helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

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

eXtreme programming



Autorzy: David Astels, Granville Miller, Miroslav Novak
Tłumaczenie: Andrzej Grażyński
ISBN: 83-7197-909-6
Tytuł oryginału: Practical Guide to Extreme Programming
Format: B5, stron: 286
Przykłady na ftp: 66 kB

„eXtreme programming" przedstawia nową metodologię i praktykę programowania w przystępny, a jednocześnie kompleksowy sposób. Autorzy omawiają podstawowe pojęcia programowania ekstremalnego: planowanie edycji, programowanie w parach, wczesne testowanie, „zręczne modelowanie" i refaktoryzację. Zostały one zaprezentowane na przykładzie konkretnego projektu, którego realizację omówiono w książce.

Kto powinien przeczytać tę książkę? Będzie ona niewątpliwie pożyteczną lekturą dla każdego, kto choć częściowo związał swe życie z tworzeniem oprogramowania – programisty, menedżera czy koordynatora projektu. I nawet gdyby miało skończyć się tylko na czytaniu – bez praktycznego zastosowania w najbliższym projekcie poznanych reguł, wiedza o programowaniu ekstremalnym okaże się bez wątpienia pożyteczna.

- Naucz się zasad programowania ekstremalnego i zastosuj je w praktyce
- Od konceptualizacji do oddania działającego systemu – poznaj najlepsze sposoby pracy na każdym etapie tworzenia programowania
- Poznaj znaczenie wszystkich uczestników procesu tworzenia oprogramowania: programistów, menedżerów i klientów
- Dowiedz się, jak rozwiązać najczęstsze problemy powstające przy wdrażaniu metodologii programowania ekstemalnego

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

Spis treści

Przedmowa **9**

Wprowadzenie **11**

Wstęp — w stronę samoaktualizacji **13**

Część I Reguły gry **19**

Rozdział 1. Zasady **21**

Współpraca z klientami 21
Metafory ułatwiają zrozumienie skomplikowanych zagadnień 22
Plan 22
Skracaj zebrania 23
Najpierw testy 24
Prostota 25
Programowanie parami 25
Kodowanie zgodnie ze standardami 26
Kolektywne kontrolowanie kodu 26
Nieustanna integracja 27
Refaktoryzacja 28
Udostępniaj rezultaty małymi fragmentami 28
Bez przemęczania się 29
Bądź przygotowany na zmiany 29

Rozdział 2. Gracze **31**

Dwa zespoły 31
Zespół klientów 32
Zespół programistów 34
Znaczenie ról 36
Ustanawianie praw 37
Konkluzja 37

Część II Konceptualizacja systemu **39**

Rozdział 3. Kreowanie wizji systemu **41**

Konceptualizacja systemu 41
Karta wizji 42
Metafory 43
Metafory w procesie projektowym 44
Konkluzja 44

Rozdział 4. Historie użytkownika ... 45
Koncepcja historii użytkownika ... 46
Historie użytkownika ... 46
Numeracja elementów na stosie ... 47
Konkluzja ... 48
Dodatek ... 51

Rozdział 5. Testy akceptujące ... 53
Czym są testy akceptujące? ... 53
Tworzenie testów akceptujących ... 54
Trudności związane z testami akceptującymi ... 56
Nieskończenie wiele testów ... 57
Automatyzacja testów akceptujących ... 57
Konkluzja ... 58

Rozdział 6. Jedno proste rozwiązanie ... 59
Czego więc poszukujemy? ... 59
Zrób to prosto ... 60
Meandry konceptualizacji ... 60
Konkluzja ... 61

Rozdział 7. Nazewnictwo ... 63
Problem ... 63
Czym są nazwy? ... 64
Stałe doskonalenie wiedzy ... 64
Konkluzja ... 65

Część III Planowanie ... 67

Rozdział 8. Oszacowania ... 69
Sporządzanie oszacowań ... 69
Założenia ... 70
Podział historii użytkownika ... 71
Trudności w planowaniu ... 72
Konkluzja ... 73

Rozdział 9. Planowanie etapów ... 75
Szybkość ... 76
Koszt etapu ... 77
Ustanawianie priorytetów ... 77
Programowanie parami ... 78
Tworzenie planu ... 79
Konkluzja ... 80

Rozdział 10. Planowanie iteracji ... 81
Definiowanie iteracji ... 81
Praktyczne aspekty planowania iteracji ... 82
Pierwsza iteracja ... 83
Kolejne iteracje ... 83
Określenie terminów ... 84
Konkluzja ... 84

Rozdział 11. Planowanie taktyczne ... 85
Rozpoczynanie iteracji ... 85
Przydzielanie zadań ... 86
„Stojące” zebrania ... 87

Śledzenie projektu ... 87
Przedterminowe wykonanie iteracji ... 88
Konkluzja ... 88

Część IV Tworzenie systemu ... 89

Rozdział 12. Programowanie parami ... 91
Organizacja programowania parami ... 91
Programowanie jako konwersacja ... 92
Presja partnera ... 93
Podwójna korzyść ... 93
Rezultaty ... 94
Tak szybkie, jak najwolniejsze ... 95
Konkluzja ... 96

Rozdział 13. Najpierw testowanie ... 99
Projekt XP ... 99
Dlaczego testy? ... 100
Co testować? ... 101
Kiedy testować? ... 101
Jak testować? Środowisko testowe ... 102
Programowanie sterowane testami ... 104
Przykład ... 106
Korzyści ... 108
Konkluzja ... 109

Rozdział 14. Projekt ... 111
Projekt a XP ... 112
Jak wzorce projektowe mają się do XP? ... 114
Architektura a XP ... 115
„Zwinne modelowanie" (AM) ... 115
Czym jest AM? ... 115
Przegląd wartości, zasad i praktyk AM ... 117
Konkluzja ... 123

