Akademia Ekonomiczna w Poznaniu

WYDZIAŁ ZARZĄDZANIA

Studium podyplomowe Zarządzania Zasobami Ludzkimi

Michał Owsiak

Redukcja wpływu negatywnych czynników w projekcie informatycznym

Praca dyplomowa

Promotor:
dr Tomasz Kopczyński

Poznań 2004

Spis treści

Wstęp **3**

1 Czynniki wpływające na porażkę w projekcie informatycznym **5**

1.1 Ratowanie terminu 5

1.2 Nadgodziny 7

1.3 Redukcja harmonogramu 8

1.4 Zaniechanie projektowania 9

1.5 Zaniechanie testowania 11

1.6 Obniżanie jakości 12

2 Przeciwdziałanie błędnym decyzjom w projekcie informatycznym **14**

2.1 Tworzenie realnego harmonogramu 15

2.2 Wspólny język 17

2.3 Pragmatyzm i jakość 20

2.4 Korzystanie z wiedzy innych 22

3 Proste błędy oraz ich skutki w przedsięwzięciach informatycznych **25**

3.1 Niezrozumienie użytkownika 25

3.2 Złudne prototypy 26

3.3 Pomijanie projektowania 27

3.4 Ratowanie terminu 28

Zakończenie **31**

Literatura **32**

Wstęp

Rewolucja informatyczna wymusiła wykształcenie nowej dziedziny nauki pozwalającej na skodyfikowanie i unormowanie procesu wytwarzania oprogramowania - *inżynierii oprogramowania*. Celem wspomnianej gałęzi informatyki jest opracowanie jak najlepszych sposobów opisu procesów towarzyszących wytwarzaniu oprogramowania.

Mimo tego, że istnieje szereg metod wspomagających proces tworzenia aplikacji, które to metody powinny wpływać na lepsze zarządzanie projektami informatycznymi, statystyki pozostają bezlitosne. Spośród wszystkich prowadzonych projektów informatycznych aż 50%, przekracza termin lub budżet przeznaczony na ich realizację. Wśród tej połowy niedoszacowanych projektów, aż 30% kończy się całkowitą rezygnacją lub porzuceniem pierwotnych założeń[1]. Naturalnym więc wydaje się zadanie pytania, gdzie leżą przyczyny tych wszystkich niepowodzeń?

W tej pracy nie zostanie udzielona jednoznaczna, pełna i satysfakcjonująca odpowiedź na powyższe pytanie. Wynika to z tego, że przyczyny upadków projektów oraz ich geneza są niezwykle różnorodne i złożone. Problemy, jakie pojawiają się w trakcie realizacji projektów *high-tech* to między innymi: nie przywiązywanie wagi do relacji międzyludzkich, zbytnie zaufanie do technologii, nie uwzględnianie rzeczywistych potrzeb użytkowników oraz błędne decyzje kierownictwa związane z ignorowaniem podstawowych zasad projektowania oprogramowania.

W niniejszej pracy przedstawione zostały te błędy, które mimo tego, że są łatwe do identyfikacji, są nadal popełniane w wielu projektach. Błędy te występują w większości przedsięwzięć informatycznych - poczynając od prostych aplikacji pisanych przez komputerowych zapaleńców, a kończąc na złożonych systemach wykorzystywanych do obsługi sektora bankowego. Zaproponowane zostały takie działania, które mogą minimalizować szkody wyrządzone przez błędne działania oraz ryzyko z nimi związane.

Przyjęte w pracy założenia uwzględniają takie elementy jak *zdrowy* rynek pracy (celem pracy nie jest wykazanie, że przy niezdrowym traktowaniu pracowników ciężko jest zakończyć sukcesem, w dłuższym czasie, jakiekolwiek przedsięwzięcie informatyczne) oraz dysponowanie zespołem kompetentnych specjalistów. W tej pracy wykazano, że przy takich warunkach można na tyle błędnie pokierować zespołem, że mimo jego kompetencji, nie będzie on w stanie naprawić błędów popełnionych przez

[1] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami*, WNT, Warszawa 2003, s. 7

menedżerów. Brak kompetencji będzie wskazany jako jeden z czynników wpływających na porażkę.

W kolejnych rozdziałach przedstawione zostały podstawowe problemy występujące w trakcie projektowania aplikacji, czynniki na które należy zwrócić uwagę oraz przykłady zaczerpnięte z toczących się, bądź zakończonych już, projektów.

W rozdziale pierwszym opisane zostały podstawowe czynniki porażki występujące w trakcie wytwarzania oprogramowania. Zostały one podzielone na czynniki społeczne (związane z oddziaływaniem osób rozwiązujących dany problem), techniczne (związane z wyborem, poszukiwaniem oraz eksploatacją danej technologii) oraz menedżerskie (związane z podejmowaniem decyzji dotyczących kierowania zespołami oraz projektem).

Rozdział drugi traktuje o sposobach przeciwdziałania oczywistym błędom w procesie wytwarzania oprogramowania. Prezentuje podstawowe sposoby hamowania zapędów menedżerów oraz temperowania technicznego zapału inżynierów przy jednoczesnej maksymalizacji możliwości każdej z grup. Wskazuje również na wagę procesu projektowania oraz planowania. Na konieczność podjęcia dialogu z użytkownikiem. Na takie traktowanie przyszłych odbiorców, by przy jednoczesnym uznaniu ich rzeczywistych potrzeb umiejętnie odrzucać te żądania, które mogą stanowić zagrożenie dla całego przedsięwzięcia mimo tego, że dotykają tylko niewielkiej części problemu.

Rozdział trzeci to, poparte dotychczasowym doświadczeniem autora, rozważania dotyczące projektów informatycznych. Szczególnie zwrócono uwagę na te aspekty, które mogły zostać przeprowadzone lepiej, dokładniej, a przede wszystkim zgodnie ze sztuką projektowania, wytwarzania i wdrażania aplikacji. Doświadczenia te w wielu przypadkach pokrywają się z wymienianymi w literaturze błędami i potknięciami spotykanymi w dużych przedsięwzięciach informatycznych.

1 Czynniki wpływające na porażkę w projekcie informatycznym

Projektowanie, programowanie i wdrażanie aplikacji komputerowych jest w znacznym stopniu narażone na problemy pojawiające się w trakcie realizacji. Wynika to z faktu, że stosunkowo nowe technologie, określane mianem *high-tech*, wkraczają w obszary, w których nie były do tej pory stosowane. Wywołuje to napięcia między grupami biorącymi udział w procesie wytwarzania tych aplikacji. Relacje międzyludzkie[2] mają więc ogromne znaczenie w całym okresie trwania przedsięwzięcia. W głównej mierze jest to związane z tym, że do końca nie ma pewności do czego tak naprawdę się dąży, realizując dane przedsięwzięcie.

Kolejnym problemem jest zbyt duże zaufanie do technologii, nie uwzględniające rzeczywistych potrzeb użytkowników. Działanie takie prowadzi do tworzenia produktów zaawansowanych technologicznie, ale w istocie rzeczy nikomu nie potrzebnych[3]. Do tego dochodzą jeszcze błędne decyzje kierownictwa związane z ignorowaniem podstawowych zasad projektowania oprogramowania[4] (czy to z braku kompetencji czy też zbytniej wiary w możliwości swoje i zespołu).

W toku pracy zostały wymienione te czynniki, które są bardzo łatwe do identyfikacji, a mimo to nadal sprawiają bardzo dużo kłopotów menedżerom oraz projektantom.

1.1 Ratowanie terminu

W przedsięwzięciach informatycznych bardzo często wykorzystywane jest pojęcie kamienia milowego (*mile-stone*). Jakkolwiek użyteczne, bywa błędnie rozumiane, przez co potrafi stanowić istotny czynnik ryzyka.

