Inżynieria oprogramowania. Jak zapewnić jakość tworzonym aplikacjom

Autorzy: Bogdan Bereza-Jarociński, Bolesław Szomański
ISBN: 978-83-246-1948-1
Format: 158×235, stron: 328

Twórz rozwiązania najwyższej jakości!

- Ile kosztuje najwyższa jakość?
- Jak ją zapewnić?
- Jakie znaczenie ma bezpieczeństwo informacji?

Inżynieria oprogramowania jest niezwykle obszerną dziedziną wiedzy, zajmującą się wszelkimi aspektami produkcji oprogramowania. Obejmuje zagadnienia takie, jak analiza, projektowanie czy też wdrożenie systemu informatycznego. Jeżeli kiedykolwiek spotkałeś się z oprogramowaniem miernej jakości, niewątpliwie na którymś z etapów jego produkcji pojawił się problem. Jak temu zapobiec?

O tym właśnie traktuje ta książka. Dowiesz się z niej, jak unikać błędów, tak aby oprogramowanie, które wytworzysz, prezentowało najwyższą jakość! Poznasz podejście do kwestii jakości w czasach współczesnych oraz zobaczysz, jak temat ten był rozumiany wcześniej. Zdobędziesz wiedzę na temat miar używanych w inżynierii oprogramowania oraz najefektywniejszych metod i technik jego wytwarzania. Autor przedstawi Ci również narzędzia, które sprawią, że Twoje rozwiązania staną się jeszcze lepsze. Ponadto zobaczysz, jak ważne są tematy związane z bezpieczeństwem informacji. Warto podkreślić, że styl tej książki łączy lekkość i przyjemność lektury z poważną tematyką poruszanych w niej zagadnień.

- Jakość integralna
- Zarządzanie ryzykiem
- Zarządzanie procesami
- Cena jakości
- Spojrzenie na jakość wczoraj, dziś i jutro
- Zarządzanie jakością
- Socjologiczne i antropologiczne podejście do jakości
- Certyfikacja w inżynierii oprogramowania
- Najlepsze metody oraz techniki
- Dostępne narzędzia, automatyzacja testów
- Istota bezpieczeństwa informacji

Spraw, aby Twoje aplikacje były najwyższej jakości!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Zarządzanie procesami

4.1. Zarządzanie jakością — władza i zgiełk

Tak jak Wenus — podobno — wyłoniła się z morskiej piany, tak z chaotycznej bieganiny, nerwowych zebrań, nadgodzin programistów, rozpaczliwej krzątaniny testerów, zszarganych nerwów kierownika projektu oraz gróźb zniecierpliwionego klienta ma się wyłonić Ona: aplikacja-marzenie. Bezbłędna. Zaspokajająca wszystkie, nawet najskrytsze marzenia klienta. Idealna.

Ważną rolę w tym procesie odgrywa testowanie. To test powinien ostrzec: „Panowie, mieliśmy budować Wenus, a na razie widzimy tutaj pięciogłowego wielbłąda!". Test przypomni, ze bogini piękności powinna mieć dwie, nie zaś trzy nogi. Test policzy palce u rąk i zawoła, że cztery palce uchodzą w komiksach, ale nie w rzeczywistości.

O ile nietrudno odróżnić pięciogłowego wielbłąda od Wenus, o tyle błędy oprogramowania nie zawsze są oczywiste i rzucające się w oczy. Zdemaskowanie ich wymaga skrzętnej pracy, wspólnego wysiłku wielu osób, którymi ktoś musi zarządzać i kierować. Jak? — o tym właśnie będzie mowa w dalszej części rozdziału.

Jak opanować stado bezgłowych kur, czyli zarządzanie konfiguracją

Zgłoszenie błędu — dokładny opis objawów i okoliczności awarii, sporządzony przez testera sporym nakładem pracy po to, by ułatwić programiście znalezienie i zlikwidowanie przyczyny awarii. Programista bardzo się dziwi: przecież ten błąd został usunięty już dwa tygodnie wcześniej! „Pewnie użyłeś złej wersji"—— powiada testerowi. „Ależ skąd, główne okno dialogowe wyświetliło numer najnowszej wersji, Z15" — oponuje tester. „Tak, ale to wersja programu głównego. Ten moduł mógł mieć inną wersję!". Sprawdzają obaj. Okazuje się, że adres *0xA1F0* zawiera wartość *0xE*, a więc wersja

numer czternaście feralnego modułu. Tester poci się i łączy program ponownie, tym razem z najnowszą wersją. Awaria nie pojawia się więcej: dobrze. Niestety, po dwóch tygodniach powraca! Co się stało? Po pół dnia dochodzeń udaje się ustalić, że nowo zatrudniony programista przez pomyłkę, łącząc program, znowu posłużył się starą wersją feralnego modułu…

Brzmi to znajomo? Oczywiście. Mamy tu do czynienia z klasycznymi symptomami niedostatków zarządzania konfiguracją.

No, ale co to ma wspólnego z zarządzaniem testami? Jak w opisanym przykładzie — bardzo wiele. System czy program (zwłaszcza niezbyt skomplikowany) może się niekiedy udać zbudować — kosztem pewnego czasochłonnego zamieszania — mimo braków w zarządzaniu konfiguracją. Natomiast zapewnienie jakości bez dobrze funkcjonującego zarządzania konfiguracją jest zwykle niemal bezużyteczne. Zidentyfikowane przez testerów awarie okazują się albo dotyczyć nieaktualnej wersji, albo wymagają detektywistycznej pracy, aby znaleźć ich przyczynę w chaosie splątanych wersji poszczególnych modułów systemu. Marnuje się w ten sposób wiele czasu i wysiłku, przez co test tylko w ograniczonym stopniu dostarcza swego najważniejszego produktu: informacji pozwalającej na znajdowanie i usuwanie błędów.

Z tego właśnie powodu często zespół testujący, a nie cały projekt informatyczny, jest gorącym zwolennikiem uporządkowania źle działającego zarządzania konfiguracją. Nie jest to dobre rozwiązanie, ale o wiele lepsze niż dobrowolne oddanie się w ręce chaosu, marnotrawstwa i bałaganu. Choć więc nie chodzi tu o testowanie *sensu stricto*, niejednemu kierownikowi zespołu testującego przyjdzie się z tą problematyką zmierzyć i warto sobie z tego zdawać sprawę. Jak konkretnie się to robi: zarządzanie i kontrolę wersji, budowę konfiguracji podstawowych (*baselines*) — to są już zagadnienia na osobny rozdział.

Rozmawiała gęś z prosięciem: raporty i śledzenie błędów

Kiedy tester natknie się na awarię będącą symptomem tkwiącego w programie błędu, fakt ten niesie w sobie dwa rodzaje informacji. Po pierwsze, ilość znajdujących się w programie błędów jest podstawową miarą jego jakości, a więc kluczową wielkością, którą należy wziąć pod uwagę, dokonując decyzji dotyczących wdrożenia, wprowadzenia do produkcji czy dostarczenia klientom nowej wersji programu. Po drugie, zaobserwowana awaria pozwala zwykle zidentyfikować będący jej przyczyną błąd, usunąć go i w ten sposób podnieść jakość programu.

Ani w jednym, ani w drugim przypadku nie wystarczy, by ta wiedza pozostała w głowie testera. Trzeba ją przekazać programiście, aby rozpoczął poszukiwania przyczyny awarii, oraz kierownikowi projektu, aby mógł sporządzić statystyki błędów i oszacować bieżącą jakość konstruowanego systemu. Nawet jeśli projekt jest jednoosobowy, nie zawsze daje się wszystko zapamiętać i prowadzenie notatek na temat znajdowanych i usuwanych błędów pozwala na uniknięcie pomyłek.

Tym celom — przekazywaniu oraz gromadzeniu informacji o awariach i błędach — służą tak zwane *raporty* albo *zgłoszenia błędów*.

Niektóre traktujące o testowaniu źródła (m.in. tłumaczona na język polski książka amerykańskiego autora Rona Pattona *Testowanie oprogramowania*[1]) poświęcają wiele miejsca udzielaniu rad, jak powinien postępować tester, aby dopilnować, żeby znaleziony błąd rzeczywiście został potraktowany poważnie i naprawiony. Takie podejście wydaje się być stawianiem sprawy na głowie. Po pierwsze, tester ma inne zajęcia niż zastępowanie — niefrasobliwego widać — kierownika projektu i ściganie programistów. Po drugie, taka sytuacja stwarza realne zagrożenie sprowokowania konfliktów między testerami a konstruktorami. Po trzecie wreszcie, tester nie musi mieć i zwykle nie ma pełnej wiedzy potrzebnej do prawidłowej oceny wagi znalezionego błędu. Do tego konieczna jest — zależnie od rodzaju błędu — jeszcze wiedza na temat struktury i priorytetów wymagań, potrzeb klienta, architektury systemu. Nie jest wcale oczywiste, że każda awaria wymaga natychmiastowego rzucenia wszystkich dostępnych środków w celu jej rozbrojenia i usunięcia! To zależy między innymi od związanego z nią ryzyka. Do oszacowania ryzyka nie wystarczy zwykle jedna osoba: konieczna jest współpraca wielu uczestników projektu, którą umożliwiają właściwie wykorzystane raporty błędów.

Zorganizowanie procedur zgłaszania i śledzenia błędów jest jednym z podstawowych zadań kierownika testów.