Rozdział 15. Sugestywne kodowanie ... 125
Nazewnictwo ... 126
Prostota jest krańcowo skomplikowana ... 127
Gwarantowane założenia ... 128
Niech kompilator zrobi to za Ciebie ... 128
Bez komentarzy ... 129
Kolektywne kontrolowanie kodu ... 131
Lepiej spłonąć niż zwiędnąć — czyżby? ... 131
Zadowoleni programiści to produktywni programiści ... 132
Ciesz się życiem ... 132
Konkluzja ... 133

Rozdział 16. Refaktoryzacja ... 135
Przykłady refaktoryzacji ... 136
Odwaga ... 137
Zapachy kodu ... 137
Kiedy refaktoryzować? ... 138
Dwa kapelusze ... 139
Przykłady refaktoryzacji ... 139
Konkluzja ... 143

Rozdział 17. Nieuchronne integrowanie 145
Kilka praktycznych wskazówek 146
Integruj tylko po kompletnym przetestowaniu 147
Antywzorzec 147
Dwa podejścia do integracji 148
A co z przeglądem kodu? 149
Konsekwencje kolektywnej kontroli kodu 149
Konkluzja 150

Część V Udostępnianie 151

Rozdział 18. Przekazanie systemu 153
Dzień dostarczenia 153
Przekazanie produkcyjne 154
Testowanie 154
Świętowanie zwycięstwa 155

Część VI Zagadnienia dodatkowe 157

Rozdział 19. Adoptowanie i adaptacja XP 159
Zrozumienie XP 159
Adoptowanie XP 160
Metody adopcji 161
Adaptacja XP 162
Praktyki XP 163
Zwrot inwestycji 163
Konkluzja 163

Rozdział 20. Skalowanie XP 165
Programowanie w dużej skali 165
Organizacja dużych projektów XP 166
Otwarta i uczciwa komunikacja 167
Integracja 168
Znaczenie mądrych ludzi 168
Konkluzja 169

Rozdział 21. Przyszłość XP 171
Dokąd zmierza XP? 171
Na polu bitwy 172
Holoniczne podejście do XP 172
Tworzenie oprogramowania jako rzemiosło 174
Ewolucja XP 175

Dodatki 177

Przykładowa aplikacja 179

Bibliografia 275

Skorowidz 279

Rozdział 14.
Projekt

Pilnie strzeż konieczności i potrzeby chwili.
I nie przejmuj się wielkimi zamierzeniami.

— W. Szekspir, Ryszard III (akt IV, sc. 4)

Wszelkie zamiary pochodzące z serca
Toną w bezmiarze obietnic.

— W. Szekspir, Troilus i Cresida (akt I, sc. 3)

Projekt w XP? Niektórzy przeciwnicy XP przekonani są co do tego, iż pojęcie projektu jest tej filozofii zdecydowanie obce. XP jawi im się jako praktyka z pogranicza hakerstwa i źródło destrukcji — programiści niczego nie projektują, tylko zakasują rękawy i piszą programy. Żadnej dyscypliny, tylko żywioł.

To jednak obraz zdecydowanie nieprawdziwy. XP dalekie jest od niezdyscyplinowanego żywiołu, jest wysoce zdyscyplinowanym podejściem do tworzenia oprogramowania. XP nie jest „antyprojektowe"; jest oparte na *ciągłym* projektowaniu, sprzeciwia się jedynie dużym projektom zapiętym a priori na ostatni guzik. Jako alternatywę dla takich projektów proponuje doraźne projektowanie małych zadań, dyktowanych potrzebą chwili.

Pozostałością „globalnej wizji projektu" jest w XP uzyskanie ogólnej perspektywy systemu, określonej przez stosowne metafory. Ta „wysokopoziomowa mapa" systemu uszczegóławiana jest w trakcie jego realizacji — kiedy programiści przystępują do realizacji związanych z tym zadań, dokonują planowania cząstkowego dotyczącego tych właśnie zadań.

W niniejszym rozdziale przyjrzymy się niektórym zadaniom związanym z projektowaniem w XP, w szczególności:

- modelowaniu i jego związkami z ideą XP,
- nowej koncepcji „zwinnego modelowania" (*Agile Modeling*[1], w skrócie AM),
- narzędziom wspomagającym AM.

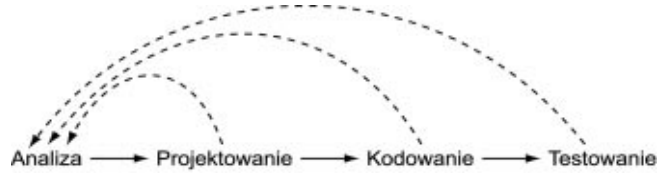
[1] Dawna nazwa — *Extreme Modeling*.

Projekt a XP

XP promuje ewolucyjne podejście do projektowania. Niegdyś, gdy próbowano realizować w taki sposób projekty informatyczne, często otrzymywano w efekcie gigantyczny bałagan. Jako antidotum na tę improwizację wymyślono więc planowanie a priori — podobne do tego, jakie architekci stosują przy budowie mostów: wszystkie najdrobniejsze szczegóły przedsięwzięcia planowane są na początku, przed ich faktycznym rozpoczęciem. Niewątpliwie przy realizacji projektów architektonicznych podejście takie jest jak najzupełniej słuszne; samo planowanie z dużym wyprzedzeniem jest zaś możliwe dlatego, że w zasadzie a priori znane są wszystkie uwarunkowania, niezmienne przez cały okres realizacji przedsięwzięcia.