Najprościej mówiąc, idea kamieni milowych to podzielenie prac nad oprogramowaniem na etapy. Podział następuje w taki sposób, że w każdym momencie można jednoznacznie określić, czy dany kamień milowy został osiągnięty, czy nie. Jeżeli kamień nie został osiągnięty, to sposób podziału powinien pozwolić na stwierdzenie, ile jeszcze należy wykonać pracy, aby dany etap został zakończony. D. Hamlet wspomina o potrzebie istnienia kamieni milowych jako podstawowego źródła informacji,

[2]T.DeMarco, T.Lister, *Czynnik ludzki*, WNT, Warszawa 2002, s. 20
[3]A.Cooper, *Wariaci rządzą domem wariatów,* WNT, Warszawa 2001, s. 111
[4]F.P.Brooks,Jr., *Mityczny osobomiesiąc*, WNT, Warszawa 2000, s. 45

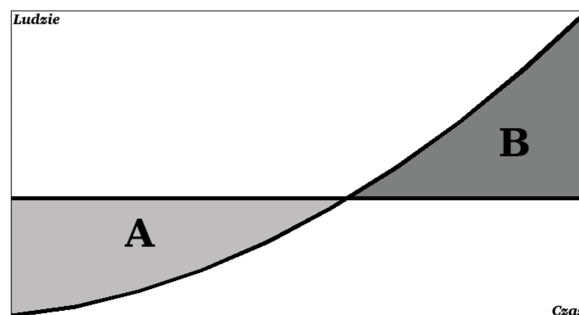
proponując jednocześnie zestaw cech opisujących dobrze zdefiniowany kamień milowy[5].

Ustalanie kamieni milowych jest mylnie rozumiane przez niektórych menedżerów jako określanie terminów przyjmowania kolejnych etapów, ale bez uwzględnienia kryteriów oceny. Prowadzi to do takiej oto sytuacji, że możemy jednoznacznie stwierdzić ile jeszcze czasu pozostało do zakończenia danego etapu, ale nie możemy powiedzieć ile jeszcze pracy musimy wykonać, aby etap ten zakończyć powodzeniem. Takie podejście sprawia, że daje o sobie znać zasada *80/20 (zasada Pareto):*

Pierwsze 80% wartości czynności osiąga się przez 20% wysiłków,
a końcowe 20% wartości wymaga włożenia 80% wysiłków[6]*.*

Podzielenie czasu przeznaczonego na realizację zadania nie uwzględniając tej zasady, to w miarę zbliżania się do ustalonego terminu ilość pracy koniecznej do wykonania będzie stopniowo rosła (zgodnie z zasadą 80/20), przekraczając założony (stały) rozkład pracy. Sytuacja ta przedstawiona została na Rysunku 1.

Rysunek 1: Ilustracja zasady 80/20



Źródło: T. DeMarco, T. Lister, *Czynnik ludzki*

Menedżerowie, kreśląc wymaganą ilość pracy jako stałą, prowadzą do niedoszacowania realnych potrzeb zarówno na początku jak i pod koniec zadania. Obszar **A**, to obszar niedoszacowania wymaganych zasobów na początku przedsięwzięcia. Zasoby te zostaną bezpowrotnie stracone, gdyż w późniejszym czasie

[5]D.Hamlet, J.Maybee, *Podstawy techniczne inżynierii oprogramowania*, WNT, Warszawa 2003, s. 28
[6]M. Cantor, *Jak kierować zespołem programistów*, WNT, Warszawa 2004, s. 108

nie będzie można spożytkować straconej pracy. Praca ta nie była potrzebna na początku przedsięwzięcia. Obszar **B,** z kolei, to obszar w którym wymagania dotyczące ilości wykonywanej pracy są znacznie większe od tych, które zostały zaplanowane.

Menedżer znajdujący się w punkcie przecięcia funkcji, dochodzi do wniosku, że musi podjąć decyzje, które pozwolą na uratowanie harmonogramu. Najprostszym wyjściem wydaje się powiększenie zespołu na tyle, aby pokryć niedobór pracy. Jest to jeden z kardynalnych błędów menedżerów opierających się tylko i wyłącznie na harmonogramie, jako źródle informacji o postępach przedsięwzięcia. Działanie to, nie dość że nie przynosi pożądanych rozwiązań, prowadzi do dalszych opóźnień. Ten paradoks jest pochodną kilku czynników. Po pierwsze, osoby odpowiedzialne za zadanie muszą przeznaczyć pewną porcję czasu na wdrożenie nowych członków zespołu. Po drugie, zwiększa się ilość komunikacji w zespole. Po trzecie, do zespołu wprowadzani są zwykle słabsi pracownicy, którzy zaczynają systematycznie spowalniać prace zespołu[7]. Wspomniane zabiegi mają na celu uratowanie terminu za wszelką cenę. Ich wynikiem, natomiast, jest coraz większe obciążenie, już i tak znacznie zaangażowanych w pracę, osób.

1.2 Nadgodziny

Nadgodziny muszą pojawić się podczas źle oszacowanego harmonogramu, jeżeli menedżer chce za wszelką cenę dotrzymać terminu. Wynikiem takiego działania jest tylko i wyłącznie pogarszający się stan zespołu.

Przepracowani programiści nie są najlepszym sprzymierzeńcem kierownika. Programiści to osoby, które w większości przypadków wykorzystują do pracy umysł. Przemęczony, niewyspany programista nie jest w stanie sprostać wyzwaniom często przypominającym łamigłówki. M. Cantor nazywa wręcz nadgodziny marszem ku klęsce[8].

Nadgodziny, jeżeli już się pojawią, pozwalają na nadgonienie czasowe projektu w krótkim terminie. Na dłuższą metę powodują systematyczne przemęczenie pracowników, co może skutkować między innymi wprowadzaniem do kodu programu nowych błędów podczas usuwania poprzednich. W efekcie powoduje to znaczny spadek formy zespołu oraz jakości wytwarzanego oprogramowania. Dodatkowo zaczyna

[7]F.P. Brooks, Jr., *Mityczny osobomiesiąc* wyd. cyt., s. 31-34
[8]M. Cantor, J*ak kierować...* wyd. cyt., s. 122-123

dochodzić do spięć pomiędzy pracownikami, którzy są zmęczeni[9].

Spięcia pomiędzy członkami zespołu wynikają często z nierównomiernego obciążenia pracowników zadaniami. Praca w nadgodzinach wymaga dłuższego pozostawania w miejscu pracy, co nie zawsze jest możliwe w odniesieniu do wszystkich pracowników. Osoby posiadające rodziny posiadają zobowiązania inne niż tylko praca. Takie osoby są zwykle odciążane przez pozostałą część zespołu, aby mogły spędzać więcej czasu ze swoimi bliskimi. Prowadzi to do dwóch istotnych pól konfliktów. Po pierwsze, osoby pracujące w innym tempie zaczynają odstawać od reszty zespołu pod względem merytorycznym. Osoby pracujące w nadgodzinach wykonują więcej pracy, co za tym idzie wyprzedzają pozostałych. Po drugie, osoby pracujące w nadgodzinach zaczynają czuć się pokrzywdzone przez kierownictwo.

Widząc taką sytuację oraz coraz szybsze zbliżanie się do końcowego terminu oddania prac, kiepski menedżer podejmuje dodatkowe kroki mające na celu zmniejszenie ilości pracy - odrzucanie części elementów harmonogramu.

1.3 Redukcja harmonogramu

Odrzucanie harmonogramu ma najczęściej miejsce w przypadku błędnie określonej linii bazowej wymagań[10]. Linia bazowa określa jakie cechy powinny zostać zawarte w produkcie, aby ten spełniał oczekiwania użytkowników. Określenie listy cech na zbyt wygórowanym poziomie prowadzi zwykle do załamania harmonogramu. Zbyt wygórowane wymagania co do produktu, mają miejsce w przypadku przyjęcia zbyt optymistycznych założenia co do możliwości produkcyjnych zespołu.