Krajobraz przed bitwą: planowanie testów, analiza ryzyka

Jak powiedział generał, a później prezydent Eisenhower, wprawdzie planowany przebieg wydarzeń nigdy się nie sprawdza, ale mimo to ten dowódca, który planował najstaranniej, ma największe szanse poradzenia sobie z (niezaplanowaną) sytuacją na polu bitwy.

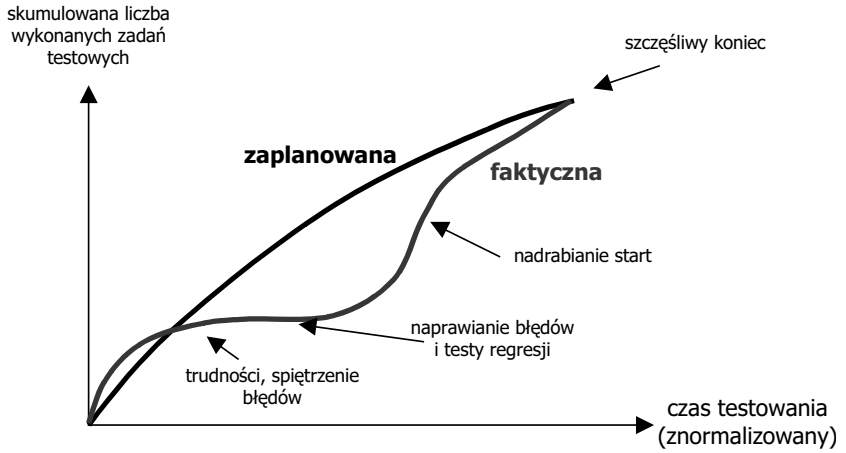
To samo dotyczy testowania. Wiadomo z góry, że dostawa do testu systemowego będzie opóźniona — w porównaniu z planem — o dwa miesiące, natomiast data dostawy do klienta nie ulegnie zmianie, przez co na test systemu, zamiast przewidzianych dziesięciu, pozostaną ledwo dwa tygodnie. Wiadomo, że jakość pierwszych dostaw będzie taka, że większość czasu trzeba będzie poświęcić na podnoszenie zawieszającego się systemu, a nie na wykonywanie przypadków testowych. Oczywiste jest też, że znajdowane błędy spowodują niezaplanowany wzrost ilości dostarczanych do testowania wersji programu, przez co czas poświęcony na ich instalację i konfigurowanie oraz na testy regresji wzrośnie — w porównaniu z planem — dramatycznie. Wreszcie wiadomo, że proces odpluskwiania (ang. *debugging*) odciągnie pewną ilość testerów na pewien czas od testowania, a ponadto środowisko testowe będzie — w niezaplanowanych wymiarach — zablokowane przez programistów usiłujących odtworzyć awarię i zlokalizować jej przyczynę.

Planując, że wydarzą się wszystkie te niezaplanowane historie, mamy realne podstawy, by poradzić sobie z wyzwaniem, jakim jest zarządzanie testami!

[1] Nakład obecnie wyczerpany — wrzesień 2008.

Husaria kontra pruska piechota: jak nie stracić impetu, nie tracąc głowy?

Impet jest w testowaniu ważny, ale musi to być impet kontrolowany, w przeciwnym razie może nas sprowadzić na manowce. Śledzimy liczbę wykonanych przypadków testowych i porównujemy z zaplanowanymi — w ten sposób ewentualne opóźnienie wyjdzie na jaw od samego początku, a nie dopiero wtedy, kiedy narośnie do katastrofalnych rozmiarów. Śledzimy liczbę otwartych, zgłoszonych błędów — w ten sposób możemy próbować oszacować ilość pozostałych jeszcze błędów, które zapewne wyszłyby na jaw w trakcie dalszego testowania systemu, dzięki czemu w każdej chwili mamy do dyspozycji dane pozwalające odpowiedzieć na nieuniknione pytanie: co kierownik testów sądzi o tym, żeby dostawa do klienta miała miejsce już pojutrze?



Rysunek 4.1.1. *Śledzenie procesu przebiegu testów*

Dostrzegłszy niebezpieczne, narastające rozbieżności między planem a rzeczywistością, kierownik testów ma do dyspozycji pięć typów środków zaradczych:

- Zmiana harmonogramu testów — odroczenie zakończenia i terminu dostawy do klienta.
- Zmiana kryteriów jakości — obniżenie poprzeczki, dopuszczenie do użytku systemu mniej przetestowanego albo mającego większą ilość nierozwiązanych błędów.
- Wykorzystanie do testowania większej ilości osób, testowanie równoległe.
- Zamiana funkcjonalności — dostarczony system nie będzie zawierał wszystkich wcześniej planowanych funkcji.
- Podniesienie wymaganej jakości dostaw do testu systemowego — to umożliwi sprawniejsze testowanie i przerzuci część działań na niższe poziomy (testy komponentów, integracyjne).
- Ponadto często stosowanym środkiem jest — kiedy gwałtownie narasta ilość zarejestrowanych zgłoszeń błędów — czasowe zawieszenie wykonywania nowych testów po to, by dać programistom szansę na usunięcie spiętrzenia

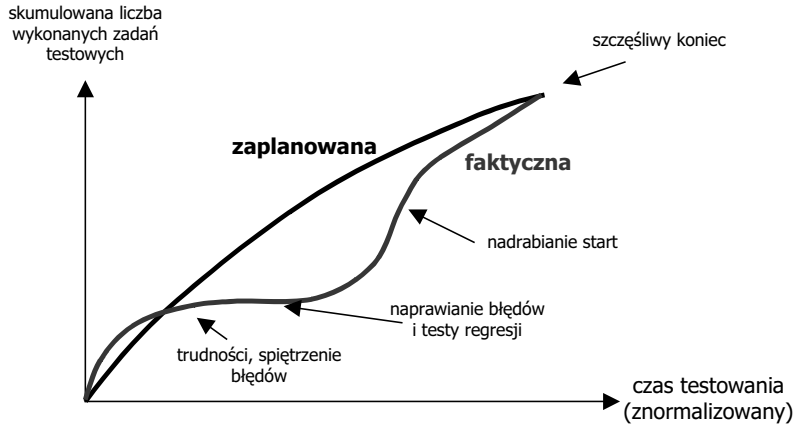
i naprawienie jak największej liczby błędów. W tym czasie zespół testowy poświęca się wyłącznie testowaniu powtarzalnemu dostaw zawierających kolejne poprawki.

Krajobraz po bitwie: czy można wypuścić produkt już jutro?

Decyzja o tym, czy można już zakończyć testowanie i wypuścić, dostarczyć albo rozpocząć wdrażanie programu, jest *de facto* decyzją biznesową, nie techniczną. Stosowana w niektórych przedsiębiorstwach zasada, że kierownik testów podpisuje zakończenie testów i niejako tym samym własną głową gwarantuje dostateczną jakość produktu, jest absurdem. Testowanie nie jest na dobrą sprawę zakończone nigdy, zawsze pozostaje — z podpisem kierownika czy bez niego — pewne ryzyko, że w programie pozostały niezauważone błędy.

Nie znaczy to jednak, że testować trzeba w nieskończoność, bo z drugiej strony czai się przecież ryzyko opóźnienia, kar umownych, niezadowolonych klientów, utraty udziałów w rynku na rzecz szybszych czy odważniejszych konkurentów. Analiza ryzyka i podjęcie decyzji jest w 100% zadaniem dla kierownictwa lub sponsorów projektu, ewentualnie dla działu marketingu. Test ma natomiast za zadanie dostarczyć decydentom jak najprecyzyjniejsze dane dotyczące ryzyka technicznego w oparciu o dotychczasowe wyniki testowania.

Istnieje wiele kryteriów oszacowania jakości produktu w oparciu o rezultaty testów. Bierze się na przykład pod uwagę, jaki procent zadań testowych został dotychczas wykonany, ile pozostało otwartych (nierozwiązanych) zgłoszeń błędów itd. Interesującą metodą jest technika oszacowania ilości pozostałych jeszcze w programie *nieznanych* błędów na podstawie funkcji najlepiej pasującej do krzywej określającej skumulowaną ilość dotychczas znalezionych błędów. Wyjaśnienie — na ilustracji poniżej.



Rysunek 4.1.2. *Szacowanie liczby pozostałych defektów*

Oczywiście istotność takich oszacowań zależy od liczby oraz jakości wykonanych testów. Do ich oceny służą rozmaite *miary pokrycia* (np. wymagań, funkcji, kodu).

Obdzieranie poległych, czyli jak być mądrym po szkodzie

Projekt zakończony, produkt sprzedany, kod i dokumentacja złożone w archiwum i przekazane do działu zajmującego się serwisem — czy to już koniec pracy? Otóż nie, bo z danych uzyskanych w trakcie testowania można jeszcze niejedną ciekawą informację wycisnąć. Wprawdzie na poprawę jakości wytworzonego przez zakończony projekt produktu informatycznego już za późno, nie da się także podwyższyć jakości decyzji, które już zapadły, ale można uzyskać wiedzę pozwalającą być może kolejne projekty poprowadzić lepiej i sprawniej.

Bogatym źródłem wiadomości jest baza danych z raportami błędów. Można na przykład szczegółowo zanalizować pewną liczbę losowo wybranych raportów i spróbować odpowiedzieć na pytanie, jaka była pierwsza przyczyna zaistnienia danego błędu? Czy były nią niejasno sformułowane wymagania, czy niedostateczna znajomość języka przez programistów, czy niedociągnięcia organizacyjne?