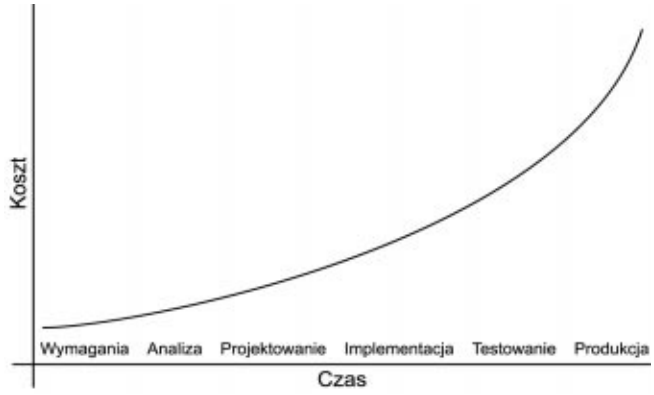
Tworzenie oprogramowania rządzi się jednak innymi regułami. Kiedy tworzysz np. dwudziesty system księgowy, możesz z dużą dokładnością przewidzieć wiele jego szczegółów; kiedy jednak jest to Twój pierwszy system tego rodzaju, brak Ci należytego doświadczenia niezbędnego ku temu, aby określić chociażby wymagania, jakie powinien spełniać produkt finalny. Jeżeli w tych warunkach zabierzesz się do planowania (czytaj: przewidywania), już niedługo przekonasz się, iż Twoje plany rozmijają się z rzeczywistością — bo zmieniły się wymagania, bo zmieniło się zdanie klientów, którzy początkowo niezbyt dobrze rozumieli, czego im naprawdę potrzeba, bo zmieniło się prawo itd. Twój początkowy plan będzie nieustannie korygowany, a być może w końcu postanowisz opracować go raz jeszcze od początku. Rysunek 14.1 w prosty sposób przedstawia to, co z Twoim projektem może się dziać przez kilka najbliższych miesięcy.

Rysunek 14.1. *Klasyczne projektowanie w zmieniających się warunkach*



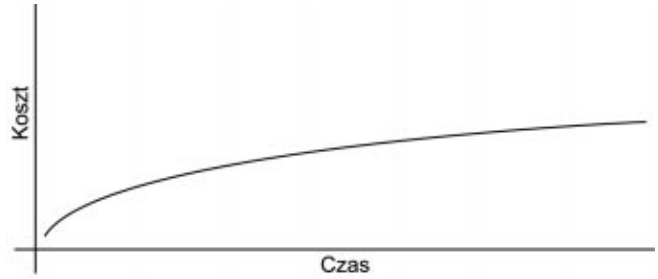
Nie ma nic za darmo; wraz z ciągłymi „nawrotami” w projekcie (który podobny będzie raczej do wodospadu) rosnąć będą także koszty — na rysunku 14.2 przedstawiono tempo ich wzrostu wraz z upływem czasu, w odniesieniu do „klasycznego” projektu programistycznego.

Rysunek 14.2. *Krzywa zmiany kosztów w klasycznym projekcie informatycznym*



Krzywa ta ulega jednak znacznemu „spłaszczeniu” w przypadku projektu XP (rysunek 14.3).

Rysunek 14.3. *Krzywa zmiany kosztów w projekcie XP*

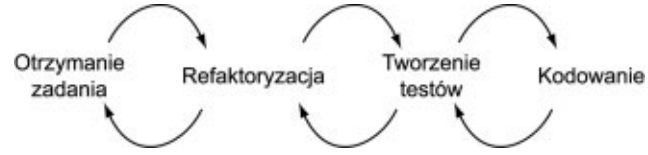


Czynnikami, które w największym stopniu wpływają na spłaszczenie krzywej kosztów projektu XP są między innymi:

- prostota projektowania,
- zautomatyzowane testowanie,
- agresywna refaktoryzacja.

W porównaniu z klasycznym projektem informatycznym (por. rysunek 14.1), projekt XP zawiera więcej „pętli”, ściślej ze sobą powiązanych. Na rysunku 14.4 przedstawione są cykle związane z procesem[2] XP; nie uwzględniono tam tzw. gry planistycznej[3] (*Planning Game*), gdyż koncentrujemy się jedynie na cyklu rozwojowym.

Rysunek 14.4. *Cykle procesu rozwojowego w XP*



Jak to wszystko działa? Dostajesz określone zadanie, siadasz do komputera, dokonujesz refaktoryzacji (uproszczenia lub poprawienia) istniejącego kodu, tworzysz niezbędne testy (być może znowu refaktoryzując kod), wreszcie tworzysz zasadniczy kod i testujesz go. Zwróć szczególną uwagę na pętlę „tworzenie testów-kodowanie”: w tej właśnie pętli będzie się kręcić Twoja praca dopóty, dopóki Twoje zadanie nie zostanie zrealizowane bezbłędnie (w sensie zaliczenia wszystkich stworzonych dla niego testów). Być może będziesz chciał dokonać „wyklarowania” istniejącego kodu, co w kategoriach rysunku 14.4 oznacza dodatkowo jeden lub kilka „obrotów” w ramach pętli „refaktoryzacja-tworzenie testów”.

Pojawienie się koncepcji programowania obiektowego (OOP) wywarło decydujący wpływ na techniki programowania, zwłaszcza w odniesieniu do systemów bazujących na interfejsach. Interfejs obiektu uzewnętrznia te elementy, które reprezentują mechanizmy

[2] W kontekście XP słowo „proces” używane jest raczej w potocznym znaczeniu.

[3] Tak określa się w XP analizę i zbieranie wymagań.

rządzące jego zachowaniem; szczegóły implementacyjne tych mechanizmów ukryte są przed użytkownikiem obiektu, co określane jest mianem *enkapsulacji* lub *hermetyzacji*. „Odseparowanie" klienta od wnętrza obiektu umożliwia jego autorom swobodne manipulowanie szczegółami implementacji, dla klienta bowiem istotne jest tylko zewnętrzne zachowanie się obiektu, określone przez jego interfejs.

Takim „interfejsem" dla testowanego fragmentu kodu jest jego zewnętrzne zachowanie się i taki właśnie interfejs jest podstawą wyprzedzającego tworzenia testów. Jednocześnie interfejs ten jest podstawą projektowania w XP — projektowania opartego na *skutkach* funkcjonowania kodu; szczegóły implementacyjne określane są dopiero w drugiej kolejności.

