadku odpowiedzialnością na negatywne skutki nadmiernej liczby przesadnie rozrośniętych procesów jest pogłębianie tych zjawisk.

Nadmierne rozbudowywanie procesów było powszechnym zjawiskiem w wielu firmach budujących oprogramowanie koło 2000 roku. Mimo że jeszcze w 2000 roku wiele zespołów tworzyło oprogramowanie bez procesów, popularyzacja bardzo skomplikowanych procesów postępowała w błyskawicznym tempie (szczególnie w największych korporacjach).

Agile Alliance

Grupa ekspertów zaniepokojonych obserwowanymi zjawiskami (polegającymi między innymi na wpędzaniu się wielu korporacji w poważne kłopoty wskutek rozrastających się procesów) zorganizowała na początku 2001 roku spotkanie, na którym powstało zrzeszenie nazwane Agile Alliance. Celem tej grupy była popularyzacja wartości i reguł, które skłonią zespoły programistyczne do szybkiego i elastycznego wytwarzania oprogramowa-

nia. Przez kilka miesięcy grupie Agile Alliance udało się opracować najważniejsze założenia. Efektem tej pracy był dokument zatytułowany *The Manifesto of the Agile Alliance*:

Manifest zwinnego wytwarzania oprogramowania

Próbujemy odkrywać lepsze sposoby wytwarzania oprogramowania, realizując własne eksperymenty w tym obszarze i zachęcając do podobnych działań innych programistów. Wskutek tych eksperymentów udało nam się sformułować następujące wskazówki:

Programiści i ich harmonijna współpraca jest ważniejsza od procesów i narzędzi.

Działające oprogramowanie jest ważniejsze od wyczerpującej dokumentacji.

Faktyczna współpraca z klientem jest ważniejsza od negocjacji zasad kontraktu.

Reagowanie na zmiany jest ważniejsze od konsekwentnego realizowania planu.

Oznacza to, że chociaż rozwiązania wymienione po prawej stronie mają pewną wartość, z punktu widzenia skuteczności projektów dużo bardziej wartościowe są działania opisane z lewej strony.

Kent Beck	Mike Beedle	Arie van Bennekum	Alistair Cockburn
Ward Cunningham	Martin Fowler	James Grenning	Jim Highsmith
Andrew Hunt	Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber	Jeff Sutherland
Dave Thomas			

Programiści i ich harmonijna współpraca jest ważniejsza od procesów i narzędzi

Ludzie zawsze są najważniejszym czynnikiem decydującym o sukcesie. Żaden, nawet najlepszy proces nie zapobiegnie porażce projektu, jeśli nie mamy do dyspozycji zespołu, któremu możemy zaufać. Co więcej, zły proces może spowodować, że nawet najlepszy zespół będzie pracował nieefektywnie. Okazuje się, że grupa doskonałych programistów może ponieść dotkliwą porażkę, jeśli nie stworzy prawdziwego zespołu.

Dobry członek zespołu wcale nie musi być genialnym programistą. Dobry pracownik może być przeciętnym programistą, ale musi posiadać zdolność współpracy z pozostałymi członkami swojego zespołu. Harmonijna współpraca z innymi pracownikami i umiejętność komunikacji jest ważniejsza od talentów czysto programistycznych. Prawdopodobieństwo odniesienia sukcesu przez grupę przeciętnych programistów, którzy dobrze ze sobą współpracują, jest większe niż w przypadku zespołu złożonego z samych geniuszy programowania niepotrafiących się porozumieć.

Dobrze dobrane narzędzia mogą mieć istotny wpływ na ostateczny efekt realizacji projektu. Kompilatory, interaktywne środowiska wytwarzania (IDE), systemy kontroli wersji kodu źródłowego itp. są kluczowymi narzędziami decydującymi o funkcjonowaniu

zespołu programistów. Z drugiej strony, nie należy przeceniać roli samych narzędzi. Nadmierna wiara w siłę narzędzi prowadzi do równie fatalnych skutków co ich brak.