Warto też przyjrzeć się statystykom raportów błędów. Kiedy pojawiło się ich najwięcej? Jaki był czas naprawy błędu? Ile raportów okazało się fałszywych? Odpowiedzi na te pytania niejednokrotnie pozwolą zidentyfikować słabe punkty w procesach i procedurach projektów lub niedostatki organizacyjne w przedsiębiorstwie.

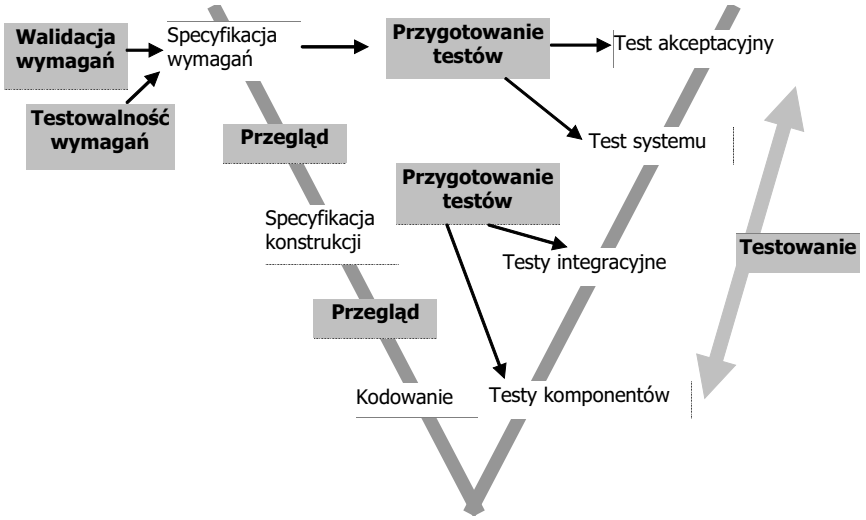
Różne formy organizacji testowania

Nie zawsze jedyną i najlepszą formą organizacji testów jest stworzenie osobnego *zespołu testowego*. Zależnie od charakteru projektu, typu produktu, przyjętej metodyki wytwarzania korzystne może okazać się zastosowanie innych rozwiązań organizacyjnych.

- Programiści sami testują własny kod. Metoda często stosowana w testach modułowych (jednostkowych, komponentów). Jej wady są oczywiste.
- Testowanie koleżeńskie (ang. *buddy testing*): programiści nawzajem testują swój kod. Stosowane między innymi w popularnym ostatnio „Programowaniu Ekstremalnym" (XP, *Extreme Programming*).
- Tester (lub testerzy) są członkami zespołu programistów, podlegają kierownikowi zespołu lub projektu.
- Osobny zespół testujący mający własnego kierownika.
- Osobny dział w przedsiębiorstwie zajmujący się pewnymi rodzajami testów.
- Outsourcing testów do innego przedsiębiorstwa: stosowane wówczas, gdy wymagana jest niezależna certyfikacja systemu oraz gdy niezbędne jest skomplikowane i kosztowne środowisko testowe (np. w testowaniu konfiguracji, testowaniu zgodności z wymaganiami środowiskowymi itp.).
- Wybór właściwej organizacji testowania jest ważnym zadaniem dla kierownika projektu. Warto pamiętać, że w większych projektach kilka różnych form organizacyjnych może istnieć jednocześnie, na przykład testowanie koleżeńskie na poziomie testów komponentów, odrębny zespół do testu systemowego, outsourcing w celu uzyskania niezależnej certyfikacji.

Kiedy zaczynać, kiedy skończyć?

Jak zwykle bywa — dobrze wiemy. Jak powinno być — zwięźle opisuje rysunek 4.1.3. Czynności wykonywane przez zespół testowy napisane są tłustym drukiem na jasnoszarym tle.



Rysunek 4.1.3. *Przegląd modelu „V"*

4.2. Znowu ten pośpiech — jak szybko ocenić jakość aplikacji?

Pośpiech w informatyce

Zrobić cokolwiek **szybko**? Znowu ten pośpiech. Znane jest przecież porzekadło: „co nagle, to po diable" i niezliczone przykłady sytuacji, kiedy zabrakło czasu i środków, aby coś wykonać dobrze, ale znalazło się potem i jedno, i drugie, aby to coś wielokrotnie poprawiać. Informatyka to branża cierpiąca od swego powstania na syndrom czarodziejskiej plasteliny. Kilkadziesiąt lat temu udało się ludziom spełnić swe odwieczne marzenie i znaleźć substancję, z której daje się szybko i łatwo zbudować wiele najrozmaitszych rzeczy: a to system bazodanowy, a to telefonię komórkową, a to wbudowany układ sterujący do pralki automatycznej. Figurki lepione z naszej czarodziejskiej plasteliny — instrukcji mikroprocesora — rzeczywiście można tworzyć zadziwiająco szybko w porównaniu z przedmiotami z drewna, metalu czy betonu, a ponadto można je potem względnie łatwo poprawiać bez potrzeby burzenia całości, jeśli coś się nie całkiem uda. Ludzikowi z plasteliny można nawet, kiedy już jest gotowy, oderwać nogę i zastąpić ją inną, lepszą, ale też wygląda on potem jak… ludzik z plasteliny.

Programowanie narażone jest na nieustanną **pokusę bylejakości i pośpiechu**, których skutkiem jest bardzo często albo fatalna jakość aplikacji, albo lekceważenie użytkownika i jego potrzeb, przez co świat zapełniają zawodne i pokraczne, niewygodne w obsłudze twory z plasteliny. Po co zbierać i analizować wymagania, skoro można zacząć budować od razu, a potem, w razie czego, wszystko się przerobi? Po co starannie projektować system, skoro można od razu zacząć kodowanie, a potem jakoś się te, niepasujące do siebie, części poskleja w całość? Po co dbać o jakość projektu, skoro w bałaganie też daje się pracować, i po co wysilać się na produkty dobrej jakości, skoro czarodziejska plastelina pozwala na pozór bezkarnie poprawiać, sztukować, zaizolować kawałkiem dętki, przymocować drutem?

Miło jest sobie pozrzędzić, ale z drugiej strony nie można zaprzeczyć, że to dzięki systemom informatycznym **dzisiejszy świat ogromnie przewyższa ten sprzed lat trzydziestu** i czterdziestu pod względem możliwości, dobrobytu, bezpieczeństwa i organizacji, cokolwiek na ten temat sądzą rozmaici zwolennicy powrotu do jaskiń czy wręcz na drzewa.

Ponadto, szydząc sobie z typowego projektu informatycznego: drwala, który nie ma czasu porządnie naostrzyć siekiery, bo tak bardzo się spieszy z rąbaniem drewna, nie sposób przecież zapomnieć o zagrożeniach z przeciwnej strony: drwalach tak zajętych ostrzeniem siekiery, że nie mają czasu na ścinanie drzew. Czynniki psychologiczne powodują, że chętnie — **uchylając się przed naprawdę trudnymi wyzwaniami** — uciekamy w rytualizację, mnożenie zbędnej dokumentacji, manię zebrań i posiedzeń, wiarę w rzekomo uzdrowicielską moc procedur, procesów, poziomów dojrzałości i sprawności, duszących prawdziwą kreatywność i skuteczność.

Czy nie ma drogi pośredniej między jedną a drugą skrajnością? Jest, oczywiście — to **pośpiech kontrolowany**, gdzie umiejętność i wprawa pozwalają poruszać się szybko, lecz pewnie, a ścieżki na skróty niekoniecznie prowadzą na manowce.

Pomiary w pośpiechu

Warunkiem skutecznego pośpiechu kontrolowanego jest umiejętność **nadzoru w biegu**, tak żeby zakręt móc przejść na piszczących oponach, ale z niego nie wylecieć, pokonując zaś na skróty bezdroża, orientować się zręcznie za pomocą mapy, kompasu, zegarka i bystrych oczu — i nie zabłądzić.

Nie jesteśmy w stanie kontrolować tego, czego nie umiemy zmierzyć. Ale *pomiar* nie jest w informatyce słowem lubianym — nawet poddany mi przez Redakcję tytuł tego artykułu omija je, zastępując niebudzącym lęku słowem *ocena*. Choć jako specjalista w branży nie raz spierałem się przy piwie, czy to testowanie, czy też utrzymanie oprogramowania jest bardziej niesłusznie lekceważone w praktyce naszego przemysłu, wydaje się, że palma pierwszeństwa należy się jednak pomiarom. Dobry kierowca rajdowy nie musi wysiadać z samochodu i mierzyć promienia skrętu taśmą tylko dzięki temu, że wprawa pozwala mu mierzyć bez przerywania jazdy. Przewodnik, na pozór bez wysiłku wyprowadzający przez gęste krzaki wprost na zamierzony punkt, nie wlecze za sobą wielokilometrowej nici Ariadny tylko dlatego, że nieustannie podczas marszu

mierzy przebytą odległość, kierunek, nachylenie terenu. W przemyśle informatycznym chętnie udajemy kierowców Formuły 1 oraz dzielnych przewodników, nie mając niezbędnych po temu umiejętności **mierzenia**.