Ponieważ cykle „testy-projektowanie-kodowanie" są względnie małe i regularnie otrzymujesz działający system[4], możesz regularnie doświadczać sprzężenia zwrotnego umożliwiającego utrzymywanie biegu projektu we właściwych ryzach; związane z tym konieczne zmiany w projekcie bywają stosunkowo niewielkie, właśnie ze względu na niewielki rozmiar wspomnianych cykli. Przystosowywanie projektu do zmieniających się wymagań odbywa się więc stosunkowo małym kosztem.

Projektowanie w XP ma więc charakter „globalny" tylko początkowo — w czasie tworzenia ogólnej perspektywy systemu i związanych z nią metafor. Potem następuje już „drobnoziarniste" projektowanie, przeplatające się z refaktoryzacją. W efekcie otrzymujemy prosty, przejrzysty projekt ucieleśniony w prostym, przejrzystym kodzie.

Jak wzorce projektowe mają się do XP?

W XP, tak jak w każdym przedsięwzięciu projektowym, obecne są wzorce projektowania. XP dostarcza zasad i zaleca pewne praktyki w procesie tworzenia oprogramowania; wzorce projektowe obejmują związaną z tym wiedzę i doświadczenie. Zasada ta obowiązuje niezależnie od tego, *jaki rodzaj* oprogramowania jest tworzony i *jak* przebiega sam proces projektowy.

Specyfiką XP jest natomiast to, że na jego gruncie wzorce projektowe nie są celem samym w sobie. W XP ważne jest nie tylko zrozumienie, jak korzystać z danego wzorca, ale także — w jakich sytuacjach z niego korzystać, a w jakich absolutnie nie. Innymi słowy — nie powinno się używać określonego wzorca projektowego dotąd, aż nie będzie to konieczne. Właściwe wykorzystywanie wzorców jest preferowanym sposobem ewoluowania projektu — rozpocznij od rzeczy najprostszych, które prawdopodobnie będą działać i przekształcaj je stopniowo w rzeczy bardziej złożone, które z czasem mogą stać się wzorcami projektowymi.

[4] To skutek charakterystycznego dla XP wczesnego i częstego integrowania kodu.

Architektura a XP

W XP trudno jednoznacznie wskazać elementy, które można by określić jako architekturę. Architektura jest tu bowiem określona częściowo przez metafory, opisujące ogólny kontekst i kształt systemu. Zdeklarowani zwolennicy XP — Beck, Jeffries, Martin i inni — wyrażają opinię, iż jakichkolwiek wstępnych projektów architektury należy bezwzględnie unikać. Refaktoryzacja jest tym czynnikiem, który stopniowo nadaje projektowi określoną postać. Nie zgadza się z tą kategoryczną opinią Fowler, twierdząc, iż konieczne są pewne ogólne założenia architektoniczne. Osobiście przychylam się do zdania pierwszej grupy — dążę bowiem do tworzenia najprostszych rzeczy, które prawdopodobnie będą działać i staram się utrzymywać kod w jak najprostszej postaci. Na jego komplikacje decyduję się tylko wtedy, kiedy okaże się to uzasadnione.

„Zwinne modelowanie" (AM)

Scott Ambler przewodzi społeczności, która opracowała koncepcję „zwinnego modelowania" (*Agile Modeling* — AM). Niniejszy fragment stworzony został głównie na podstawie internetowych materiałów związanych z tą tematyką oraz książki Scotta.

AM znajduje się dopiero w stadium formowania, stąd wszelkie związane z nim definicje (dotyczące wartości, zasad, praktyk itp.) mają charakter płynny i są przedmiotem wielu dyskusji. Więc to, o czym będę pisać w dalszym ciągu, odzwierciedla aktualny (w momencie wydania książki) stan tego, co moim zdaniem jest istotą AM. Ogólny charakter AM można bowiem już uważać za wysoce ustalony, a we wspomnianej społeczności panuje powszechna zgoda co do podstawowych wartości AM. Toczą się jednak zawzięte dyskusje na temat zasad — tak więc głównym celem prezentowanego tu materiału jest przedstawienie czytelnikowi ogólnego klimatu AM.

Czym jest AM?

AM jest opartą na praktyce metodologią efektywnego modelowania i dokumentowania systemów informatycznych. Mówiąc prosto, AM jest kolekcją wartości, zasad i praktyk modelowania oprogramowania, które mogą zostać zastosowane w procesie jego tworzenia szybko i bez komplikacji. AM jest bardziej efektywne od tradycyjnych technik modelowania, ponieważ nie pretenduje do tego, by tworzone modele były perfekcyjne — wystarczy, że będą one dobre. Elementy AM mogą być stosowane w procesie zbierania wymagań, analizy, definiowania architektury i projektowania.

AM nie jest procesem *opisanym a priori*; innymi słowy, nie definiuje ono *szczegółowych* procedur ani sposobów postępowania w celu stworzenia takiego czy innego modelu. Zamiast tego dostarcza ono wskazówek do tego, jak efektywnie przeprowadzać proces modelowania. AM nie jest jednak bardziej prymitywne od innych technik modelowania — wielu programistów uważa, iż za jego pomocą mogą wykonać więcej niż w przypadku tradycyjnych narzędzi. Jednocześnie AM postrzegane jest jako narzędzie dla osób wrażliwych (*„touchy feely"*), wręcz raczej dla „artystów" niż „naukowców".

Podstawowe wartości AM, mające zapewnić efektywność modelowania, określone zostały przez stowarzyszenie Agile Alliance. Ten system wartości daje pierwszeństwo:

- indywidualizacji i współdziałaniu — przed procesami i narzędziami;
- działającemu oprogramowaniu — przed obszerną dokumentacją;
- współpracy z klientami — przed negocjacją kontraktu;
- reagowaniu na zmiany — przed sztywnym trzymaniem się planu.