Zaleca się stosowanie z początku stosunkowo prostych narzędzi. Nie należy zakładać, że bardziej rozbudowane i zaawansowane narzędzia będą lepsze od prostych, do czasu weryfikacji ich przydatności w praktyce. Zamiast kupować najlepsze i niewyobrażalnie drogie systemy kontroli wersji kodu źródłowego, warto poszukać darmowych narzędzi tego typu i korzystać z nich do czasu, aż wyczerpią się oferowane przez nie możliwości. Zanim kupimy najdroższy pakiet narzędzi CASE (od ang. *Computer-Aided Software Engineering*), powinniśmy skorzystać ze zwykłych tablic i kartek papieru — decyzję o zakupie bardziej zaawansowanych rozwiązań możemy podjąć dopiero wtedy, gdy tradycyjne metody okażą się niewystarczające. Przed podjęciem decyzji o zakupie niezwykle drogiego i zaawansowanego systemu zarządzania bazą danych powinniśmy sprawdzić, czy do naszych celów nie wystarczą zwykłe pliki. Nigdy nie powinniśmy zakładać, że większe i lepsze narzędzia muszą w większym stopniu odpowiadać naszym potrzebom. Wiele z nich bardziej utrudnia, niż ułatwia pracę.

Warto pamiętać, że budowa zespołu jest ważniejsza od budowy środowiska. Wiele zespołów i wielu menedżerów popełnia błąd polegający na budowie w pierwszej kolejności środowiska w nadziei, że wokół niego uda się następnie zebrać zgrany zespół. Dużo lepszym rozwiązaniem jest skoncentrowanie się na budowie zespołu i umożliwienie mu organizacji środowiska odpowiadającego jego potrzebom.

Działające oprogramowanie jest ważniejsze od wyczerpującej dokumentacji

Oprogramowanie bez dokumentacji jest prawdziwą katastrofą. Kod źródłowy nie jest jednak idealnym medium komunikowania przyczyn stosowania poszczególnych rozwiązań i struktury systemu. Lepszym sposobem jest uzasadnianie podejmowanych przez zespół programistów decyzji projektowych w formie zwykłych dokumentów.

Z drugiej strony, zbyt duża liczba dokumentów jest gorsza od zbyt skromnej dokumentacji. Wytwarzanie wielkich dokumentów opisujących oprogramowanie wymaga dużych nakładów czasowych, a w skrajnych przypadkach uniemożliwia synchronizację dokumentacji z kodem źródłowym. Dokumentacja niezgodna z faktycznym stanem kodu staje się wielkim, złożonym zbiorem kłamstw, które prowadzą tylko do nieporozumień.

Niezależnie od realizowanego projektu dobrym rozwiązaniem jest pisanie i utrzymywanie przez zespół programistów krótkiego dokumentu uzasadniającego podjęte decyzje projektowe i opisującego strukturę budowanego systemu. Taki dokument powinien być możliwie **krótki** i dość ogólny — nie powinien zajmować więcej niż dwadzieścia stron i powinien dotyczyć **najważniejszych** decyzji projektowych oraz opisywać strukturę systemu na najwyższym poziomie.

Skoro dysponujemy jedynie krótkim dokumentem uzasadniającym decyzje projektowe i strukturę budowanego systemu, jak będziemy przygotowywać do pracy nad tym systemem nowych członków zespołu? Taki trening nie powinien polegać na przeka-

zaniu dokumentów, tylko na żmudnej pracy z nowym programistą. Transfer wiedzy o systemie musi polegać na wielogodzinnej pomocy nowemu współpracownikowi. Tylko blisko współpracując z programistą, możemy z niego szybko uczynić pełnowartościowego członka naszego zespołu.

Najlepszymi mediami umożliwiającymi przekazanie informacji niezbędnych do pracy nowych członków zespołu jest kod źródłowy i sam zespół. Kod nigdy nie jest sprzeczny ze swoim faktycznym działaniem. Z drugiej strony, mimo że precyzyjne określenie funkcjonowania systemu na podstawie samego kodu źródłowego może być trudne, właśnie kod jest jedynym jednoznacznym źródłem informacji. Zespół programistów utrzymuje swoistą mapę drogową stale modyfikowanego systemu w głowach swoich członków. Oznacza to, że najszybszym i najbardziej efektywnym sposobem prezentacji założeń systemu osobom spoza zespołu jest przelanie tej mapy na papier.