Brak umiejętności sprawnego mierzenia uniemożliwia **zarządzanie ryzykiem**, zastępując je unikaniem ryzyka — lub bezsensowną brawurą. Unikanie ryzyka w inżynierii oprogramowania rodzi projekty sztywne, zbiurokratyzowane, nieskuteczne, omijające właściwe wyzwania. Bezsensowna brawura oznacza fanfaronadę przy wyznaczaniu celów, środków i terminów, po czym… To, co zdarza się potem, także można oczywiście zmierzyć. Odpowiednią miarą, nie do końca jeszcze uznaną przez fizyków, jest „och-nie-sekunda" (ang. *oh-no-second*), stosowana do określenia czasu upływającego od chwili, gdy się zorientowaliśmy, że popełniliśmy NAPRAWDĘ DUŻY BŁĄD (np. klikając „wyślij do wszystkich" na koniec maila pełnego wspomnień z bardzo gorącej nocy).

O zarządzaniu ryzykiem i o skutecznych pomiarach napiszę wkrótce, jak mi czas i Redakcja pozwolą. Na razie pora przejść do sedna: jak szybko ocenić jakość produktu, czyli **jak mierzyć w biegu?**

Precz z grzybami

Wyobraźmy sobie, że pełnimy funkcję Naczelnika Jakiejś Jednostki Administracyjnej. Najnowsza polityka rządu kładzie szczególny nacisk na oczyszczanie lasów z grzybów. Dlaczego — nieważne, ale nietrudno sobie wyobrazić… Grzyby przecież bywają trujące, a ludność musi być chroniona przed zagrożeniami. Następnie jesteśmy nowoczesnym europejskim krajem, a grzyby nie mają witamin, nie poddają się racjonalnej hodowli i wzbudzają — jako pozbawione chlorofilu — zastrzeżenia wojujących środowisk wegetariańskich. Poza tym grzyby to pasożyty, co kłóci się z ideami solidaryzmu społecznego (lub jest ich złośliwą karykaturą), a ich preferencje seksualne też są — zdaje się — nad wyraz nieprawomyślne. Niechęć do grzybów ma wyraźnie ponadpartyjny charakter, więc lasy mają być odgrzybione, a za dwa dni przyjedzie — o czym dał nam cynk kolega z Sąsiedniej Jednostki Administracyjnej — Nadzwyczajna Komisja, żeby sprawdzić stan odgrzybienia naszego lasu podmiejskiego. Tak więc mamy SZYBKO OCENIĆ JAKOŚĆ LASU!

Nie muszę dodawać, że dotąd w tej sprawie nie zrobiono nic. **Gdyby las był już wcześniej systematycznie odgrzybiany, nie byłoby paniki**. Oczywiście identycznie jest z potrzebą *szybkiej* oceny jakości aplikacji. Gdyby projekt był od początku prowadzony porządnie, jakość aplikacji byłaby po prostu znana — realizowana i mierzona cały czas. Cóż, jednak świat jest niedoskonały, więc idziemy mierzyć w pośpiechu.

Grzybobranie

Zasada podstawowa — nie da się trafnie ocenić stanu zagrzybienia lasu, nie wysyłając tam ludzi odpowiednio zmotywowanych, umiejących szukać grzybów! Można, rzecz prosta, wysłać do lasu krótkowidza, który na grzybach się nie zna, dla całkowitej pewności mówiąc mu złowieszczym głosem: „Mam nadzieję, że przyniesie mi pan DOBRE

wiadomości!". Wtedy ocena jakości lasu będzie wprawdzie odpowiednio szybka, ale całkowicie nietrafna, a nie o to chyba nam chodzi. Śmiejąc się z takiej metody, nie zapominajmy, że dokładnie tak odbywa się często próba szybkiej oceny jakości aplikacji — jeśli nie wykonują jej fachowi testerzy, odpowiednio nagradzani za przynoszenie wieści o błędach, wynik pomiaru jest bezwartościowy.

Dobry grzybiarz szuka grzybów tam, gdzie spodziewa się je znaleźć. Wykorzystując sobie tylko znane intuicje, wie gdzie zwykle rosną koźlaki, gdzie rydze, a gdzie opieńki, dzięki czemu przynosi ich pełne kosze. Tak samo doświadczony tester wykorzystuje swe wcześniejsze doświadczenia, aby szukać błędy ocenianej aplikacji tam, gdzie spodziewa się je znaleźć. Jak rydze lubią rosnąć pod świerkami, tak błędy lubią się na przykład gromadzić w pobliżu wartości krańcowych, na granicach przedziałów, i dobry tester tam właśnie będzie ich szukał. Dalej, błędy chętnie dojrzewają w miejscach odludnych, których nikt od dawna nie testował, bo kod jest tak zawiły, że lepiej go nie ruszać. Wiemy też, że obecność kilku błędów zwykle oznacza, że jest ich tam o wiele więcej — wynikają bowiem z tych samych błędów projektowania. Kolejną regułę streszcza powiedzenie „gdzie kucharek sześć…" — jeśli kod był wielokrotnie zmieniany, jeśli modyfikowało go wielu programistów — warto poszukać błędów. Zasad jest wiele, a profesjonalni testerzy powinni je znać.

Wróćmy do podmiejskiego lasu. Doświadczony grzybiarz szuka grzybów tam, gdzie zwykle rosną, ale niekoniecznie tam, gdzie będzie ich szukać nasza Nadzwyczajna Komisja. Jeśli członkowie Komisji są łagodnymi, leniwymi grubasami, zadowolą się pobieżnym sprawdzeniem bezpośredniej okolicy wygodnych ścieżek i tam właśnie — wbrew instynktowi grzybiarza — trzeba przeszukać teren szczególnie starannie. Jeśli w skład Komicji wchodzą ambitne, młode wilki, będą się starać wykazać, szukając w nietypowych miejscach — niechże więc grzybiarze strwożonego Naczelnika Jednostki na wszelki wypadek przepatrzą miejsca pod kamieniami, wśród gęstych krzaków czy w inne, do których podejrzewamy, że chętnie skierują się młode wilki.

Przenosząc się na chwilę z powrotem w dziedzinę oceny jakości aplikacji, należy oceniać przede wszystkim to, czym najczęściej posługują się użytkownicy końcowi. Skoro nie ma się do dyspozycji dostatecznie długiego czasu, aby ocenić wszystko, warto skoncentrować się na obszarach, gdzie — z racji intensywnego użytkowania — prawdopodobieństwo awarii, jeśli są tam błędy — jest najwyższe. Dzięki temu jakość aplikacji — mierzona średnim czasem między awariami — będzie wyższa, oczywiście przy założeniu, że znalezione podczas oceniania błędy będą też usuwane.

Grzyb grzybowi nierówny. Doniesiono Naczelnikowi Jednostki, że Komisja jest szczególnie uczulona na muchomory sromotnikowe, pewnie ze względu na ich kształt. Dlatego naczelnik uczula swoich grzybiarzy, aby szukali — wbrew swoim naturalnym, grzybiarskim instynktom — właśnie sromotników. Tak samo przy szybkiej ocenie jakości aplikacji koncentrujemy się na tych błędach, których skutki z punktu widzenia użytkowników są szczególnie złe, a mniej czasu poświęcamy błędom, o których wiadomo, że — jeśli nawet gdzieś są — nie będą dla użytkowników zbyt dotkliwe.

W środku lasu jest ostaniec — pionowa, kilkunastometrowa skała. Może na jej szczycie też rosną jakieś grzyby, a któryś z członków Komisji uprawia sporty ekstremalne i tam się wdrapie? Może, ale z drugiej strony, spenetrowanie wierzchołka skały wymagałoby

drabin, straży pożarnej, kto wie, czy nie helikoptera, co pochłonęłoby znaczną część środków dostępnych na szybką ocenę jakości lasu, przez co gorzej zostałyby spenetrowane jego łatwiej dostępne rejony. W tej sytuacji chyba rozsądniej jednak będzie zostawić w spokoju skałę. Tłumaczyć się potem, że w lesie wprawdzie jest pełno grzybów, ale za to wolny od nich jest trudno dostępny ostaniec — to nie będzie brzmiało dobrze.

Stąd wypływa kolejny wniosek dla oceniania jakości aplikacji — jeśli mamy ograniczone zasoby, a słowo „szybko" oznacza, że brakuje nam najcenniejszego z nich, czyli czasu — trzeba uwzględnić, na ile trudne i kosztowne są pewne testy, tak żeby dostępne środki rozdysponować raczej równomiernie, a nie tylko w jednym, szczególnie zasobochłonnym obszarze.

Testowanie uwzględniające ryzyko

Powyższe rozważania są streszczeniem podejścia znanego jako testowanie uwzględniające ryzyko (ang. *risk based testing*). Jeśli jakość musimy ocenić szybko, testujemy (oceniamy, mierzymy) przede wszystkim to, co najważniejsze, uwzględniając cztery kluczowe parametry:

- **Prawdopodobieństwo błędu** — szkoda czasu, aby szukać błędów tam, gdzie być może ich nie ma.
- **Konsekwencje awarii —** przy ocenie jakości należy szukać raczej awarii katastrofalnych niż niegroźnych, kosmetycznych.
- **Prawdopodobieństwo zastosowania** — trafniej ocenimy jakość, biorąc pod uwagę przede wszystkim to, czym użytkownicy posługują się na co dzień, niż to, z czego korzystają raz do roku lub wcale.
- **Łatwość testowania** — przy szybkiej ocenie warto też wziąć po uwagę, by — o ile nie chodzi o awarie katastrofalne — raczej unikać wikłania się w próby oceny tego, czego pomiar jest zbyt kosztowny i czasochłonny.

Jakie to łatwe...