Ogólna charakterystyka AM

Indywidualizacja i współdziałanie kontra procesy i narzędzia

Ważni są ludzie i komunikacja. DeMarco twierdzi, iż „socjologia może więcej niż technika, a nawet pieniądze".

Działające oprogramowanie kontra dokumentacja

Celem wysiłku programistów jest tworzenie oprogramowania, i to takiego, które wykonuje oczekiwane funkcje prawidłowo i efektywnie — czyli po prostu takiego, które działa. Nie jest tworzenie dokumentacji, która jest dla oprogramowania elementem pomocniczym (choć niezbędnym).

Uczestniczyłem kiedyś w projekcie, w którym poprzedni zespół projektowy wyprodukował tyle dokumentacji, iż do jej wydrukowania konieczny był zakup szybkiej drukarki (inaczej „zatkałaby się" kolejka wydruków). Jak na ironię, większość z tych dokumentów używana była bardzo rzadko (jeżeli w ogóle), były one bowiem tak ściśle związane z konkretnymi rozwiązaniami, i ogólnie tak dyletanckie, że praktycznie nie dało się z nich korzystać.

Modele powinny być rzeczami dynamicznymi, a nie statycznymi dokumentami. Wymagają one bowiem zazwyczaj różnych zmian, stosownie do nowych wymagań pojawiających się w czasie realizacji projektu. Modele sporządzone na papierze mają więc z definicji krótki żywot i używane są przeważnie w charakterze „migawek" odzwierciedlających stan aktualny. Rzeczywiste modele zawarte są w kodzie produktów.

Współpraca z klientami kontra negocjacja kontraktu

Negocjowanie kontraktu jest z założenia procesem *spornym*. Znacznie lepszych wyników można się spodziewać, jeżeli negocjacje ustąpią miejsca współpracy. Zespół programistów nie jest bowiem w stanie w pełni zarządzać dynamicznym, zmieniającym się modelem bez bieżącego udziału klientów.

Negocjowanie oznacza dążenie do ustalenia a priori niezmiennych później wymagań, które tym samym stają się nawet ważniejsze i od klientów, i od programistów. Ścisła współpraca pomiędzy zespołami umożliwia natomiast zmianę tych wymagań „w locie", celem np. przystosowania ich do zmienionych warunków, daje także programistom możliwość poszukiwania lepszych rozwiązań, dzięki lepszemu rozumieniu tych wymagań.

Współpraca z klientami staje się znacznie ułatwiona, jeżeli zespół programistów ma możliwość kontaktu „na miejscu" z ich przedstawicielem (przedstawicielami), co jest praktyką ściśle zalecaną przez XP.

Reagowanie na zmiany kontra sztywne trzymanie się planu

Jeżeli oprogramowanie ma odpowiadać wymaganiom klientów, musi być ono tworzone w środowisku umożliwiającym szybkie reagowanie na zmieniające się warunki, wpływające na zmiany tychże wymagań.

Zmienność wymagań klientów jest faktem. Trudno przewidywać wszelkie wymagania, jeżeli jednak byłoby to w jakiś sposób możliwe, to i tak nie będzie gwarancji, iż wymagania te zostaną od razu właściwie zrozumiane. Zazwyczaj także klienci nie od razu zdają sobie sprawę z wszystkich swych potrzeb — nie są oni zazwyczaj informatykami i ich zdanie w tej materii może ulegać zmianie w miarę jak będą obserwować udostępniane im sukcesywnie fragmenty tworzonego systemu.

By sprostać zmieniającym się wymaganiom, potrzebna jest więc elastyczność; polega ona na ciągłym, „przyrostowym" modyfikowaniu projektu. Należy unikać projektowania „na zapas", a ograniczyć się do zmian wymuszonych aktualnymi okolicznościami — jeżeli zmiana, którą zamierzasz wprowadzić do projektu, nie jest teraz potrzebna, zrezygnuj z niej (to tzw. zasada *YAGNI — You Ain't Gonna Need It*).

Tak właśnie wygląda (w dużym uproszczeniu) koncepcja „zwinnego modelowania" (AM), takie są jej wartości, zasady i praktyki, o których wcześniej wspominałem. Nie sposób oczywiście nie dostrzec ich dużego podobieństwa do XP.

Przegląd wartości, zasad i praktyk AM

Wartościami AM, przejętymi z XP i przystosowanymi do konkretnych zagadnień, są: *komunikacja*, *prostota*, *sprzężenie zwrotne*, *odwaga* i *skromność*. Do tego, aby modelowanie zakończyło się sukcesem, niezbędna jest komunikacja pomiędzy wszystkimi uczestnikami projektu, konieczne jest dążenie do poszukiwania jak najprostszego rozwiązania spełniającego wszystkie potrzeby, wymagane jest także sprzężenie zwrotne w celu należytego spożytkowania wysiłku programistycznego; nie da się tego wszystkiego zrealizować bez należytej odwagi, jak również pewnej dozy skromności pozwalającej zdać sobie sprawę z własnej omylności i wyciągnąć wnioski ze swych błędów.