Wiele zespołów wpadło w pułapkę pogoni za doskonałą dokumentacją kosztem właściwego oprogramowania. Takie podejście prowadzi do fatalnych skutków. Można tego błędu uniknąć, stosując następującą regułę:

Pierwsze prawo Martina dotyczące dokumentacji

Nie pracuj nad żadnymi dokumentami, chyba że w danym momencie bardzo ich potrzebujesz.

Faktyczna współpraca z klientem jest ważniejsza od negocjacji zasad kontraktu

Oprogramowanie nie może być zamawiane jak zwykły towar oferowany w sklepie. Opisanie potrzebnego oprogramowania i znalezienie kogoś, kto je dla nas opracuje w określonym terminie i za ustaloną cenę, jest po prostu niemożliwe. Wielokrotnie podejmowane próby traktowania projektów informatycznych w ten sposób zawsze kończyły się niepowodzeniem. Część tych niepowodzeń była dość spektakularna.

Kadra menedżerska często ulega pokusie przekazania zespołowi programistów swoich potrzeb z myślą o odbyciu kolejnego spotkania dopiero wtedy, gdy wrócą z gotowym systemem (w pełni zgodnym z oczekiwaniami zamawiających). Ten tryb współpracy prowadzi jednak albo do tworzenia oprogramowania fatalnej jakości, albo do zupełnego niepowodzenia projektów.

Warunkiem powodzenia projektu jest stała współpraca z klientem, najlepiej w formie regularnych, możliwie częstych spotkań. Zamiast opierać się wyłącznie na zapisach kontraktu, zespół programistów może ściśle współpracować z klientem, aby obie strony stale miały świadomość wzajemnych potrzeb i ograniczeń.

Kontrakt precyzujący wymagania, harmonogram i koszty przyszłego systemu z natury rzeczy musi zawierać poważne błędy. W większości przypadków zapisy takiego kontraktu stają się nieaktualne na długo przed zakończeniem realizacji projektu, a w skrajnych

przypadkach dezaktualizują się przed jego podpisaniem! Najlepsze kontrakty to takie, które przewidują ścisłą współpracę zespołu programistów z przedstawicielami klienta.

W 1994 roku miałem okazję negocjować kontrakt dla wielkiego, wieloletniego projektu (obejmującego pół miliona wierszy kodu), który zakończył się pełnym sukcesem. Członkowie zespołu programistów, do którego należałem, otrzymywali stosunkowo niewielkie uposażenia miesięczne i całkiem spore premie po dostarczeniu każdego większego bloku funkcjonalności. Co ciekawe, tego rodzaju bloki nie były szczegółowo opisane we wspomnianym kontrakcie. Umowa gwarantowała, że premie będą wypłacane dopiero wtedy, gdy dany blok przejdzie pozytywnie testy akceptacyjne klienta. Kontrakt nie określał też szczegółów przeprowadzania samych testów akceptacyjnych, których przebieg zależał wyłącznie od klienta zainteresowanego jak najlepszą jakością produktu końcowego.

W czasie realizacji tego projektu bardzo blisko współpracowaliśmy z klientem. Niemal w każdy piątek przekazywaliśmy mu jakąś część budowanego oprogramowania. Już w poniedziałek lub we wtorek kolejnego tygodnia otrzymywaliśmy listę zmian oczekiwanych przez klienta. Wspólnie nadawaliśmy tym modyfikacjom priorytety, po czym planowaliśmy ich wprowadzanie w ciągu kolejnych tygodni. Nasza współpraca z przedstawicielami klienta była na tyle bliska, że testy akceptacji nigdy nie stanowiły problemu. Doskonale wiedzieliśmy, kiedy poszczególne bloki funkcjonalności spełniały oczekiwania klienta, ponieważ jego przedstawiciele stale obserwowali postępy prac.