Do określenia błędnej listy cech może również doprowadzić błędne rozumienie problemu *„zbierania wymagań”*[11]. Jeżeli wymagania będą rozumiane jako oderwany od rzeczywistości zestaw funkcjonalności definiowany przez grupę projektantów, to siłą rzeczy będzie on odbiegał od rzeczywistych oczekiwań użytkowników.

Innego rodzaju „pułapką czasową” jest zwlekanie z prezentacją produktu klientowi. Prezentacja produktu wywołuje u przyszłego użytkownika syndrom *„tak, ale”*[12]. Jest to dosyć frustrująca reakcja klienta na efekt pracy programistów. Jest ona związana

[9] T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 194-196
[10] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 221-223
[11] A.Hunt, D.Thomas, *Pragmatyczny programista*, WNT, Warszawa 2002, s. 230-23
[12] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 94

z tym, że zdefiniowanie, a raczej zapisanie wyobrażeń o przyszłym produkcie jest skomplikowanym procesem. Klient nie potrafi dokładnie opisać swoich oczekiwań. Co za tym idzie, jest zwykle rozczarowany pierwszą wersją produktu. Jeżeli syndrom ten wystąpi stosunkowo późno, może zabraknąć czasu na wykonanie niezbędnych poprawek w produkcie. Doprowadzi to załamania harmonogramu.

Wydaje się więc, że jednym z czynników sukcesu może być szybka prezentacja produktu oraz określenie prawdziwych wymagań na podstawie reakcji użytkowników. Powyższe stwierdzenie jest prawdziwe, ale tylko wtedy, gdy spełnione zostaną ostre kryteria prezentacji wyników pracy nad prototypem. Prezentacja prototypu musi być dokonywana przy pewnych założeniach. Kluczowym jest odrzucenie samego prototypu po jego akceptacji ze strony klienta[13]. Założenie to jest często łamane przez menedżerów widzących możliwość zaoszczędzenia czasu w przypadku pozytywnej reakcji klienta na prototyp. Natomiast proces ten powinien przebiegać według słów Mao Zedonga: „Burzyć, aby budować[14]".

Menedżer, widzący zadowolenie po stronie odbiorcy, zapomina o tym, że ma do czynienia tylko i wyłącznie ze szkicem aplikacji. Efektem tego jest przyjęcie prototypu jako bazy dla dalszego rozwoju aplikacji. Zachowanie prototypu wprowadza zamieszanie w szeregach pracowników - oprogramowanie, nad którego poprawnością nikt nie pracował, nagle staje się obowiązują bazą produktu. Natomiast F.P. Brooks mówi wręcz o konieczności odrzucenia pierwszego projektu, a nie tylko samego prototypu[15].

Poprzez wspomniane powyżej działanie całkowicie pomija się etap projektowania - jeden z kluczowych czynników sukcesu w projekcie informatycznym.

1.4 Zaniechanie projektowania

Pominięcie projektowania jest pierwszym krokiem ku odrzuceniu jakości kosztem zyskania dodatkowego czasu. Projekt oprogramowania jest tym, czym projekt architektoniczny jest dla budynku. A. Cooper pisze wręcz o przesunięciu projektowania „na później" w sarkastycznym tonie: *„To tak, jakby operator betoniarki powiedział cieślom, że mogą robić formy, jakie tylko chcą, ale dopiero jak on zakończy wylewanie*

[13] A.Cooper, *Wariaci rządzą ...* wyd. cyt., s. 84-89
[14] J.Ridderstråle, K.Nordström, *Funky biznes*, WIG-Press, Warszawa 2001, s. 99
[15] F.P. Brooks, Jr., *Mityczny osobomiesiąc* wyd. cyt., s. 106-107

betonu"[16].

Niemniej jednak etap projektowania bywa odrzucany, ponieważ takie działanie prowadzi do oszczędności czasowych we wstępnych fazach przedsięwzięcia. Jednakże takie oszczędności powodują ogromne spustoszenie w późniejszym czasie[17].

Projektowanie jest bardzo czasochłonnym działaniem składającym się z wielu etapów. Wymaga stosunkowo dużych nakładów pracy przy jednoczesnym braku postępów w postaci działającego kodu programu. Efektem prac projektowych jest zbiór dokumentów, ustaleń oraz szkiców aplikacji[18]. Taka sytuacja może być irytująca. Zarówno klient jak i menedżer nie widzą postępów prac, gdyż nie widzą jej efektów w postaci namacalnego produktu. Przeciwieństwem takiej sytuacji jest prototyp, gdzie złudzenie postępu w pracach jest niemal natychmiastowe. Różnica jest jednak znaczna. Prace nad prototypem to prace odkrywcze. Przyjęte założenia sprawiają, że tak stworzona aplikacja jest konstrukcją niestabilną. Projekt natomiast, to efekt pracy dotyczącej określenia realnych wymagań dotyczących produktu. W trakcie prac projektowych dochodzi do odkrywania faktycznych wymagań użytkowników oraz wykrycia sprzeczności na bardzo wczesnym etapie prac. Koszt wykrycia błędu na etapie projektowania szacuje się na jedną setną kosztu naprawy tego samego błędu na etapie pielęgnacji oprogramowania[19].

G. Booch, guru współczesnej teorii modelowania, pisze o potrzebie modelowania jako rzeczy nieodzownej jakiemukolwiek większemu przedsięwzięciu informatycznemu. Według niego jest to element nieodzowny oprogramowaniu wysokiej jakości[20]. I. Graham, zwraca na ten element uwagę jako jeden z czynników odpowiedzialnych za poprawną analizę problemu:

„W analizie obiektowej i projektowaniu obiektowym zazwyczaj pomija się całkowicie kwestie związane z inżynierią wymagań i pozyskiwaniem wiedzy. (...) przekonamy się, że istnieją głębokie teoretyczne i praktyczne przyczyny, aby się z takim podejściem nie zgadzać"[21].

[16] A. Cooper, *Wariaci rządzą... wyd. cyt.*, s. 123

[17] Tamże, s. 84

[18] D. Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 193

[19] Tamże, s. 11

[20] G. Booch, J. Rumbaugh, I. Jacobson, *UML przewodnik użytkownika*, WNT, Warszawa 2002, s. 3-8

[21] I. Graham, *Metody obiektowe w teorii i w praktyce*, WNT, Warszawa 2004, s. 419

Zaniechanie projektowania jest czynnikiem obniżającym jakość produktu. Obniżenie jakości natomiast wpływa zarówno na morale zespołów wytwarzających ten produkt, jak i wizerunek firmy - *„brak jakości jest nie tylko drogi, jest katastroficzny*[22]*”*.

Do braku jakości, albo inaczej, jej większego obniżenia, prowadzi jeszcze jeden, często stosowany wybiegu menedżerów, odrzucenie testowania.

1.5 Zaniechanie testowania

Testowanie to proces służący podwyższaniu jakości oprogramowania. Celem testowania jest usunięcie wszystkich wad z produktu programowego[23]. Z założenia jest to niemożliwe, niemniej jednak testowanie pozwala na znaczne podwyższenie jakości programu. Chroni przyszłego użytkownika przed przykrymi niespodziankami w postaci błędnych reakcji eksploatowanego systemu. Testowanie zapewnia jakość produktu, ale jednocześnie wymaga dużych nakładów pracy[24] i przede wszystkim wymaga spójnego projektu. Projekt taki stanowi wzorzec dla osób testujących. O skutkach porzucenia projektowania już wspomniano.