AM opiera się na kolekcji *zasad*, do których zaliczyć można *dążenie do prostoty* przy modelowaniu oraz *podatność na zmiany*, wynikającą ze zmiennego charakteru wymagań klientów. AM wymaga zrozumienia roli *przyrostowej modyfikacji* systemu, co jest praktycznym wyrazem wspomnianej podatności na zmiany, oraz *szybkiego sprzężenia zwrotnego*, pozwalającego ocenić, czy modelowanie faktycznie posuwa się w dobrym kierunku. Modelowanie powinno być ponadto *ukierunkowane na konkretny cel* — nie należy pracować nad modelowaniem czegokolwiek, co nie jest temu celowi podporządkowane; efektywność modelowania wymaga ponadto *dysponowania wieloma modelami* odzwierciedlającymi różne *punkty widzenia*. Modele niekoniecznie muszą być

odrębnymi dokumentami, lecz mogą być organicznie *wbudowane* w kod programu — obowiązuje tu naczelna zasada *wyższości zawartości nad reprezentacją*, zgodnie z którą ta sama koncepcja może być modelowana na kilka różnych sposobów. Względy efektywności modelowania wymagają dokładnego poznania dostępnych *narzędzi* i *modeli wzorcowych*; efektywność pracy w zespole uzależniona jest od wielu czynników psychologicznych — programiści powinni więc zrozumieć, iż każdy z nich może się zawsze czegoś *nauczyć od kogoś innego*, a *otwarta, przyjazna i uczciwa komunikacja* sprzyja efektywnej współpracy. Wreszcie, najważniejszym aspektem wykonywanej pracy powinna być jej *jakość*, a zastosowanie AM w konkretnych warunkach może wymagać pewnej *adaptacji* jego zasad.

AM postuluje także konkretne *działania* (praktyki). Fundamentalnym zaleceniem jest *równoległe tworzenie kilku modeli*, tak by zawsze można było łatwo *dostosować któryś z nich* do konkretnej, nowej sytuacji i kontynuować pracę w nowych warunkach. Modelowanie powinno być prowadzone *małymi krokami* — należy unikać monumentalnych, uniwersalnych modeli przypominających „wieżę z kości słoniowej". Jako że modele są tylko abstrakcyjną reprezentacją oprogramowania, należy dążyć do *sprawdzenia ich w rzeczywistym kodzie* i udowodnienia tym samym, że funkcjonują w praktyce, nie tylko w teorii. Aby wszystkie potrzeby (klientów i programistów) zostały należycie zrozumiane i zaspokojone, *wszyscy oni muszą aktywnie uczestniczyć* w realizacji projektu. Jeżeli chodzi o *cele* modelowania, to mogą być one dwojakie: koncepcyjne i komunikacyjne. *Modelowanie koncepcyjne* ułatwia zrozumienie złożonych idei zawartych w projekcie systemu, zadaniem *modelowania komunikacyjnego* jest natomiast przedstawienie informacji o pracach wykonywanych (lub przeznaczonych do wykonania) przez zespół. Praktyczną realizacją zasady *dążenia do prostoty* jest tworzenie modeli o *prostej zawartości* (obejmującej jedynie niezbędne zagadnienia i nie zawierającej zbytnich szczegółów), *prostym opisie* (przez zastosowanie odpowiednio prostej notacji) z zastosowaniem *prostych narzędzi*. Należy unikać *nadmiernej liczebności modeli* — pozbywać się modeli tymczasowych i uaktualniać pozostałe *tylko wtedy, gdy staną się nieadekwatne*. Sprawna *komunikacja* może być zapewniona poprzez *publiczną prezentację* modeli — np. za pomocą projektorów lub w sieci WWW — dzięki wykorzystaniu *standardów modelowania*, dzięki *kolektywnej kontroli kodu* itp. Opłacalność wysiłku programistycznego znacznie wzrasta, jeżeli w modelu zawarte są elementy *testujące*, jeżeli model ten oparty jest na znanych *szablonach* oraz jeżeli wykorzystuje on elementy innych *istniejących* modeli. Nie jest także wykluczone, iż wdrożenie nowego systemu, wykorzystującego nowoczesne mechanizmy bazodanowe — np. oparte na sieciach WWW — będzie wymagało jego integracji z systemami wykorzystującymi starsze bazy danych; należy więc przewidzieć niezbędne modele uwzględniające różne warianty wykonania tej operacji.

Wartości AM

AM oferuje następujące wartości:

- **Komunikacja** — niezbędna jest efektywna komunikacja pomiędzy wszystkimi uczestnikami projektu: programistami, menedżerami i klientami. Jest ona jednym z celów modelowania.

- **Uczciwość** — najbardziej nawet efektywna komunikacja jest bezużyteczna, jeżeli nie towarzyszy jej uczciwość: uczciwość programistów pod względem oszacowań, problemów, opóźnień itd., i uczciwość klientów odnośnie do priorytetów, wymagań, kierunków zmian itd. Tylko w takich warunkach ludzie mogą mieć zaufanie do efektów czyjejś pracy.
- **Prostota** — tworzenie jak najprostszych rzeczy, które prawdopodobnie będą działać, jest naczelną zasadą XP stosującą się zarówno do kodowania, jak i modelowania.
- **Sprzężenie zwrotne** — stałe sprzężenie zwrotne jest niezbędne do uzyskania pewności, że poszczególne działania posuwają się we właściwym kierunku, bądź do jak najwcześniejszego skorygowania tych działań, jeżeli wymaga tego ich niezgodność z obecnymi warunkami.
- **Odwaga** — należy mieć odwagę do wykonywania działań zalecanych przez XP (i AM w szczególności): podejmowania ważnych decyzji i ponoszenia odpowiedzialności z tego tytułu, dokonywania ważnych refaktoryzacji modelu itp.
- **Skromność** — należy być świadomym własnej omylności i popełnianych błędów. Należy być otwartym na pomysły innych kolegów, bo być może są one lepsze od własnych pomysłów. Należy być przygotowanym na realizację dobrych idei niezależnie od tego, czyjego są one autorstwa.