Warum einfach? Kompliziert geht es auch! — powiadają nasi zachodni sąsiedzi. Popełniam zdaje się błąd, przedstawiając łatwo zrozumiałą przypowieść o grzybach zamiast epatowania licznymi skomplikowanymi nazwami i trzyliterowymi skrótami. Przeczytawszy wstępną wersję artykułu, ktoś powiedział „to całe testowanie jest w gruncie rzeczy bardzo proste". Owszem, jeśli wiemy, gdzie rosną grzyby (znamy się dobrze na informatyce, na testowaniu i na projektach informatycznych), jeśli znamy dziedzinę zastosowania (ocena częstości zastosowania oraz konsekwencji awarii) oraz technologię testów (ocena łatwości testowania). Przydaje się też niezła znajomość technik pomiaru oraz analizy ich wyników, trochę statystyki… POZA TYM cała reszta to rzeczywiście odrobina zdrowego rozsądku.

Bilet do Davos

Całe kosze usuniętych z lasu grzybów wywieziono daleko — czy można spokojnie oczekiwać inspekcji? Czy może mimo wszystko lepiej zarezerwować dla siebie i rodziny miejsca w samolocie do Szwajcarii i skromny apartament w Davos, na wypadek gdyby Nadzwyczajna Komisja jakiś duży grzyb jednak wykryła?

Trudno powiedzieć — testowanie uwzględniające ryzyko pozwala skutecznie znajdować błędy, nawet w pośpiechu dość trafnie ocenić jakość aplikacji, ale nigdy nie wie się dokładnie, na ile jest ono dokładne: czy czasem — mimo starań — jakaś funkcja nie została pominięta, jakaś część systemu zapomniana? Żeby tę pewność uzyskać, należałoby — wróćmy do historii o lesie — wziąć dokładną mapę lasu, podzielić ją na kwadraty i cały las systematycznie przeczesać. Wprawdzie wiele z miejsc zidentyfikowanych tym sposobem byłoby zupełnie bezsensownych, na przykład plaża, gdzie jako żywo grzyby nie rosną, lub środek bagna, gdzie komisja nigdy nie dotrze, ale jest to koszt systematyczności, cena za ubezpieczenie od skutków przeoczenia lub zapomnienia.

W odniesieniu do oceny jakości aplikacji odpowiednikiem mapy jest model działania lub struktury aplikacji, a odpowiednikiem dzielenia mapy na kwadraty — projektowanie testów z modelu za pomocą algorytmu. To jest konieczne, aby móc ocenić tak zwane pokrycie testowe, a więc oszacować niezawodność wykonanych ocen, ale na to przy ocenie szybkiej nie mamy zwykle czasu. Jedno jest więc pewne — ocena szybka jest zawsze mniej pewna niż ocena spokojna, oczywiście pod warunkiem staranności jednakowej w obu przypadkach.

Jak spieszyć się powoli

Spiesząc się, nie trzeba rezygnować z myślenia. Nie chodzi przecież o to, by wykonywać mnóstwo szybkich, nerwowych ruchów, głośno krzyczeć przez kilka na raz telefonów i pracować — nieefektywnie — po dwadzieścia godzin na dobę, tylko o to, by mimo pośpiechu pozostać skutecznym i skoncentrowanym na celu.

Pogodzić pośpiech ze spokojną systematycznością usiłują metodyki „systematycznego testowania w pośpiechu" (tak celnie określił je dr Lucjan Stapp z Politechniki Warszawskiej w swym wystąpieniu na jednym z zebrań Stowarzyszenia Jakości Systemów Informatycznych).

Jedna z nich to **testowanie eksploracyjne** (ang. *exploratory testing*): zespół technik wspomagających testerów w sytuacji na pozór beznadziejnej, kiedy jednocześnie trzeba uczyć się aplikacji, wykonywać testy i projektować nowe zadania testowe. Za pomocą szeregu **kreatywnych** sposobów — przydatnych także wówczas, gdy pośpiech nie jest aż tak wielki — projektuje się nowe testy na podstawie obserwacji i analizy wyników testów właśnie wykonywanych. Jednym słowem, podejście eksploracyjne poprawia skuteczność testów wtedy, gdy nie ma czasu, by je starannie zaplanować, tylko trzeba strzałem z biodra szybko ocenić jakość aplikacji.

Częściowo odmienne podejście prezentuje **testowanie zwinne** (ang. *agile testing*). Jego podstawa to zasada testowania parami: testerzy pracują w dwuosobowych zespołach, testy wykonują wspólnie. Takie podejście budzi uzasadnione wątpliwości, czy nie jest po prostu dublowaniem kosztów, ale praktyczne doświadczenia sugerują, że faktycznie pozwala na większą skuteczność.

Kilka lat temu głośno było o metodyce **programowania ekstremalnego** (ang. *extreme programming*), gdzie programiści pracują w parach, zamieniając się pisaniem kodu i przygotowywaniem (automatycznych) testów dla tworzonego właśnie kodu. Programowanie ekstremalne postuluje ponadto ograniczenie do minimum tradycyjnej, ciężkiej dokumentacji, bliską współpracę programistów z przedstawicielami klienta, częste — nawet kilka razy na dzień — budowanie całego systemu. Z jednej strony programowanie ekstremalne obiecuje wprawdzie niższą czasochłonność projektów, czym pasuje do naszego tematu; z drugiej strony — postuluje przygotowywanie testów oceniających jakość, jeszcze zanim powstanie kod, co nie do końca już zgadza się z paradygmatem „szybkiej oceny jakości aplikacji".

Wszystkim, którzy szukają cudownych dróg na skróty i sposobów, jak szybko wykonać to, co najlepiej wykonywać spokojnie, dobrze w tym miejscu przypomnieć bajkę o żółwiu i o zającu.

4.3. Po co mierzyć? Miary w inżynierii oprogramowania

Miary często kojarzą nam się z czymś pozytywnym; mówi się: *umiarkowany*, *miarkować*, *znać miarę*, *na miarę*, *miarowo*.

Z drugiej strony, brak miary też chwilami brzmi obiecująco, jak w słowie *bezmierny*.

Miary są niebezpieczne dla tych, którzy usiłują coś ukryć, wolą mętniactwo i niejednoznaczność od precyzji. Miary trzeba dobrze rozumieć, tak więc niektórym ludziom wydają się one niebezpieczne; według Flawiusza to Kain wynalazł „miary i wagi, zmienił ową niewinną i szlachetną prostotę, w jakiej żyli ludzie, póki ich nie znali, w życie pełne oszustwa[2]".

Miary, które znamy na co dzień, wydają się oczywiste, ale nie ma nic oczywistego w tym, aby od prostego „zimno", „średnio" i „ciepło" przejść do skal, gdzie wartości liczbowe przypisywane temperaturze powietrza odnoszone są do długości słupka rtęci zamkniętej w szklanej rurce. Fakt, że istnieją trzy różne, powszechnie stosowane skale temperatury: Fahrenheita, Celsjusza i Kelvina, z których każda ma punkt zerowy przy innej temperaturze, a dwie pierwsze różnią się rozmiarem jednostki skali, wskazuje, że miary nie są niczym oczywistym, że są przyjmowanym częściowo arbitralnie sposobem odwzorowania natężenia pewnych atrybutów rzeczywistości — oj, to brzmi bardzo naukowo, ale ma konkretny, praktyczny sens.

[2] Józef Flawiusz, *Starożytności żydowskie,* I.2.2, wyd. polskie: Warszawa 1962, s. 105 — cytat wg prezentacji Andrzeja Kobylińskiego na Konferencji SJSI, Serock, maj 2005.

Inżynieria — zestaw zdefiniowanych, powtarzalnych technik projektowania, wytwarzania i utrzymania rozmaitych rzeczy — nie może istnieć bez pomiarów. Trudno wyobrazić sobie inżyniera, który nie umie mierzyć długości, ciężaru, napięcia czy natężenia, albo lekarza bez termometru i narzędzi do precyzyjnego pomiaru chemicznego składu krwi. Jedynie inżynieria oprogramowania choruje na brak umiejętności pomiaru nie tylko własnych procesów, ale nawet produktów!

Czego nie można zmierzyć, tego się nie wie

Zapyta ktoś — jakie to ma znaczenie? Czy to kolejna z wielu mód, kolejne gołosłowne twierdzenie, jakoby coś bardzo złego działo się z przemysłem informatycznym, podczas gdy gołym okiem widać, że dzieje się całkiem dobrze?

Dobrze — zależy w odniesieniu do czego. W porównaniu z przemysłem elektronicznym przyrost wydajności w produkcji oprogramowania jest od wielu lat skromny. Produkty programistyczne często okazują się zawodne, a ich spektakularne niepowodzenia — jak na przykład osławione dwuletnie opóźnienie oddania do użytku lotniska w Denver w 1995 roku z powodu przekroczenia terminu dostawy oprogramowania systemu bagażowego — przechodzą do legendy. Wielkie jest zróżnicowanie produktywności programistów w różnych firmach i projektach — według niektórych danych różnice wynoszą nawet 600:1 (sześćset do jednego, to nie jest błąd w druku).

Jednocześnie — tu akurat branża informatyczna niczym specjalnym się nie wyróżnia — gołosłowne twierdzenia i slogany występują masowo. „Nasze najnowsze techniki gwarantują 100% wzrostu produktywności", „użycie narzędzi pozwoli skrócić testowanie o ⅔" — przykłady można mnożyć. Czego nam brakuje, by móc przeciwstawić wiedzę próbom manipulacji? Systematycznego stosowania pomiarów i dostępu do ich wyników.