Testowanie powinno odbywać się przez cały czas trwania implementacji - *„Testuj wcześnie. Testuj często. Testuj automatycznie*[25]*”*. Wprowadza to oczywiście dodatkowe nakłady pracy. Zwykle, celem zyskania dodatkowego czasu, wprowadza się testowanie jako dodatkowy element harmonogramu (zwykle tuż przed terminem oddania produktu). Jeżeli w takiej sytuacji dojdzie do załamania harmonogramu, to może dojść do zaniechania testowania, a co za tym idzie, znacznego obniżenia jakości produktu.

Produkt, który nie przeszedł testów, jest faktycznie testowany przez użytkownika końcowego. Usterki, które będą występować w trakcie eksploatacji produktu, będą systematycznie obniżać jego wartość w oczach, w tym przypadku skazanych na produkt, użytkowników. Niemniej jednak nie tylko ocena produktu przez klienta jest tutaj istotna. Świadomość, że wytwarzany produkt jest dalece niedoskonały i uciążliwy dla korzystających z niego osób, bardzo niekorzystnie wpływa na morale zespołu

[22] M. Cantor, *Jak kierować... wyd. cyt.*, s. 49

[23] M.E. Bays, *Metodyka wprowadzania oprogramowania na rynek*, WNT, Warszawa 2001, s. 49

[24] R.V. Binder, *Testowanie systemów obiektowych*, WNT, Warszawa 2003, s. 43-46

[25] A. Hunt, D. Thomas, *Pragmatyczny programista* wyd. cyt., s. 270

wytwarzającego ten produkt. Programiści przestają utożsamiać się z własnym dziełem, zaczynając traktować je jako niemiły obowiązek[26]. Zyskanie czasu i dobrego wizerunku w oczach klienta, dzięki zachowaniu harmonogramu, zostaje zamienione na niższą jakość.

1.6 Obniżanie jakości

Wspomniano w poprzednim punkcie, że testowanie jest niezwykle istotnym elementem w procesie wytwarzania aplikacji komputerowych. Jest to w zasadzie jedyny możliwy sposób na stwierdzenie, czy program spełnia wymagania stawiane przez klientów. Jeżeli pominięty zostanie etap testowania, to siłą rzeczy nastąpi obniżenie jakości produktu.

Nawet przy idealnie skonstruowanym projekcie nie ma możliwości zapewnienia 100% jakości[27] oprogramowania. Błędy wprowadzone do kodu programu mogą być po prostu pomyłkami edytorskimi. Takim sytuacjom nie jest się w stanie w pełni przeciwdziałać. Niemniej jednak najtragiczniejszą decyzją dla projektu jest ta, gdy menedżer całkowicie wyklucza etap testów.

Kolejnym czynnikiem wpływającym na jakość jest przyjęcie prototypu aplikacji jako bazy dla dalszego jej rozwoju. Początkujący programiści czują się usatysfakcjonowani takim rozwojem wypadków - ich praca została doceniona i mogą rozwijać to co rozpoczęli. Programiści z większym doświadczeniem zdają sobie sprawę z tego, że nierealny termin wdrożenia aplikacji, to tylko i wyłącznie produkt o niższej jakości[28], natomiast uznanie prototypu jako podstawy dalszych działań to jedna z oznak „oszczędności czasu".

Jeżeli prace są realizowane przy współudziale ludzi będących pragmatykami, obniżenie jakości jest równie tragiczne co stwierdzenie, że projekt nie będzie miał w ogóle miejsca. Inżynierowie, oprócz tej cechy, że dążą do rozdrobnienia problemu, chcą rozwiązywać problemy nietrywialne i chcą to robić dokładnie. Wykroczenie przeciw jakości, jest jakby nie patrzeć, zaprzeczeniem tej idei. Zasadę oporu pracowników wobec niesłusznych idei poznano już około tysiąca lat przed naszą erą: *„Nikt nie sprawi, żeby*

[26] T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 153-154
[27] D.Hamlet, J.Maybee, *Podstawy techniczne...* wyd. cyt., s. 448-450
[28] T.DeMarco, T.Lister, *Czynnik ludzki wyd. cyt.,* s.154

wojsko skutecznie walczyło, jeżeli każe mu się walczyć w niesłusznej wojnie[29]*"*.

Z korzyścią dla firmy jest więc, gdy wytworzenie produktu o wysokiej jakości przyjmie się jako jeden z głównych celów. „*Tylko najlepsze jest wystarczająco dobre*[30]", twierdzą J.Ridderstråle oraz K.Nordström, autorzy *Funky Biznes,* zwracając uwagę na to, że pracownicy umysłowi nie są już zainteresowani *tylko i wyłącznie* pracą. Pracownicy oczekują od firmy możliwości rozwoju oraz zapewnienia im poczucia dumy i radości z tego co robią. Obniżanie jakości produktu prowadzi do tego, że pracownik nie jest w stanie zaspokoić tego rodzaju potrzeb. Co za tym idzie przestaje identyfikować się z firmą oraz wytwarzanym produktem. To z kolei prowadzi do dalszego nakręcania spirali obniżania jakości.

Wymienione w rozdziale pierwszym czynniki, wpływające na niepowodzenie projektu informatycznego, nie są wszystkimi możliwymi. Uwzględnione zostały te, które są stosunkowo łatwe do zidentyfikowania, a mimo to nadal występują w ramach procesów związanych z wytwarzaniem oprogramowania. W rozdziale został położony nacisk na czynnik techniczny w wymiarze społecznym, a nie na czynnik ludzki. Oczywiście ten drugi ma również ogromne znaczenie[31]*.* Zaskakujące jest również to, jak małą wagę przywiązuje się do faktycznego określenia wymagań przyszłych użytkowników, a następnie zaprojektowania rozwiązania przed przystąpieniem do realizacji produktu[32].

Z kolei brak uwzględniania potrzeb pracowników oraz lekceważące do nich podejście prowadzą do konfliktów pomiędzy kierownikami, a programistami, co z kolei doprowadzić może do porażki całego przedsięwzięcia[33].

[29]S.Zi, *Sztuka wojenna*, vis-à-vis Etiuda, Kraków 2003, s. 18
[30]J.Ridderstråle, K.Nordström, *Funky biznes* wyd. cyt., s. 102
[31]T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 20
[32]A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 83
[33]J.Ridderstråle, K.Nordström, Funky biznes wyd. cyt., s. 163

2 Przeciwdziałanie błędnym decyzjom w projekcie informatycznym

Od początku powstania inżynierii oprogramowania ludzie starają się odnaleźć tą jedyną najdoskonalszą receptę na wszystkie bolączki projektów informatycznych. Pragną zapanować nad nimi poprzez stworzenie narzędzia umożliwiającego pełną automatyzację procesu wytwarzania oprogramowania. Fr. P. Brooks twierdzi, że nie istnieje i nigdy nie będzie istniała „brakująca srebrna kula[34]". Mało tego, uważa, że zbytnie zaufanie technice, zamiast zdrowemu rozsądkowi, może doprowadzić każdy projekt do katastrofy.

Mówienie, że nie istnieje uniwersalne rozwiązanie problemów programistycznych, nie jest jednak tym samym co mówienie, że nie ma sposobów na ograniczanie ryzyka i zasięgu oddziaływania skutków szkodliwych decyzji. Na przestrzeni lat zostały opracowane metody projektowania aplikacji, które pozwalają na lepsze rozpoznanie prawdziwych wymagań stawianych systemowi. Należą do nich, między innymi: metoda Boocha, metoda Shlaera-Mellora, metoda CRC (*Class-Responsibilities-Collaboration*)[35].