Zasady AM

Oto najważniejsze zasady AM:

- **Jeden model wart jest 1024 linii kodu** — jasny, szkicowy model może ujawnić wiele istotnych szczegółów, które mogłyby pozostać niezauważone podczas studiowania kodu.
- **Modeluj zgodnie z celem** — modelowanie dla samego modelowania jest bezużyteczne; powinno być ono podporządkowane określonemu celowi.
- **Stosuj właściwy model, wykorzystuj wiele modeli** — niekiedy pojedynczy model okazuje się wystarczający do wyjaśnienia wszystkich zawiłości; ze względu jednak na zróżnicowany punkt widzenia poszczególnych uczestników projektu, może okazać się pożyteczne opracowanie kilku modeli, stanowiących kilka różnych spojrzeń na to samo zagadnienie, mechanizm, koncepcję itp.
- **Poznaj swoje modele** — to oczywiste: jeżeli sam nie rozumiesz swych modeli, w jaki sposób możesz wyjaśniać je innym?
- **Poznaj swoje narzędzia** — w każdej pracy wiele zależy od stosowanych narzędzi. Jeżeli poznasz narzędzia będące do Twojej dyspozycji, będziesz używał ich z pełnym zaufaniem i bez obaw.
- **Ucz się nieustannie** — traktuj każdą sytuację lub wyzwania również jako okazję do nauczenia się czegoś nowego. Dla mnie osobiście jedną z takich sytuacji jest praca w parze z kimś bardziej lub mniej doświadczonym

— bardziej doświadczony kolega może podzielić się ze mną swą wiedzą, mniej doświadczony może natomiast prezentować bardziej świeże spojrzenie na dany temat, może także domagać się wyjaśnienia różnych kwestii, które dla mnie wydają się oczywiste. Zawsze można się przy tym czegoś nauczyć — na przykład nieoczekiwanie odkryć nowy sposób rozwiązania problemu, lepszy od dotychczas stosowanego.

Praktyki AM

Wreszcie, istnieje kilka konkretnych praktyk postulowanych przez AM:

- **Modeluj w celu zrozumienia** — tworzenie modeli jest okazją do zgłębienia problemu i poznania jego przestrzeni rozwiązań. Uwzględnij możliwość zrezygnowania z niektórych modeli — mogą one bowiem być (według) tylko „krokami na drodze do oświecenia".
- **Modeluj komunikatywnie** — bądź zwięzły, ale wyrazisty. Staraj się uczynić model tak treściwym, jak tylko potrafisz.
- **Twórz proste modele** — nie próbuj budować modelu zasługującego na miano „matki wszystkich modeli". Buduj małe modele odzwierciedlające małe części systemu. Modele ogólne mają swe prawa, ale nie wtedy, kiedy próbujesz zrozumieć szczegóły.
- **Używaj prostych narzędzi** — nie ignoruj możliwości korzystania z kartki i ołówka. Parafrazując znaną zasadę XP — używaj jak najprostszych narzędzi, za pomocą których prawdopodobnie wykonasz swoją pracę. Niektóre skomplikowane narzędzia są narzędziami tylko z nazwy, tworzą one bowiem modele... dla samej sztuki modelowania.
- **Modeluj z partnerem** — to naturalne rozszerzenie programowania parami. Wszelkie korzyści wynikające z dwuosobowego tworzenia kodu przekładają się w prosty sposób na modelowanie.
- **Modeluj w grupie** — niekiedy okazuje się celowe tworzenie modelu z udziałem całego zespołu (lub dużej jego części). Bywa tak w sytuacjach złożonych, obejmujących wielorakie zależności, kiedy podejmowane decyzje również miewają wielorakie konsekwencje. Spojrzenie na model przez wiele par oczu zawsze jest korzystne — bowiem w przeciwieństwie do kodowania, to nie para jest „optymalną liczebnością". Należy jednak mieć na uwadze fakt, że w zbyt licznym zespole znacznie trudniej o koncentrację.
- **Uzewnętrzniaj swe modele** — wykorzystuj każde miejsce, gdzie mógłbyś trwale umieścić szkic swego modelu: tablice, ściany, jak również swą sieć intranetową.

UML a XP

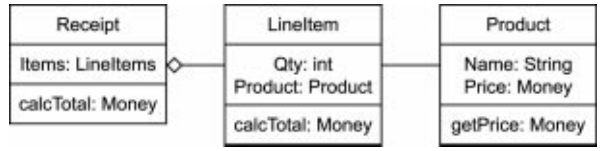
Ujednolicony język modelowania — *Unified Modeling Language* (UML) jest obecnie przemysłowym standardem notacyjnym stosowanym do graficznego przedstawiania zależności funkcjonalnych na potrzeby programowania obiektowego. Byłem świadkiem wielu dyskusji o tym, w jakim stopniu (o ile w ogóle) UML zgodny jest z zasadami XP; czytałem wiele artykułów na ten temat, większość z nich jednak dotyczyła raczej ludzi stosujących XP *zamiast* UML.

Idea UML bywa często źle rozumiana. Jak przed chwilą stwierdziłem, standard UML jest przede wszystkim *wizualną notacją* i jako taki był i jest z powodzeniem wykorzystywany. Nie jest więc pod żadnym względem sprzeczny z XP i może być w naturalny sposób użyty na potrzeby AM.

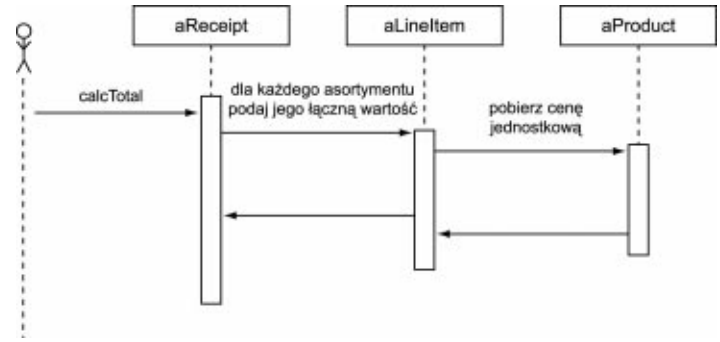
UML wykorzystuje wiele różnych typów diagramów. Większość z nich nie jest powszechnie używana na gruncie XP, chociaż niektórzy klienci mogą ich sobie zażyczyć w dokumentacji towarzyszącej systemowi (co mogłoby znaleźć odzwierciedlenie w historii użytkownika pt. „Oto jest diagram funkcjonalny systemu").