Planowanie projektów informatycznych okazuje się w praktyce niejednokrotnie zwykłym wróżeniem z fusów. Jak oszacować pracochłonność projektu produkcji oprogramowania, którego wymagania są opisem słowno-muzycznym? Niełatwo na takiej podstawie oszacować wielkość produktu, na przykład w formie liczby wierszy kodu, a nawet mając do dyspozycji takie oszacowanie, nie ma prostej i godnej zaufania metody, aby przełożyć je na liczbę koniecznych osobodni.

Brak umiejętności mierzenia powoduje, że jakiekolwiek dyskusje porównujące zalety i wady różnych metod, technik, języków programowania i modelowania cierpią na chroniczną dowolność, na odwoływanie się do anegdotycznych, statystycznie nieistotnych przykładów. Nie umiejąc mierzyć przebiegu projektów, nie potrafimy na dobrą sprawę nimi zarządzać. Intuicja, objawienia, i-cing — wszystko to bardzo ciekawe metody pod warunkiem, że *uzupełniają* proces decyzyjny i przewidywania oparte na pomiarach, a nie usiłują je *zastępować*. Zarządzanie ryzykiem staje się fikcją, jeśli ani nie potrafimy zagrożeń zidentyfikować, ani oszacować kosztów zapobiegania, ani ocenić ich prawdopodobieństwa. W opublikowanym niedawno artykule napisałem, że „w przemyśle informatycznym chętnie udajemy kierowców Formuły 1 oraz dzielnych przewodników, nie mając niezbędnych ku temu umiejętności mierzenia".

Jak nauczyć się trudnej sztuki wykonywania pomiarów i analizy ich wyników? Jak nie dać się w przyszłości zagadać elokwentnym zwolennikom Jedynie Słusznej Drogi, czy będzie nią czysty Brak Procedur, czy ISO 9000, czy CMM, czy cokolwiek innego, tylko dojść samodzielnie do własnych wniosków, wybierać to, co rzeczywiście najlepsze dla nas i naszych firm?

Doskonałym, praktycznym przewodnikiem jest wydana pod koniec ubiegłego roku książka[3] dr. Andrzeja Kobylińskiego z SGH pod niezbyt chwytliwym marketingowo tytułem *Modele jakości produktów i procesów programowych*. Nazwałbym ją chętniej *Informatyk mierzy*.

Książka

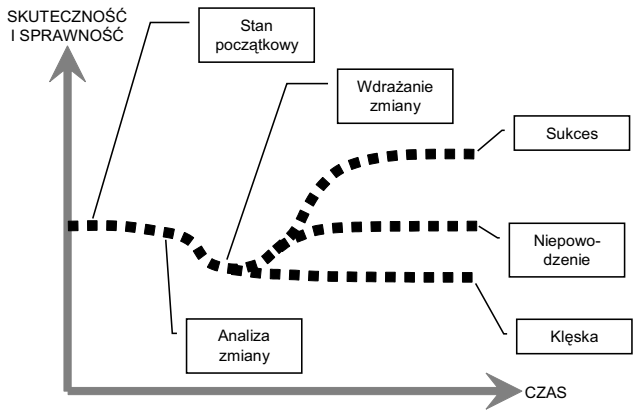
Pierwsza, podstawowa zaleta omawianej książki dla praktyków informatyki to jej przystępność. Choć jest to pozycja naukowa, akademicka, jednak zarówno jej język, jak i zorientowany na praktyczne potrzeby tryb wykładu umożliwiają skuteczną lekturę także osobom mającym głównie praktyczne doświadczenia informatyczne.

Jakość to pojęcie kluczowe dla praktyki informatycznej, a mimo to niebezpiecznie niejednoznaczne. Choć definicji jakości jest wiele, a każda zasługuje na osobną monografię, istnieje przecież duża, niezaspokojona potrzeba jednoznacznego określenia jakości tak, by móc ją mierzyć, oceniać, porównywać, zapisywać w kontraktach i wymaganiach. W jaki sposób jakość wpisuje się w praktykę projektu informatycznego — to tematyka pierwszej części książki.

Część druga dotyczy **jakości produktu**. Omawia atrybuty jakości produktów, zależności między nimi oraz sposoby ich pomiarów. To tematyka o dużym znaczeniu dla wszystkich udziałowców projektu informatycznego. Niedostatki wiedzy na ten temat powodują, że klienci niejasno i niepoprawnie określają swoje wymagania, analitycy systemowi sporządzają niepełne modele, inżynierowie jakości mają problemy z jednoznaczną oceną jakości gotowego lub budowanego właśnie produktu.

Jeśli w projektach zdarzają się kłopoty, próbuje się je rozwiązywać, usprawniając **procesy i procedury**. Jest wiele różnych modeli, jak postępować, aby określić i wdrożyć usprawnienia, a każdy z nich ma swoich zwolenników i przeciwników. Czym się od siebie różnią, który najlepiej pasuje do naszych potrzeb? Nie jest to pytanie teoretyczne. Zmienia się środowisko, w którym firmom przychodzi działać i konkurować, więc firmy muszą także się zmieniać, by sprostać nowym wyzwaniom. Różne są przyczyny i cele zmian, a udoskonalenie sposobu pracy jest jednym z ważniejszych. Próba udoskonalenia może zakończyć się trojako:

[3] Andrzej Kobyliński *Modele jakości produktów i procesów programowych*, Szkoła Główna Handlowa w Warszawie, Warszawa 2005, ISSN 0867-7727. Można ją kupić w Oficynie Wydawniczej SGH, *www.wydawnictwo.waw.pl*.



Rysunek 4.3.1.
Koszty zmiany w czasie

Aby zmniejszyć ryzyko niepowodzenia, trzeba sprawnie przeprowadzić analizę możliwej zmiany oraz skutecznie zrealizować jej wdrożenie. Jak posiąść niezbędną do tego celu wiedzę? Czytając część trzecią książki, gdzie znajdziemy zarówno przegląd istniejących, znanych metod oceny i doskonalenia procesów, jak i porównanie wynikających z nich korzyści oraz zagrożeń.

Tematyka budząca spore emocje, a przy tym znacząca dla powodzenia projektów oraz firm, to kwestia relacji między udoskonalaniem procesów a jakością produktów. Każdy pewnie słyszał sarkastyczne historyjki o wdrożeniach certyfikatów ISO 9000, wymagających jakoby jedynie naklejenia karteczek z nazwą na wszystkie znajdujące się w firmie przedmioty, mnożących zbędną papierologię i nieprzekładających się w żaden sposób na jakość produktów. Na przykład słynny Tom DeMarco jest sceptycznie usposobiony do korzyści płynących z osiągania wyższych poziomów CMM, twierdząc, że prowadzi to wyłącznie do polepszenia chwilowej sprawności kosztem zmniejszenia elastyczności i utraty strategicznej efektywności.

Część czwarta książki poświęcona jest temu właśnie zagadnieniu ujętemu w unikalny, własny sposób autora.

Podsumowując — książka Andrzeja Kobylińskiego to pozycja, którą każdy menedżer firmy czy kierownik projektu informatycznego będzie czytać z dużym zainteresowaniem, a należy ją także polecić inżynierom jakości oraz inżynierom testów. Bo przecież, żeby mówić o jakości, trzeba ją umieć mierzyć, test jest zaś podstawowym sposobem pomiaru!

4.4. Między biurokracją a chaosem: ADP

Kłopot

Kiedy byłem dzieckiem, przeżyłem okres fascynacji robieniem na drutach. Opanowawszy sztukę zapętlania kilku rodzajów oczek, stworzyłem coś na kształt długiego na metr, nierównego szalika składającego się z bezładnej mieszanki wzorów i oczek. Nie dało się tego do niczego używać i zaraz potem porzuciłem krótkotrwałe zamiłowanie.

Podobnie, choć nie aż tak źle, wyglądało tworzenie oprogramowania w pierwszych dwóch dekadach istnienia komputerów. Przedsięwzięcia informatyczne (ang. *projects*) były chaotyczne i nieplanowe, nie stosowano systematycznej inżynierii wymagań (ang. *requirements engineering*), kod tworzono bez uprzedniego projektowania (ang. *design*), więc produkt końcowy zwykle nie miał trafnej funkcjonalności, nie był wygodny w użytkowaniu ani łatwy w utrzymaniu.

W roku 1968 podczas konferencji inżynierii oprogramowania NATO (*NATO Software Engineering Conference*) termin inżynieria oprogramowania po raz pierwszy pojawił się oficjalnie [1]. Doczekało się uznania twierdzenie, że tworzenie oprogramowania to coś więcej niż zgodne z regułami języka programowania stawianie szeregu instrukcji; że istnieją systematyczne, zdyscyplinowane i mierzalne metody wykonywania tego procesu.

Wkrótce — 7 października 2008 r. [2] — przyjdzie nam zatem obchodzić 40. rocznicę tego wydarzenia[4]. Gdzie inżynieria oprogramowania znajduje się dzisiaj?

Akcja i kontrakcja

Na poziomie kodowania (implementacji) i częściowo także projektowania zaszły ogromne zmiany. Od paradygmatu GOTO, przez metody strukturalne, przez nieudane próby języków 4. generacji, przemysł informatyczny wszedł w erę metod i języków obiektowych.