W roku 1995 zaproponowana została nowa metoda opisu procesów projektowych w dziedzinie inżynierii oprogramowania, określana mianem UML (*Unified Modeling Language*)[36]. Metoda ta ma na celu takie przygotowanie opisu budowanego systemu, aby można było go stosować w projekcie informatycznym - na każdym poziomie abstrakcji. Dzięki tej metodzie można między innymi doprowadzić do zmniejszenia skutków syndromu „użytkownik i programista[37]". Niemniej jednak UML jest tylko językiem[38], natomiast to co jest przede wszystkim niezbędne to realny harmonogram oraz przyjęcie za konieczne stworzenia projektu systemu. O tym, że notacja sama w sobie, albo proces sam w sobie, nie mają kluczowego znaczenia może świadczyć stwierdzenie: „*Niektórzy inżynierowie oprogramowania przysięgają, że narzędzia CASE są podstawową sprawą dla produktywności projektowania. Inni przysięgają, że CASE to*

[34] F.P. Brooks, Jr., *Mityczny osobomiesiąc* wyd. cyt., s. 159
[35] D.Hamlet, J.Maybee, *Podstawy techniczne... wyd. cyt.,* s. 374-390
[36] G.Booch, J.Rumbaugh, I.Jacobson, *UML przewodnik...* wyd. cyt., s. xxii
[37] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 96
[38] szerzej o UML w rozdziale 2.2

strata czasu[39]".

2.1 Tworzenie realnego harmonogramu

Realny harmonogram produktu wiąże się nie tylko z określeniem czasu trwania przedsięwzięcia, ale przede wszystkim ze zdefiniowaniem realnych celów. Typowy przebieg procesu wytwarzania oprogramowania można przedstawić w następujący sposób: Menedżer ustala z klientem czego ten drugi oczekuje od produktu; następnie programiści i projektanci proszeni są o szybki szkic aplikacji. Zostaje stworzony prototyp który jest przedstawiany klientowi jako namacalny dowód możliwości zespołu. Ze względu na ograniczenia czasowe zarówno projektanci jak i programiści muszą pójść na kompromis związany z wymaganiami pierwszych, a możliwościami drugich. Prototyp zostaje przedstawiony klientowi[40].

Największym nieszczęściem jest sytuacja, w której klient stwierdzi, że to co zostało przygotowane, odpowiada jego oczekiwaniom. W takiej sytuacji menedżer dochodzi do dwóch wniosków. Po pierwsze nie było potrzeby tworzenia projektu aplikacji. Skoro dysponuje produktem odpowiadającym oczekiwaniom klienta oznacza to, że zespół programistów w istocie rzeczy nie potrzebuje jakichkolwiek projektów. Drugi wniosek to ten, że skoro w tak krótkim czasie został przygotowany produkt, który zadowala klienta, to w czasie przeznaczonym na całość przedsięwzięcia można wykonać znacznie więcej, niż zakładał pierwotny harmonogram. Należy zwrócić uwagę na to, że takie same odczucia może mieć również klient. Jest to jednak rozumowanie niepoprawne, gdyż prototyp jest tylko „szkicem" aplikacji, a nie produktem końcowym.

Sposób w jaki można rozprawić się z pierwszym problemem to tylko i wyłącznie twarde stanowisko ze strony menedżera. Kierownictwo musi zostać uświadomione, że „budujemy prototypy, aby uczyć się[41]". Przygotowanie prototypu ma na celu wykrycie zależności, których nie jesteśmy pewni. Służy określeniu możliwości technologii, którą zdecydowaliśmy się wykorzystać do budowy oczekiwanego przez klienta rozwiązania. Pozwala na uzyskanie szkicu aplikacji, który następnie można przedstawić klientowi. To czemu na pewno nie służy tworzenie prototypu, to zastępowanie nim procesu projektowania. Zachowanie prototypu ma jeszcze jeden

[39] D.Hamlet, J.Maybee, *Podstawy techniczne...* wyd. cyt., s. 417
[40] A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 87-89
[41] A.Hunt, D.Thomas, *Pragmatyczny programista* wyd. cyt., s. 59

negatywny aspekt. Wprowadza do aplikacji nieporządek i nie pozwala na zachowanie zasady DRY (*Don't Repeat Yourself*)[42]. Zasada ta mówi o tym, by jednoznacznie określić w projekcie miejsce każdego z elementów systemu. Nie zachowanie tej zasady prowadzi do tego, że aplikacja staje się coraz mniej odporna na wszelkie poprawki. W związku z tym w harmonogramie musi znaleźć się miejsce zarówno na przygotowanie prototypu (wykonanie rozpoznania w obszarach nowej technologii) jak i gruntowne zaprojektowanie aplikacji.

Szczególnym przypadkiem tworzenia aplikacji *ah hoc* jest wykorzystanie techniki „pocisków smugowych[43]". Metoda ta polega na przyrostowym tworzeniu produktu i każdorazowej korekcie po konsultacjach z klientem w ramach dostarczenia kolejnej wersji oprogramowania. Wyrażane przez klienta opinie oraz uwagi dotyczące aplikacji są spisywane, a następnie, wynikające z nich poprawki wprowadzane są do wytwarzanego oprogramowania. Po wprowadzeniu wszystkich poprawek produkt jest ponownie prezentowany klientowi, aby ten mógł ocenić, czy to co zostało wprowadzone odzwierciedla jego oczekiwania. Nie ma jednak pewności co do tego czy metoda „pocisków smugowych" jest dobrym sposobem na całkowitą rezygnację z projektu. Metoda ta jest dodatkowo wymagająca ze względu na konieczność częstych kontaktów z klientem, co nie zawsze jest możliwe. Opis metody, ujęty w jednym zdaniu, sprowadza się do stwierdzenia: *dodaj funkcjonalność do produktu, a następnie upewnij się, czy odpowiada ona klientowi.*

Kolejnym krokiem w trakcie konstrukcji harmonogramu jest określenie wymagań udziałowców co do wytwarzanego oprogramowania. Na samym początku należy jednoznacznie zidentyfikować potrzeby klientów, aby móc następnie stwierdzić czy programiści są w stanie je spełnić. Znajomość tych potrzeb pozwala na podjęcie decyzji o tym, które z oczekiwań klientów znajdą swoje odzwierciedlenie w wytwarzanym produkcie w ramach ustalonego harmonogramu. Negocjacje są nieodzownym elementem tego etapu prac nad produktem programowym, a znajomość zasad nimi rządzących jest jak najbardziej pożądana. Proste techniki wywierania wpływu na ludzi mogą okazać się nieocenioną pomocą w trakcie przygotowywania wspólnego stanowiska wykonawcy oraz klienta[44].

[42] A.Hunt, D.Thomas, *Keep It DRY, Shy, and Tell the Other Guy*, IEEE Software 2004, s. 101

[43] A.Hunt, D.Thomas, *Pragmatyczny programista* wyd. cyt., s. 52-57

[44] R.B.Cialdini, *Wywieranie wpływu na ludzi*, GWP, Gdańsk 2001

Sposób określenia rzeczywistych wymagań stawianych produktowi jest zależny zarówno od możliwości firmy jak i wielkości projektu. Z powodzeniem można stosować między innymi warsztaty wymagań, burzę mózgów, rysunkowe szkice ujęć, przypadki użycia[45], odgrywanie ról, czy wreszcie wspomniane już stosowanie prototypów[46].