Szczególnie użyteczne dla XP okazują się diagramy trzech następujących typów:

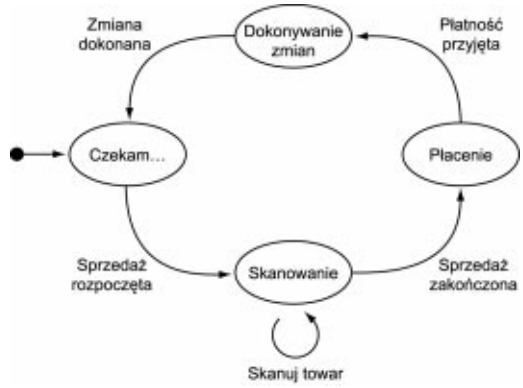
- diagramy klas:



- diagramy sekwencji:



- diagramy stanów:



Głównym celem rysowania diagramów jest pomoc w myśleniu o problemie i jego możliwych rozwiązaniach. Diagramy te nie są jednak dziełami sztuki i nie ma sensu marnotrawić czasu na ich artystyczny wygląd. Typowy diagram rysowany jest odręcznie na tablicy, na kartce, na serwetce i in.; diagramy rysowane „same dla siebie" są więcej niż bezużyteczne. Peter Coad stwierdził kiedyś — cytując swego ojca — „podejmując się zrobić cokolwiek, rezygnujesz jednocześnie ze zrobienia czegoś innego"; każdą minutę straconą na cyzelowanie diagramów mógłbyś z pożytkiem przeznaczyć np. na tworzenie kodu.

Niekiedy bez diagramów nie można się po prostu obejść. Pomagają w myśleniu, pomagają w komunikowaniu się. Diagramy sporządzone na papierze mają ograniczony żywot — należy pozbywać się ich natychmiast, jak tylko spełnią swoje zadanie i okażą się niepotrzebne. Ich ewentualna „aktualizacja" to czysta strata czasu, a przy okazji ryzykujemy to, iż prędzej czy później posłużymy się diagramem *nieaktualnym*.

Osobiście odradzam korzystanie z różnych narzędzi do tworzenia diagramów — za pomocą ołówka rysuje się diagramy znacznie szybciej, poza tym papierowe szkice wyrzuca się ze znacznie mniejszymi oporami niż te towarzyszące kasowaniu „elektronicznej" dokumentacji. W następnej ramce opisany został jednak ważny wyjątek od tej zasady.

Narzędzia do „zwinnego modelowania"

Niniejsza książka jest z założenia praktycznym wprowadzeniem do XP, byłaby więc niekompletna bez wskazania, jak *w praktyce* prowadzić „zwinne modelowanie".

Zacznijmy od rzeczy najprostszych. Najlepszymi narzędziami do rysowania diagramów są: kartka i ołówek (pisak) albo tablica i kreda. Łatwo narysować, jeszcze łatwiej zetrzeć (wyrzucić).

„A co z programami do modelowania?" — spytasz zapewne. Tu będę zdecydowanie stronniczy; pracuję dla TogetherSoft i bardzo podoba mi się ich zwinne podejście do modelowania, i do tworzenia oprogramowania w ogólności. Ich naczelną zasadą jest nieustanna synchronizacja modelu z kodem — co wydaje się oczywiste, jeżeli w ogóle model ma być graficznym odzwierciedleniem kodu. „Projekt ukrywa się w kodzie" głosi jedna z zasad XP i tę zasadę koledzy niezmiennie realizują. Za pomocą odpowiednich narzędzi mogą oni w bezpośredni sposób dokonywać różnorodnych „operacji" na modelu — dodawać i usuwać klasy oraz zmieniać ich nazwy; dodawać i usuwać atrybuty i metody; w wizualny sposób tworzyć powiązania itd.; to wszystko znajduje natychmiastowe odzwierciedlenie w kodzie — rysowanie diagramów jest więc jednocześnie programowaniem. Narysowanie na przykład diagramu sekwencji skutkuje automatycznym wypełnieniem ciała odpowiednich metod.

I vice versa — wszelkie zmiany w kodzie odzwierciedlane są automatycznie w diagramach klas, i (na żądanie) w diagramach sekwencji.

Oczywiście, oprócz tych wyrafinowanych narzędzi koledzy posiadają także te bardziej podstawowe (edytor, debugger i in.).

Istnieją również innego rodzaju narzędzia do tworzenia diagramów — takie, które nie są ściśle zintegrowane z generatorami kodu. Niektóre z nich realizują wręcz sprzężenie w drugą stronę — umożliwiają narysowanie ślicznych diagramów na podstawie *istniejącego* kodu. Nie zapewniają one oczywiście żadnej synchronizacji z odnośnym kodem — po zmianie tego ostatniego narysowane diagramy mogą więc nadawać się tylko do wyrzucenia.

Konkluzja

Niniejszy rozdział poświęcony był zagadnieniu projektowania na gruncie XP. W zgodnej, lecz błędnej opinii niektórych, XP w ogóle obywa się bez projektowania; nawet na konferencji XP2001 słyszałem stwierdzenia w rodzaju „oni tak lubią XP, bo dzięki niemu wcale nie muszą projektować". Tymczasem projektowanie jest integralną częścią XP, z tą jednak różnicą, iż w przeciwieństwie do dużych projektów „a priori" jest to „zwinne" projektowanie *przyrostowe*, polegające na ciągłym korygowaniu (refaktoryzacji) projektu stosownie do bieżących potrzeb. Unika się dzięki temu opasłej dokumentacji i nieaktualnych diagramów. W projekcie uwzględniane są tylko te rzeczy, które okazują się rzeczywiście niezbędne.