Wymagania stawiane temu projektowi stale były modyfikowane. Bardzo często decydowaliśmy się na wprowadzanie bardzo istotnych zmian. Zdarzało się, że rezygnowaliśmy lub dodawaliśmy całe bloki funkcjonalności. Mimo to zarówno kontrakt, jak i sam projekt okazał się pełnym sukcesem. Kluczem do tego sukcesu była intensywna współpraca z klientem i zapisy kontraktu, które nie definiowały szczegółowego zakresu, harmonogramu ani kosztów, tylko regulowały zasady współpracy wykonawcy ze zleceniodawcą.

Reagowanie na zmiany jest ważniejsze od konsekwentnego realizowania planu

O sukcesie bądź porażce projektu polegającego na budowie oprogramowania decyduje zdolność częstego i szybkiego reagowania na zmiany. Pracując nad planami realizacji projektu, koniecznie musimy się upewnić, że tworzony harmonogram będzie na tyle elastyczny, aby można było w jego ramach wprowadzać zmiany (zarówno w wymiarze biznesowym, jak i technologicznym).

Projektowi polegającego na tworzeniu systemu informatycznego nie można szczegółowo zaplanować. Po pierwsze, musimy się liczyć ze zmianami otoczenia biznesowego, które będą miały wpływ na stawiane nam wymagania. Po drugie, kiedy nasz system wreszcie zacznie działać, możemy być niemal pewni, że przedstawiciele klienta złączą wspominać o zmianach dotychczasowych wymagań. I wreszcie, nawet jeśli mamy pełną wiedzę o wymaganiach i jesteśmy pewni, że nie będą się zmieniały, nie jesteśmy w stanie precyzyjnie oszacować czasu ich realizacji.

Mniej doświadczeni menedżerowie ulegają pokusie nanoszenia całego projektu na wielkie arkusze papieru i rozklejania ich na ścianach. Wydaje im się, że takie schematy dają im pełną kontrolę nad projektem. Mogą łatwo śledzić wybrane zadania i wykreslać je ze schematu zaraz po ich wykonaniu. Mogą też porównywać obserwowane daty realizacji zadań z datami planowanymi i właściwie reagować na wszelkie odstępstwa od harmonogramu.

Prawdziwym problemem jest jednak to, że struktura takiego schematu szybko się dezaktualizuje. Wraz ze wzrostem wiedzy zespołu programistów na temat budowanego systemu rośnie wiedza klienta o potrzebach samego zespołu, a część zadań reprezentowanych na wykresie okazuje się po prostu zbędna. W tym samym czasie mogą być odkrywane zupełnie inne zadania, które należy dodać do istniejącego schematu. Krótko mówiąc, **zmianie** ulegają nie tylko daty realizacji poszczególnych zadań, ale także same zadania.

Dużo lepszą strategią planowania jest tworzenie szczegółowych harmonogramów i wykazów zadań na najbliższy tydzień, ogólnych planów na kolejne trzy miesiące oraz bardzo ogólnych, wręcz szkicowych planów na dalsze okresy. Powinniśmy mieć precyzyjną wiedzę o zadaniach, nad którymi będziemy pracować w przyszłym tygodniu, mniej więcej znać wymagania, jakie będziemy realizować w ciągu kolejnych trzech miesięcy, oraz mieć jakieś wyobrażenie o możliwym kształcie naszego systemu po roku.

Stopniowo obniżany poziom szczegółowości formułowanych planów oznacza, że poświęcamy czas na szczegółowe szacowanie tylko tych zadań, którymi będziemy się zajmowali w najbliższej przyszłości. Modyfikowanie raz opracowanego, szczegółowego planu działań jest trudne, bo w jego tworzenie i zatwierdzanie był zaangażowany cały zespół. Ponieważ jednak taki plan dotyczy tylko najbliższego tygodnia, pozostałe szacunki pozostają elastyczne.