Występuje jeszcze jeden element wpływający na podniesienie realności harmonogramu. Mowa tutaj o zaangażowaniu osób odpowiedzialnych za realizację przedsięwzięcia w proces ustalania harmonogramu. Po pierwsze uzyskuje się tak zwany efekt „reguły zaangażowania[47]", co prowadzi do poczucia większej odpowiedzialności po stronie inżynierów. Po drugie można dowiedzieć się o tych elementach przedsięwzięcia, które mogą narazić projekt na duże ryzyko niepowodzenia, a tym samym powinny być szczególnie nadzorowane. W końcu to inżynierowie będą wytwarzać produkt i to oni wiedzą najlepiej jakich pułapek można się spodziewać w trakcie jego realizacji. Niemniej jednak wszechobecny optymizm programistów należy skorygować. Mimo iż programiści biorący udział w określeniu czasu wymaganego do zakończenia przedsięwzięcia, przedstawiają swoje propozycje, to należy przyjąć, że średnio są one dwa razy mniejsze, niż rzeczywiste potrzeby dotyczące czasu. O zbytnim optymizmie programistów wspomina w *Mitycznym osobomiesiącu* Fr.P. Brooks[48]. W ustalaniu realnego harmonogramu jak najbardziej należy zasięgać opinii inżynierów, ale jednocześnie należy zdawać sobie sprawę z konieczności korekty rzeczywistego harmonogramu.

Dobrze jest gdy rozmowy toczące się pomiędzy wszystkimi odpowiedzialnymi za przedsięwzięcie osobami toczą się w taki sposób, że każdy ma świadomość tego o czym mówi i że w pełni to rozumie. Dodatkowo warto zadbać o to, by dokumentacja projektu była napisana w taki sposób, że każdy uczestnik będzie mógł się skupić na odpowiednim dla niego poziomie ogólności. Takie zalety posiada między innymi notacja UML[49].

2.2 Wspólny język

Aby ustalić wspólne stanowisko warto upewnić się, że każda ze stron dokładnie rozumie swoje oczekiwania oraz możliwości drugiej strony. Ma to znaczenie w relacjach na każdym stopniu hierarchii komunikacji z klientem. Klient powinien

[45] szerzej o przypadkach użycia w Rozdziale 2.2

[46] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 91-176

[47] R.B.Cialdini, *Wywieranie wpływu...* wyd. cyt., s. 65-109

[48] Fr.P.Brooks,Jr., *Mityczny osobomiesiąc* wyd. cyt., s. 26

[49] G.Booch, J.Rumbaugh, I.Jacobson, *UML przewodnik...* wyd. cyt.

wiedzieć co zostało mu zaoferowane, natomiast menedżer powinien wiedzieć, czego klient oczekuje. Projektant powinien rozumieć oczekiwania menedżera, natomiast menedżer powinien orientować się w technicznej notacji projektu. Projektant powinien rozumieć ograniczenia wynikające z technologicznych aspektów problemu przedstawiane przez inżynierów, natomiast programiści powinni być pewni oczekiwań stawianych oprogramowaniu.

Zaspokojenie tych wszystkich potrzeb wydaje się być stosunkowo trudnym przedsięwzięciem. Klient zwykle wspomina tylko i wyłącznie o swoich potrzebach i nie interesują go w najmniejszym stopniu aspekty techniczne przedsięwzięcia. Menedżer stara się dostosować ofertę stosując, hasła dotyczące technologicznych aspektów przedsięwzięcia, do oczekiwań klienta. Projektanci dążą do prezentowania wszystkich aspektów technologicznych w postaci już to znanych modeli, lub przy pomocy nowych rozwiązań i w zasadzie nie są do końca zainteresowani potrzebami jako takimi. Ważne jest dla nich to, aby technologia była odpowiednio dobrana do produktu. Programiści natomiast są związani tylko i wyłącznie z aspektami związanymi z językiem implementacji. Nie jest dla nich istotne w jaki sposób całość będzie współpracowała. Dla nich ważny jest element nad którym pracują, co związane jest w dużej mierze ze stosowaniem zasady D. Parnasa - ukrywania informacji[50]. Rozwiązaniem tych problemów może być stosunkowo młoda technika zarządzania projektami informatycznymi określana mianem *Unified Modeling Language* (UML).

UML ma na celu zdefiniowane jednoznacznego sposobu komunikacji pomiędzy wszystkimi grupami biorącymi udział w pracach nad produktem. Takie określenie zasad komunikacji pozwala na uzyskanie pewności, że wszystkie strony wiedzą o czym mówią. Język UML został pomyślany w taki sposób, aby każdy udziałowiec mógł w wygodny sposób opisać swoje wymagania. Jest językiem graficznym dzięki czemu nawet osoby nie obeznane z jego szczegółami są w stanie zrozumieć informacje, które czytają. W jego skład wchodzą diagramy, które umożliwiają opisanie założeń na każdym etapie wytwarzania produktu.

Do opisu wymagań stawianych przez klientów wykorzystuje się tak zwane przypadki użycia. Do opisu wewnętrznej struktury aplikacji stosuje się między innymi perspektywę interakcji i czynności, zaś do opisu modelu wdrożenia korzysta się z diagramów

[50]D.Hamlet, J.Maybee, *Podstawy techniczne...* wyd. cyt., s. 253

komponentów. Łącznie do dyspozycji mamy dziewięć typów diagramów, które pozwalają na uzyskanie spojrzenia z różnych perspektyw na ten sam system.

Przypadki użycia są podstawową formą komunikacji z użytkownikiem[51]. W ramach każdego przypadku użycia zapisuje się informacje o tym, co dany przypadek użycia realizuje. Uzyskanie rzeczywistych wymagań oraz zgody co do ich poprawności uzyskuję się poprzez zapisanie scenariuszy wykonania dla każdego z przypadków użycia. Dzięki zastosowaniu opisu słownego każdego z przypadków użycia stosunkowo łatwo można dojść do porozumienia z klientem czy to co zostało ustalone, jest faktycznie tym, czego oczekuje. Nieodzowną jest więc dbałość o dokumentację w przypadku korzystania z przypadków użycia. Musi być ona spisana w sposób jasny i zrozumiały dla obu stron. Jakiekolwiek punkty, nie będące w pełni zrozumiałymi dla którejkolwiek ze stron, powinny być natychmiast wyjaśniane[52].

Ważnym aspektem przypadków użycia jest to, że opisują one to, co ma być zrealizowane, a nie to, jak ma być dana funkcjonalność zrealizowana. Jest to bardzo istotny element tej techniki pozyskiwania wymagań. Bywa, że inżynierowie w ramach przypadków użycia starają się „przemycić" konkretne rozwiązania techniczne co może prowadzić do nieporozumień na dalszym etapie prac oraz prowadzi do braku zrozumienia dokumentu przez klientów.

Wybór formy opisu wszelkich zjawisk związanych z projektem informatycznym jest niezwykle ważny. Warto wykorzystać taką metodę, która pozwala na elastyczność przy jednoczesnym zachowaniu prostoty opisu zjawisk. UML, dzięki zastosowaniu diagramów, pozwala na błyskawiczną ocenę funkcjonalności systemu. Natomiast dzięki zachowaniu możliwości wyrafinowanego opisu szczegółów (język OCL) pozwala rozwinąć skrzydła biegłym projektantom i programistom[53].

UML jest językiem stosowanym do opisu procesów wytwarzania oprogramowania, ale nie uwzględnia wytwarzania rodziny produktów. Do tego typu przedsięwzięć wydają się być lepszymi modele FAST i PASTA[54]. W tej pracy nie będą one opisywane, aczkolwiek warto wiedzieć, że istnieją również metody opisu procesu wytwarzania rodziny produktów.

Sam język jest ważny, jednakże nie wystarczy do osiągnięcia sukcesu

[51]G.Booch, J.Rumbaugh, I.Jacobson, *UML przewodnik...* wyd. cyt., s. 223
[52]D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 158-161
[53]J.Warmer, A.Kleppe, *OCL precyzyjne modelowanie w UML*, WNT, Warszawa 2003
[54]D.M.Weiss, C.T.R.Lai, *Asortyment produktów programowych*, WNT, Warszawa 2003

w przedsięwzięciu. Potrzebne jest jeszcze spoiwo pozwalające na scalenie zespołów wytwarzających produkt.