Od tworzenia każdej aplikacji niemal od zera, przez biblioteki funkcji, dotarliśmy do hierarchii klas, bibliotek łączonych dynamicznie [3] i wielojęzykowej platformy CORBA [4]. Od niewydarzonego, topornego interfejsu wierszowego [5] przeszliśmy do znacznie wygodniejszych interfejsów graficznych [6]; zaczęto też coraz systematyczniej uprawiać inżynierię interakcji [7].

Mniej radykalne przemiany zachodziły na wyższych poziomach procesu: projektowania architektury, inżynierii wymagań oraz testowania, ale i tutaj można bez wątpienia wskazać wiele nowych metod, modeli oraz narzędzi.

Dziś — w porównaniu z sytuacją sprzed 40 lat — mamy więc do dyspozycji o wiele więcej znacznie potężniejszych sposobów pracy, dzięki którym daje się tworzyć produkty coraz bardziej złożone coraz szybciej.

W tyle za rosnącą skutecznością i sprawnością metod technicznych pozostawała i pozostaje nauka o zarządzaniu przedsięwzięciami. Nasza wiedza o tym, jak optymalnie organizować przedsięwzięcia informatyczne, jak dobierać oraz wiązać ze sobą metody i narzędzia, uwzględniając rodzaj produktu, wymagania niezawodności oraz szereg czynników ludzkich, wciąż pozostaje w powijakach. Pojawiło się i zyskało krótkotrwałą sławę wiele nowości: głośno było raz o TQM, kiedy indziej o *clean-room software engineering*, o technikach przyrostowych, o *daily build*, o modelowaniu obiektowym, o RUP — nazwy można mnożyć. Narastająca złożoność i ociężałość modeli procesów wytwarzania oprogramowania spowodowała kontrakcję: od około 15 lat coraz większą popularnością cieszą się metodyki lekkie i zwinne (ang. *agile* [8]).

[4] Esej napisany we wrześniu 2007 roku.

Procesy — zarówno ciężkie, jak i zwinne — nie są jednak dane raz na zawsze, powinny się zmieniać. Jak mierzyć i oceniać, a w miarę potrzeby zmieniać, udoskonalać, usprawniać nasze procesy? Powstał szereg metametodyk (metodyk usprawniania metodyk), które można podzielić na dwie wielkie stronnictwa: metametody ciężkie i metametody lekkie (terminologia własna autora artykułu).

Metametody ciężkie: rezerwat leśnych dziadków

Ciężkich, rozbudowanych metod mierzenia i udoskonalania procesów jest wiele; wymieniamy je poniżej. Pozwalam sobie w odniesieniu do nich używać złośliwego określenia *rezerwat leśnych dziadków* ze względu na ich skłonność do odrywania nadbudowy od bazy (samemu będąc niewątpliwym leśnym dziadkiem, mam jak widać skłonność do używania terminologii z minionych epok, nie tylko w informatyce). Ciężkie metody postulują budowę i utrzymywanie ogromnego aparatu nadzorującego, przez co wymagają znacznych zasobów i nakładów, a ich stosowanie w praktyce przedsięwzięć informatycznych powoduje duże spowolnienie procesu i niechęć jego uczestników. Stąd syndrom rezerwatu: ambitne, rozbudowane metodyki żyją życiem niezależnym od realiów projektów.

Przykłady takich metod/modeli to rodziny ISO 9000 – ISO 9001, Six Sigma, CMMI, COBIT, ITIL, TickIT, ISO/IEC 12207, Bootstrap, SPICE, ISO/IEC 15504 oraz metodyki udoskonalania procesu testowego TMM, MMAST, TAP, TCMM, TIM, TOM, TSM, TPI.

To imponująca i groźna lista skrótowców — ze szczegółami oraz praktyką można się zapoznać między innymi na spotkaniach sieci SPIN (*Software Process Improvement Network*, *www.spin-pl.org*) w Gdańsku, Krakowie, Szczecinie, Warszawie i we Wrocławiu[5].

Metametody lekkie: rezerwat młodych wilków

Metody lekkie można podsumować — narażając się na zarzut niejakiej przesady — jako minimalizowanie wszelkich działań oprócz czysto konstrukcyjnych. Czyli w pewnym stopniu następuje odwrócenie sytuacji: zamiast rezerwatu leśnych dziadków mamy rezerwat młodych wilków, które — z błogosławieństwem swoich proroków — biurokracji metod ciężkich przeciwstawiają zorientowane na skuteczność konkretnego projektu podejście na skróty. Przykłady takich metod to XP (*eXtreme Programming*), metodyki *Agile* (np. *Agile SCRUM*), metody ewolucyjne (iteracyjne), prototypowanie, *daily build*, RAD (ang. *Rapid Application Development*). W dziedzinie testowania specyficzną formą lekkiej metodyki są testy eksploracyjne (ang. *exploratory testing*).

[5] Od tego czasu działalność SPIN-ów zaktywizowała się w Gdańsku i w Krakowie, a osłabła w innych miastach (stan z sierpnia 2008 roku).

Niedostatki rezerwatów

Słabości metod i modeli ciężkich są oczywiste: w wielu przypadkach ich złożoność i koszty postępowania według ich wskazań po prostu przewyższają korzyści, tak jak koszt instalacji linii robotów przemysłowych w małomiasteczkowym warsztacie samochodowym.

Nawet jednak, jeśli potencjalnie inwestycja w ciężką metodykę ma szanse się zwrócić, to jej nieelastyczność, ignorowanie mechanizmów psychologicznych i społecznych, zawiłość i oddalenie od konkretów, stanowią poważną przeszkodę w ich zastosowaniu.

Z drugiej strony, metody lekkie są na dobrą sprawę rezygnacją walkowerem z korzyści gromadzenia i stosowania dobrych praktyk, z międzyprojektowego transferu wiedzy i umiejętności, z systematyzacji i dyscypliny, które nie zawsze są tylko obciążeniem.

Wyraźnie potrzebna jest trzecia droga — droga między biurokracją a chaosem.

ADP — nareszcie!

Zapoznając się z metodyką, opisaną przez Adama Kolawę i Dorotę Huizinga w ich mającej się ukazać we wrześniu książce *Automated Defect Prevention: Best Practices in Software Management* [9], co chwila miałem chęć zawołać „nareszcie!". Nareszcie pewne oczywiste prawdy, które aż dziw, że nie zostały wyartykułowane wcześniej, doczekały się opisania! Nareszcie mamy model procesu informatycznego, opisujący projektową rzeczywistość w miejsce hermetycznego, pełnego pobożnych, acz nierealnych życzeń świata modeli ciężkich. Nareszcie otrzymujemy metodykę, która w spójną jedność łączy trzy zwykle obce sobie dotąd światy:

- skuteczne projektowanie oraz implementację kodu;
- pomiary, ocenę oraz udoskonalanie procesów i procedur;
- psychologię uczestników przedsięwzięcia informatycznego.

Aż się prosi, aby metodzie ADP nadać jakąś bardziej porywającą nazwę, skoro przychodzi jej konkurować ze swoistym fundamentalizmem tradycyjnych — zarówno lekkich, jak i ciężkich — metod. Czy istnieje korelacja między jakością projektu a jakością produktu oraz czy jest to zależność przyczynowa? Wydaje się, że tej kluczowej kwestii powinna być poświęcona większość prac dotyczących procesów oraz ich udoskonalania, ale tak nie jest. Dominuje — zaiste przypominające niekiedy religijny fanatyzm — cyzelowanie metod oraz szczegółowe opisywanie przypisywanych im cudów, bez prób choćby odpowiedzi na pytanie, na ile rzetelnie owe cuda zostały zarejestrowane i na ile metodzie właśnie można przypisać ich zaistnienie. Oto groźny cytat: *Znakomite wyniki oraz sama liczba znanych przedsiębiorstw, które stosują SixSigma, są dowodem na to, że SixSigma nie jest przemijającą modą! Od telekomunikacji, poprzez sektor finansowy, aż po przedsiębiorstwa zajmujące się technologiami informatycznymi, wiele renomowanych firm zademonstrowało moc SxSigma w swoich organizacjach* [13].

Trudno o bardziej rażący przykład myślenia fundamentalistycznego, mylącego rzeczowe argumenty z pozorną siłą okrzyków pasujących raczej do manifestacji niż do naukowej czy inżynierskiej debaty.

ADP od środka

Patrząc na spis zagadnień poruszanych przez ADP, nie widzi się na pierwszy rzut oka jej rewolucyjnego charakteru. Główne rozdziały książki to „Planowanie oraz infrastruktura", „Specyfikacja i zarządzanie wymaganiami", „Projektowanie architektury oraz projektowanie szczegółowe" i tak dalej aż do „Testowania" i „Wdrożenia". Pozornie *nihil novi sub sole*.

Siła ADP tkwi jednak nie w efekciarskim stawianiu na głowie faktu, że informatyczne przedsięwzięcia składają się w rzeczywistości z takich a nie innych faz czy etapów, lecz na sposobie ich modelowania, pomiaru oraz realizacji procesów udoskonalania radykalnie realistycznym i zorientowanym przede wszystkim na praktyczną skuteczność.

Spójrzmy, jak ADP określa swoje cele.