Podstawowe zasady

Na podstawie ogólnych reguł opisanych w poprzednim podrozdziale można sformułować następujący zbiór dwunastu szczegółowych zasad. Właśnie te reguły odróżniają praktyki programowania zwinnego od tradycyjnych, ciężkich procesów:

1. **Naszym najwyższym nakazem jest spełnienie oczekiwań klienta przez możliwie wczesne dostarczenie wartościowego oprogramowania.** W kwartalniku „MIT Sloan Management Review” opublikowano kiedyś analizę praktyk wytwarzania oprogramowania, które ułatwiają przedsiębiorstwom budowę produktów wysokiej jakości³. W przytoczonym artykule zidentyfikowano szereg różnych praktyk, które miały istotny wpływ na jakość ostatecznej wersji systemu. Opisano między innymi ścisłą korelację pomiędzy jakością końcowego produktu a wczesnym dostarczeniem częściowo funkcjonującego systemu. W artykule dowiedziono, że **im mniej**

³ *Product-Development Practices That Work: How Internet Companies Build Software*, „MIT Sloan Management Review”, zima 2001, nr 4226.

funkcjonalny będzie początkowo dostarczony fragment systemu, tym wyższa będzie jakość jego wersji końcowej. W tym samym artykule zwrócono też uwagę na związek pomiędzy jakością systemu końcowego a częstotliwością dostarczania klientowi coraz bardziej funkcjonalnych wersji pośrednich. **Im częściej przekazujemy klientowi gotowy fragment funkcjonalności, tym wyższa będzie jakość produktu końcowego.**

Zbiór praktyk programowania zwinnego nakazuje nam możliwie wczesne i częste dostarczanie fragmentów oprogramowania. Powinniśmy przekazać klientowi początkową wersję budowanego systemu już w ciągu kilku tygodni od rozpoczęcia prac nad projektem. Od momentu przekazania pierwszej, najskromniejszej wersji oprogramowania powinniśmy co kilka tygodni dostarczać klientowi coraz bardziej funkcjonalne warianty. Jeśli klient uzna, że zaoferowana funkcjonalność jest wystarczająca, może się zdecydować na wdrożenie pośredniej wersji produktu w środowisku docelowym. W przeciwnym razie klient dokonuje przeglądu istniejącej funkcjonalności i przekazuje wykonawcy wykaz oczekiwanych zmian.

- 2. Traktujmy zmiany wymagań ze zrozumieniem, nawet jeśli następują w późnych fazach wytwarzania.** Procesy zwinne muszą się zmieniać wraz ze zmieniającymi się uwarunkowaniami biznesowymi, w których funkcjonuje strona zamawiająca. Wszystko jest kwestią właściwego podejścia. Uczestnicy procesów zwinnych z reguły nie obawiają się zmian — uważają je wręcz za zjawisko pozytywne, które prowadzi do lepszego rozumienia przez zespół programistów oczekiwań klienta.

Zespół pracujący nad zwinnym projektem musi poświęcać wiele czasu na utrzymywanie właściwej elastyczności struktury swojego oprogramowania, ponieważ tylko takie podejście pozwala zminimalizować wpływ zmian wymagań na kształt budowanego systemu. W dalszej części tej książki omówimy reguły, wzorce projektowe i praktyki programowania obiektowego, które ułatwiają nam zachowywanie takiej elastyczności.

- 3. Działające oprogramowanie należy dostarczać klientowi możliwie często (w odstępach kilkutygodniowych lub kilkumiesięcznych).** Powinniśmy przekazywać stronie zamawiającej pierwszą wersję pracującego oprogramowania tak szybko, jak to tylko możliwe, i kontynuować ten proces na wszystkich etapach realizacji projektu. Nie wystarczy dostarczanie plików dokumentów ani nawet najbardziej rozbudowanych planów. Żaden dokument nie zastąpi rzeczywistej funkcjonalności przekazywanej klientowi. Naszym ostatecznym celem jest dostarczenie zamawiającemu oprogramowania, które w pełni będzie spełniało jego oczekiwania.
- 4. Ludzie biznesu powinni ściśle współpracować z programistami na wszystkich etapach projektu.** Warunkiem zwinności projektu jest częsta i intensywna współpraca klientów, programistów i ośrodków biznesowych zaangażowanych w jego

finansowanie. Projektu polegającego na tworzeniu oprogramowania nie można traktować jak broni „odpal i zapomnij”. Tego rodzaju projekty muszą być stale kierowane.