2.3 Pragmatyzm i jakość

Wspomniano już o wadze etapu projektowania i konieczności jego istnienia. Projekt to element w oparciu o który następuje późniejsza budowa systemu[55]. Niezwykle istotne jest więc, by projekt został przygotowany z należytą dokładnością i uwzględnieniem wszelkich *pułapek* mogących pojawić się w trakcie trwania projektu Oczywistym jest, że nie można przewidzieć wszystkich możliwych zdarzeń, ale można przewidzieć oczywiste punkty stanowiące zagrożenie dla powodzenia projektu, a mimo to je zignorować. Takie działanie jest często wynikiem braku pragmatycznego podejścia i braku dbałości o jakość.

Wspomniano również o tym, że jakość jest niezwykle istotnym czynnikiem motywującym: „*Jakość znacznie wykraczająca poza wymagania użytkownika końcowego jest środkiem prowadzącym do wyższej wydajności*[56]". Dodatkowo jakość wpływa na postawę konkurencji wobec firmy, a co za tym idzie, sprawia że pracownicy po drugiej stronie barykady nie postrzegają „nas" tylko i wyłącznie jako inną firmę programistyczną. „*iVillage ustanowiła wysokie standardy budzące szacunek konkurencji*[57]", wspomina P. Carpenter, pisząc o jej osiągnięciach w walce z konkurencją poprzez podwyższanie jakości oferowanych usług. Pojawia się więc pytanie w jaki sposób można doprowadzić do podwyższenia jakości produktu, a tym samym do podwyższenia morale zespołu wytwarzającego ten produkt w projekcie informatycznym?

Jednym z kluczowych elementów jest wprowadzenie *mody* na jakość. Tyle, że samo stwierdzenie - *Od dziś wytwarzamy produkt wysokiej jakości* - nie wystarczy. Należy przede wszystkim dać możliwości zespołom, aby były w stanie realizować ten postulat.

Zaufanie harmonogramowi oraz unikanie chęci zyskania czasu może być dobrym początkiem. Jeżeli zdecydowano się na przygotowanie prototypu, a następnie w oparciu o ten prototyp przygotowanie projektu, to niech tak zostanie[58]. Przy pierwotnym planie powinno pozostać się nawet wówczas, gdy klient zapewnia nas, że to co widzi

[55] A.Jaszkiewicz, *Inżynieria oprogramowania*, Helion, Gliwice 1997, s. 133
[56] T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 36
[57] P.Carpenter, *e-brands*, WIG-Press, Warszawa 2001, s. 34
[58] A.Hunt, D.Thomas, *Pragmatyczny programista* wyd. cyt., s. 61

w postaci prototypu jest spełnieniem jego oczekiwań i gotowy jest na wcześniejszą dostawę produktu kosztem obniżenia jakości. Menedżer powinien zdawać sobie sprawę, że akceptując takie stanowisko, działa przeciw swoim pracownikom. Będą oni musieli zmagać się z nieprzygotowanym do dalszej pracy materiałem. Tym samym będą systematycznie doprowadzać do obniżenia jego wartości zarówno w oczach klienta jak i w oczach zespołu.

Dbałość o jakość oznacza pilnowanie jakości na każdym kroku wytwarzania produktu. Kluczem zachowania spójnego jakościowo produktu jest unikanie i przeciwdziałanie najdrobniejszym błędom i uchybieniom. Wymaga to uświadomienia pracownikom, że jakość to nie ocena produktu na etapie testowania, tylko dbałość o produkt w każdym momencie.

Bardzo częstą przyczyną pogłębiającej się degradacji produktu jest pozostawianie w nim małych „zarysowań". Są to prozaiczne błędy polegające na nie umieszczaniu komentarzy w kodzie programu, powielaniu źródeł tej samej informacji na potrzeby chwili, zaniechaniu umieszczania opisu słownego diagramów. Te wszystkie błędy są prozaicznymi uchybieniami, i jako takie, same nie stanowią szkody dla produktu. Błędy te mają jednak negatywny wpływ na morale zespołu wytwarzającego produkt. Mianowicie, po upływie czasu każdy zaczyna zauważać mnogość tych *pojedynczych* błędów, które zaczynają być coraz bardziej uciążliwe. Błędy te wprowadzają poczucie braku zainteresowania jakością, co z kolei prowadzi do dalszego jej obniżenia. Każda z osób odpowiedzialnych za produkt przestaje przywiązywać wagę do tego, by produkt był „dobry". Syndrom ten nosi nazwę „*wybitego okna*[59]".

Pragmatyzm to jakość, ale również wiedza. Truizmem wręcz jest wspominanie o konieczności zapewnienia pracownikom możliwości samo rozwoju i doskonalenia się. Jakość nie powstaje z niczego. Jeżeli pracownicy, poprzez swój wkład, mają tworzyć produkt o wysokiej jakości, to muszą dysponować wiedzą, która pozwoli im na osiągnięcie celu. Autorzy *Funky biznes* zwracją uwagę na to, że „*w funkowej firmie najważniejszą rolę odgrywają nie tyle podstawowe kompetencje, co najważniejsi fachowcy*[60]". Oczywiście fachowców można albo pozyskać, albo wyszkolić. Jedno i drugie wiąże się z kosztami. Od menedżera zależy więc, czy jest gotów poświecić jakość kosztem oszczędności, czy też jednocześnie zadba o morale i dobre samopoczucie pracowników

[59] A.Hunt, D.Thomas, *Pragmatyczny programista* wyd. cyt., s. 5
[60] J.Ridderstråle, K.Nordström, *Funky biznes* wyd. cyt., s. 108

oraz jakość produktu.

Kolejnym problemem jest zaufanie do inżynierów. Bywa tak, że menedżer nie potrafi zaufać, programistom, ponieważ uważa, że pracowników należy pilnować, aby osiągali wyniki i realizowali zaplanowany harmonogram. Zaskakujące jest jednak to, że pracownicy czasami wiedzą lepiej jak osiągnąć cel i w jaki sposób do niego dążyć[61]. Dobry kierownik potrafi zaufać swoim ludziom i pozwala im na „szaleństwo" w ramach rozsądku. Zdaje sobie również sprawę z tego, że pracownicy, jeżeli pozwoli im się na swobodną pracę w grupie zapaleńców, zdziałają więcej niż zespoły ograniczone biurokracją i strukturami.

Projekty informatyczne nie są zawieszone w próżni. Istnieje wiele technik, algorytmów i metod, które potrafią ułatwić życie programistom oraz projektantom, a mimo tego nie są stosowane. D. Hamlet zachęca wręcz: „*Zawsze kradnij kod, zamiast pisać go od nowa. Nie kradnij jednak złego kodu oraz kodu, który nie działa*[62]". Być może jest to kwestia dumy, a być może po prostu niewiedzy, ale w projektach informatycznych ani nie uczy się na własnych błędach, ani nie korzysta z cudzych[63].

2.4 Korzystanie z wiedzy innych

Zaskakujące jest to jak często programiści nie zwracają uwagi na istniejące już rozwiązania. Jednym z powodów jest zapewne ten, na który zwraca uwagę T. DeMarco: „*Szczególnie zniechęcająca jest statystyka dotycząca czytelnictwa. Przeciętny programista na przykład nie ma żadnej książki związanej z jego pracą i nigdy takiej nie czytał*[64]". Zapewne może to być jeden z powodów niskiej jakości oprogramowania i niepowodzeń w projektach informatycznych. W końcu „wynajdywanie koła" zajmuje znacznie więcej czasu niż skorzystanie z doświadczeń innych podczas jego wynajdywania.