Zadowoleni ludzie

Dotąd sprawy psychologii, ludzkiego zadowolenia, motywacji i kreatywności musiały wdzierać się do inżynierii oprogramowania albo chyłkiem, w aurze lekkiego skandalu, albo traktowane były instrumentalnie jako dowody na wyższość metod lekkich: *XP jest odświeżającym, nowym podejściem. XP jest skuteczny, ponieważ kładzie nacisk na zaangażowanie klienta i promuje pracę zespołową. A zatem jak by to miało działać? Najbardziej zadziwiającym aspektem XP jest prostota reguł i procedur. Na początku wydają się niezręczne i naiwne, ale wkrótce stają się mile widzianą odmianą. Klienci są zadowoleni z bycia partnerami w procesie programowania, a programiści aktywnie uczestniczą w tym procesie bez względu na doświadczenie* [14].

Kto ośmieliłby się postawić pytanie, czy wdrożenie na przykład ITIL zwiększa radość życia zatrudnionych? Kto, z drugiej strony, sprzeciwiłby się gołosłownemu — co nie oznacza, że koniecznie nieprawdziwemu! — twierdzeniu, że OOP jest w jakiś mistyczny sposób lepsze i bardziej odpowiada ludzkim potrzebom niż programowanie strukturalne? — tylko nieliczni [15].

ADP zdołało wpisać czynnik ludzki w samą metodę, nie tylko jako ozdobnik. Nie mamy tu miejsca, aby opisać wiele, posłużę się więc dwoma znamiennymi cytatami: *osiągnięcie równowagi pomiędzy dyscypliną i kreatywnością jest trudne* oraz *zarówno nadmierna nuda, jak i nadmierny niepokój czynią ludzi mniej efektywnymi i bardziej skłonnymi do błędu.*

Tym niekorzystnym tendencjom ma zapobiegać właściwie, elastycznie stosowane ADP.

Wysoka jakość produktu

Ten podstawowy cel metodyk ginie często w nawale szczegółów samej metody. ADP stawia go na drugim miejscu.

Organizacja: wyższa produktywność i sprawność w działaniu

Nareszcie nie chodzi o to, byśmy się stali dojrzalsi (choćby kosztem skuteczności i sprawności), ani o to, byśmy potrafili umieścić etykietki na każdej procedurze i każdym artefakcie naszej organizacji, tylko o produktywność i sprawność. Jak je osiągnąć, zależy od produktu, charakterystyki projektu, ludzkich kwalifikacji i preferencji, a ADP ułatwia ich określenie oraz pomiar stopnia realizacji. W tym sensie ADP jest nie tyle metodyką, co metametodyką, nie receptą, lecz „receptą jak-wybrać-właściwą-receptę", od lat bezskutecznie poszukiwaną przez praktyków.

Proces nadzorowany, udoskonalany i dający się utrzymać

CMMI poziom 5 czy lepiej poziom 2? Formalne modelowanie wymagań czy nieprecyzyjny opis słowny? Nie ma na te pytania jedynej słusznej odpowiedzi, natomiast niezależnie od tego, jak brzmi w konkretnym przypadku, opłaca się wiedzieć, co robimy (*proces nadzorowany*), umieć, jeśli trzeba, poprawić procedury (*proces udoskonalany*) oraz zachować zdolność do przestrzegania w rzeczywistości tych praktyk, które uznajemy za dla nas najlepsze (*proces dający się utrzymać*) — oto sens ADP.

Przedsięwzięcie zarządzane poprzez podejmowanie decyzji

Podejmowanie decyzji, zarządzanie zmianami i ryzykiem — to kolejny klucz do powodzenia przedsięwzięć informatycznych. Aby móc jednak podejmować dobre decyzje, niezbędna jest informacja, niezależnie od tego, czy proces decyzyjny jest sformalizowany i racjonalny, czy intuicyjny i podświadomy [16]. ADP kładzie nacisk na gromadzenie i analizę danych projektowych oraz na automatyzację tego procesu.

Zapobieganie pomyłkom i błędom

Oprogramowanie jest substancją pozwalającą — inaczej niż kamień, stal czy drewno — względnie łatwo usuwać defekty w zbudowanych z niej produktach nawet wtedy, gdy produkt jest w pełni ukończony. Ta cenna właściwość ma jednak ogromnie niekorzystne skutki uboczne, promuje bowiem myślenie krótkowzroczne, usuwanie defektów zamiast ich przyczyn, co powoduje chroniczną zawodność produktów

informatycznych. ADP zaleca podejście przeciwne: analizę przyczyn pomyłek i błędów (ang. *root cause analysis*) oraz ich systematyczne usuwanie wsparte automatycznymi narzędziami.

Zasady ADP

Znajomość podstawowych zasad ADP pozwala zrozumieć skuteczność i prostotę tej metodyki.

- **Stworzenie infrastruktury jednoczącej ludzi i technologię.** Metody takie jak na przykład CMMI mówią wiele o tym, *co* należy zrobić, ale niewiele o tym, *jak*: nie podają szczegółowych wskazówek wspomagających przełożenie zacnych założeń w konkretną praktykę i projektowe korzyści. ADP przeciwnie — określa, jak za pomocą narzędzi ułatwić i usprawnić rzeczywiste wdrożenie zaleceń.
- **Określenie i zastosowanie dobrych praktyk —** to normatywna część ADP, a więc podobna nieco do metodyk i modeli tradycyjnych. To, czym ADP korzystnie się wyróżnia, to uznanie istnienia dobrych praktyk na wielu różnych poziomach, od ogólnych zasad SQA po szczegółowe wskazówki dotyczące implementacji kodu.
- **Dostosowanie dobrych praktyk** — nie istnieje uniwersalny zbiór dobrych praktyk; każda firma, dziedzina i projekt powinny ogólne dobre praktyki modyfikować i uzupełniać, gromadząc własne, szczególne doświadczenia. A więc model czy metodę nie tyle się wdraża według z góry założonego skryptu, co projektuje, buduje i wdraża jednocześnie.
- **Pomiary i monitorowanie statusu projektu** — automatyczny system gromadzenia i analizy stanu projektu pozwala zarówno na sprawne określenie jego statusu, jak i na identyfikację jego słabych stron i możliwych usprawnień. Stosowanie tej zasady pozwala — posługując się terminologią CMMI — wdrożyć elementy poziomu piątego CMMI nawet w organizacjach niespełniających wszystkich formalnych jego wymogów!
- **Automatyzacja** — weźmy do ręki strukturę dowolnego dużego modelu, na przykład COBIT; przecież niemożliwe, narażone na liczne pomyłki i porażająco — nie bójmy się tego słowa! — nudne byłoby jego wdrożenie bez zakrojonej na szeroką skalę automatyzacji przestrzegania jego wytycznych! ADP kładzie ogromny nacisk na automatyzację procedur, przez co ich przestrzeganie staje się możliwe bez narażania ludzi na konieczność uciążliwego, ręcznego wprowadzania danych wymaganych przez procedury, odciąża ich od frustrujących, rutynowych zadań, ogranicza zagrożenie pomyłkami.
- **Przyrostowe wdrożenie praktyk ADP**. To zupełna nowość — dotąd nawet metodyki zalecające przyrostowe tworzenie oprogramowania nie przewidywały przyrostowego wdrażania samych siebie! Istnieje szereg czynników psychologicznych powodujących opór przed zmianami. Liczne kursy „miękkie" usiłują nas nauczyć, jak z tym oporem sobie radzić rozmaitymi psycho- i socjotechnikami, ale nikt jakoś dotąd nie zaproponował oczywistego rozwiązania (dobrze skądinąd znanego politykom podwyższającym podatki) — stopniowo i krok po kroku!

Who is who

Adam Kolawa [10], założyciel w 1987 roku amerykańskiej firmy Parasoft [11], której jest prezesem, oraz Dorota Huizinga [12], profesor informatyki na Uniwersytecie Fullertona w Kalifornii, to osoby mające szczególne kompetencje do stworzenia nowego paradygmatu — ADP.

ADP nie poleca żadnych określonych narzędzi. Książka zawiera obszerną listę narzędzi, zarówno komercyjnych, ogólnodostępnych i open-source, mogących znaleźć zastosowanie we wdrażaniu metodyki.

Więcej danych na temat wydarzeń, usług i szkoleń na temat ADP można znaleźć w Internecie[6].

Referencje

[1] *http://en.wikipedia.org/wiki/Software_engineering*

[2] *http://en.wikipedia.org/wiki/List_of_publications_in_computer_science#Software_engineering:_Report_of_a_conference_sponsored_by_the_NATO_Science_Committee*

[3] *http://en.wikipedia.org/wiki/Dynamic_linking#Dynamic_linking*

[4] *http://en.wikipedia.org/wiki/Corba*

[5] *http://en.wikipedia.org/wiki/Command_line_interface*

[6] *http://en.wikipedia.org/wiki/History_of_the_graphical_user_interface*

[7] *http://en.wikipedia.org/wiki/Human%E2%80%93computer_interaction*

[8] *http://en.wikipedia.org/wiki/Agile_software_development*

[9] *http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html*

[10] *http://www.socaltech.com/fullstory/0000180.html*

[11] *http://en.wikipedia.org/wiki/Parasoft*

[12] *http://dorota.ecs.fullerton.edu/*

[13] *http://www.motorola.com/content.jsp?globalObjectId=3081*

[6] Rok po napisaniu tego artykułu powstała ADPQB — ADP Qualifications Board. Adres: *www.adpqb.org*.

[14] *http://www.ccsr.cse.dmu.ac.uk/conferences/ccsrconf/ethicomp2001/abstracts/bogdan.html*

[15] *http://www.devx.com/opinion/Article/26776/0*

[16] *http://www.bettersoftware.eu/white_papers/dobre_decyzje_0.4.pdf*

[17] *http://www.bettersoftware.eu/adp.html*