5. **Projekty należy planować wokół dobrze umotywowanych programistów. Należy im zorganizować niezbędne środowisko i wsparcie, a także obdarzyć ich potrzebnym zaufaniem.** To ludzie są najważniejszym czynnikiem decydującym o powodzeniu projektu. Inne czynniki (w tym proces, środowisko i sposób zarządzania) mają znaczenie drugorzędne i mogą być dowolnie dostosowywane do potrzeb ludzi zaangażowanych w realizację projektu.
6. **Najbardziej efektywną metodą przekazywania informacji do i w ramach zespołu programistów jest rozmowa w cztery oczy.** Projekty zwinne polegają w dużej mierze właśnie na rozmowach prowadzonych przez członków zespołu. Podstawowym trybem takiej komunikacji są bezpośrednie kontakty programistów. Ewentualne dokumenty są pisane i aktualizowane równoległe do takich rozmów (zgodnie z harmonogramem prac nad oprogramowaniem).
7. **Podstawowym miernikiem postępu prac nad projektem jest ilość i jakość działającego oprogramowania.** Miarą postępu projektu zwinnego jest ilość oprogramowania, która w danej chwili spełnia oczekiwania klienta. Szacowanie bieżącego stanu prac w żadnym razie nie powinno polegać na weryfikacji realizowanej fazy projektu, objętości wytworzonej dokumentacji czy ilości gotowego kodu infrastrukturalnego. O 30% postępie możemy mówić tylko wtedy, gdy 30% funkcjonalności działa i spotyka się z akceptacją klienta.
8. **Procesy zwinne ułatwiają utrzymywanie ciągłości wytwarzania. Sponsorzy, programiści i użytkownicy powinni mieć możliwość utrzymywania stałego tempa pracy przez cały czas realizacji projektu.** Projekt zwinny dużo bardziej przypomina maraton niż bieg sprinterski na 100 metrów. Zespół programistów nie powinien od razu próbować osiągnąć maksymalnej wydajności, której utrzymanie w dłuższym okresie byłoby niemożliwe. Lepszym rozwiązaniem jest utrzymywanie wysokiej, ale stałej szybkości pracy.

Narzucanie zbyt wysokiego tempa pracy prowadzi do wypalenia psychicznego, podejmowania decyzji o wyborze rozmaitych skrótów i ostatecznego fiaska. Zwinne zespoły potrafią same utrzymywać stałą wydajność — żaden członek takiego zespołu nie musi korzystać z zapasów jutrzejszej energii z myślą o dłuższej bądź szybszej pracy dzisiaj. Programiści wchodzący w skład zwinnych zespołów utrzymują stałe tempo prac, które umożliwia im realizację standardów najwyższej jakości przez cały okres trwania prac nad realizacją projektu.
9. **Pozytywny wpływ na zwinność projektu ma stała dbałość o doskonałość techniczną i właściwy projekt.** Wysoka jakość jest kluczem do dużej wydajności. Oznacza to, że właściwym sposobem zachowania zadowalającej szybkości jest utrzymywanie możliwie prostej i przemyślanej struktury oprogramowania. Wszyscy

członkowie zwinnego zespołu powinni się koncentrować na wytwarzaniu kodu źródłowego o najwyższej jakości, na jaką pozwalają ich umiejętności. Nie doprowadzają do sytuacji, w której muszą kogoś zapewniać, że zaległe zadania zrealizują w bliżej nieokreślonej przyszłości. Eliminują źródła potencjalnych opóźnień w zarodku.

10. **Kluczowym elementem pomyślnego zakończenia projektu jest prostota, czyli sztuka maksymalizacji ilości pracy, której nie wykonujemy.** Zwinne zespoły nie próbują na siłę budować wielkich i skomplikowanych systemów — zawsze starają się dochodzić do stawianych im celów najkrótszą drogą. Nie przywiązują zbyt dużej wagi do ewentualnych problemów dnia jutrzejszego ani nie próbują za wszelką cenę zмагаć się z problemami bliższymi aktualnie realizowanych zadań. Ich celem jest osiągnięcie możliwie najwyższej jakości z jednoczesnym zapewnieniem elastyczności umożliwiającej łatwe modyfikowanie oprogramowania w odpowiedzi na przyszłe problemy.
11. **Źródłem najlepszych architektur, specyfikacji wymagań i projektów są zespoły, którym dano wolną rękę w zakresie organizacji.** Zwinny zespół programistów musi sam organizować swoją pracę. Odpowiedzialność za poszczególne zadania nie powinna być przydzielana wybranym członkom zespołu z zewnątrz, tylko komunikowana całemu zespołowi. Sam zespół powinien następnie decydować, jak te zadania najefektywniej zrealizować.

Członkowie zwinnego zespołu wspólnie pracują nad wszystkimi aspektami zleconego projektu. Każdy członek takiego zespołu może zgłaszać swoje propozycje dotyczące dowolnego aspektu realizowanego zlecenia. Za takie zadania, jak tworzenie architektury, definiowanie wymagań czy testy, nigdy nie odpowiadają poszczególni programiści. Odpowiedzialność za tak kluczowe czynności musi obejmować cały zespół, a każdy jego członek musi mieć możliwość wpływania na sposób ich realizacji.

12. **W stałych odstępach czasu zespół zaangażowany w tworzenie oprogramowania powinien analizować możliwości usprawnienia pracy i dostosowywać swoje dalsze działania do wniosków płynących z tej analizy.** Zwinny zespół stale doskonali swoją organizację, reguły, konwencje, relacje itp. Członkowie takiego zespołu doskonale zdają sobie sprawę z tego, że środowisko, w którym pracują, stale się zmienia, i wiedzą, iż ich zwinność zależy w dużej mierze od zdolności dostosowywania się do tych modyfikacji.

Konkluzja

Celem zawodowym każdego programisty i każdego zespołu programistów jest generowanie możliwie jak najwyższej jakości kodu dla pracodawców i klientów. Mimo to nadal przerażający odsetek tego rodzaju projektów kończy się niepowodzeniem lub nie

przynosi korzyści właściwym organizacjom. Środowisko programistów wpadło w pułapkę nieustannego rozwijania procesów, które mimo dobrych intencji dodatkowo utrudniły realizację projektów. Reguły i praktyki zwinnego wytwarzania oprogramowania powstały właśnie po to, by ułatwić zespołom wychodzenie ze spirali rozbudowywania procesów i koncentrowanie się wyłącznie na technikach prowadzących do osiągnięcia właściwych celów.

Kiedy pisano tę książkę, istniało wiele zwinnych procesów i metodyk, spośród których mogliśmy wybierać: SCRUM⁴, Crystal⁵, FDD (od ang. *Feature-Driven Development*)⁶, ADP (od ang. *Adaptive Software Development*)⁷ oraz programowanie ekstremalne (ang. *eXtreme Programming — XP*)⁸. Z drugiej strony, zdecydowana większość skutecznych zespołów zwinnych zdołała wybrać takie kombinacje części spośród wymienionych procesów, które tworzą ich własne, autorskie formy zwinności, lekkości. Do najczęściej stosowanej kombinacji należy połączenie metodyk SCRUM i XP, w którym metodykę SCRUM wykorzystuje się do zarządzania wieloma zespołami stosującymi metodykę XP.

Bibliografia

- [Beck, 99] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
- [Highsmith, 2000] James A., *Highsmith, Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House, 2000.
- [Newkirk, 2001] James Newkirk i Robert C. Martin, *Extreme Programming in Practice*, Addison-Wesley, 2001.

⁴ Zob. www.controlchaos.com.

⁵ Zob. http://alistair.cockburn.us/index.php/Crystal_methodologies_main_foyer.

⁶ Peter Coad, Eric Lefebvre i Jeff De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall, 1999.

⁷ [Highsmith, 2000].

⁸ [Beck, 99], [Newkirk, 2001].