Istnieją stosunkowo proste techniki podnoszenia jakości programu, a mimo to programiści unikają ich niczym ognia. Podstawową wydaje się być zasada DRY. Mówi ona o konieczności minimalizacji źródeł informacji w produkcie programowym. Czym więcej źródeł informacji tym więcej zależności i tym większa złożoność produktu. Z kolei

[61] T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 186
[62] D.Hamlet, J.Maybee, *Podstawy techniczne...* wyd. cyt., s. 52
[63] T.DeMarco, T.Lister, *Czynnik ludzki* wyd. cyt., s. 17
[64] Tamże, s. 26

złożoność prowadzi do zwiększenia czasu wymaganego do wprowadzenia zmian w programie[65]. Przeprowadzenie szkolenia i wprowadzenie w życie techniki DRY może znacznie podnieść jakość produktu.

Kolejną zasadą jest unikanie „*wybitych okien*[66]". Tutaj jednak wymagana jest nie tylko sama świadomość problemu, ale również mechanizmy kontrolne. Należy tworzyć taką kulturę programowania, w której ocena efektów pracy programistów należy do członków zespołu. Sama świadomość tego, że kod programu będzie oglądany przez innych oraz krytykowany, sprawia, że programiści zaczynają dbać o swój fragment programu.

Kolejnym elementem wpływającym na znaczną ilość błędów w produkcie są ciągłe zmiany w procesie konsolidacji produktu. Bywa, że tuż przed przygotowaniem wersji dla klienta, wprowadza się szereg zmian w procesie generowania finalnego produktu. Prowadzi to do zamieszania i wprowadza kolejny element zagrożenia. Jedyną rozsądną radą jest obarczenie odpowiedzialnością za cały proces konsolidacji określonego członka zespołu. Dzięki temu uzyska się pewność, że proces ten będzie spójny. Dodatkowo, warto wykorzystywać dostępne narzędzia celem automatyzacji całego procesu, tak aby jedynym co musi zrobić człowiek było naciśnięcie przycisku START[67].

Osobną kwestią jest sposób opracowywania oprogramowania. Mimo stosowania języków obiektowych, które z założenia miały służyć do tworzenia systemów mogących być powtórnie użytymi, nie stosuje się takiej praktyki. Efektem jest tworzenie wciąż i wciąż tych samych fragmentów kodu, przez co następuje strata czasu na nieefektywne wykonywanie tej samej pracy tylko „trochę inaczej". Z pomocą może tutaj przyjść programowanie komponentowe. Jego celem jest wykorzystanie metod obiektowych, ale dodatkowo zamknięcie obiektów wewnątrz komponentów z jasno określonymi interfejsami. Dzięki takiemu podejściu można łatwiej i pełniej wykorzystać raz już napisane elementy oprogramowania[68].

W rozdziale drugim zasygnalizowane zostały pewne sposoby przeciwdziałania błędom w zarządzaniu projektem. Błędy te wynikają często z nieznajomości

[65] A.Hunt, D.Thomas, *Keep It DRY...* wyd. cyt., s. 101
[66] A.Hunt, D.Thomas, *Zero-Tolerance Construction*, IEEE Software Sep/Oct 2002, s. 100
[67] A.Hunt, D.Thomas, *Three Legs, No Wobble*, IEEE Software Jan/Feb 2004, s. 18
[68] C.Szyperski, *Oprogramowanie komponentowe*, WNT, Warszawa 2001

technicznych aspektów projektowania[69], czy też lekceważenia konieczności posiadania wspólnej płaszczyzny komunikacji w postaci języka UML[70]. Wspomniano również o lekceważącym podejściu do problemów osób odpowiedzialnych bezpośrednio za wytwarzane oprogramowanie - programistów. Bardzo często lekceważone są podstawowe techniki programistyczne, które są dobrze opracowane i opisane[71], a to prowadzi zarówno do błędów jak i straty czasu.

[69] A.Cockburn, *Jak pisać efektywne przypadki użycia,* WNT, Warszawa 2004

[70] J.Cheesman *Komponenty w UML*, WNT, Warszawa 2004

[71] B.W.Kernighan, R.Pike, *Lekcja programowania,* WNT, Warszawa 2002

3 Proste błędy oraz ich skutki w przedsięwzięciach informatycznych

Niniejszy rozdział zawiera opis przedsięwzięć informatycznych, które zakończyły się w znacznym stopniu porażką. Analizie podlegały trzy przedsięwzięcia informatyczne, w których autor brał udział.

*Przedsięwzięcie **A*** polegało na wytwarzaniu aplikacji multimedialnej. W pracach brał udział zespół złożony z dziesięciu pracowników odpowiedzialnych za różne aspekty aplikacji.

*Przedsięwzięcie **B*** to wdrożenie dużego systemu bankowego w jednym z banków w Polsce. Przedsięwzięcie wymagało współpracy wielu podmiotów oraz określenia znacznej listy wymagań ze strony użytkowników.

*Przedsięwzięcie **C*** to prace nad aplikacją pisaną dla jednego z banków w Polsce. Polegało na przygotowaniu kompleksowego oprogramowania mającego na celu ułatwienie pracy dotychczas realizowanej przez inne oprogramowanie.

3.1 Niezrozumienie użytkownika

Zaskakujące jest to, że mimo szeroko dostępnej literatury na temat pozyskiwania wymagań, projektanci systemów informatycznych nie potrafią jednoznacznie zidentyfikować prawdziwych potrzeb użytkowników. J. Ridderstråle i K. Nordström proponują aby „*co jakiś czas wczuwać się w sytuację klienta i zadawać pytanie:* Co tak naprawdę oni kupują? *Dziewięćdziesiąt dziewięć razy na sto okaże się, że nie to co ty sprzedajesz*[72]". Niestety w projektach informatycznych tego się nie praktykuje. Zakłada się, że wiemy co produkujemy, ale nie upewniamy się, czy jest to prawda. Prowadzi to do takiej sytuacji, że proponowany jest klientom produkt, którego nie potrzebują. Miesiące pracy na daremno tylko dlatego, że nie zapytano klientów o to, czego pragną[73].

W ramach przedsięwzięcia **A** porażka wynikała z tego, że nie odczytano oczekiwań klientów co do produktu. Wprawdzie aplikacja sama w sobie była interesująca, ale brakowało kilku szczegółów, które usprawniłyby pracę z programem. Klienci narzekali na trudności podczas pracy z programem, na jego powolne działanie. Mimo tego nikt

[72] J.Ridderstråle, K.Nordström, *Funky biznes* wyd. cyt., s. 192
[73] A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 103

nie zdawał się zwracać na to uwagi. Tłumaczono, że taka jest polityka firmy i produkt musi w ten sposób działać ze względów bezpieczeństwa. Jednak dla klientów to nie bezpieczeństwo było istotne, a szybkość. Tego właśnie oczekiwali od produktu.

W ramach przedsięwzięcia **B** o ignorowaniu potrzeb użytkowników zadecydowali „wszechwiedzący" pracownicy firmy wdrożeniowej. Zadecydowali bowiem o tym, co będzie dla użytkowników lepsze na podstawie swoich dotychczasowych doświadczeń. Nie było dla nich ważne to, że te cechy, które posiadał dotychczasowy system, są istotne dla pracowników. Jakość oraz nowe możliwości nowego systemu powinny, wedle ich przekonania, z nawiązką wyrównać straty starych funkcjonalności. Można jedynie wyobrazić sobie rozgoryczenie pracowników związane z wyrzucaniem, tego co dobre, w imię „lepszej" jakości. W trakcie prac okazało się, że to co miało być lepsze, posiada znaczne ograniczenia. Niestety w tego typu przedsięwzięciach nie ma odwrotu. Poprzez takie podejście, przyszłych użytkowników skazuje się na nowy produkt zamiast ułatwiać im życie.

Podobne podejście zastosowano w przedsięwzięciu **C**. Podczas wytwarzania produktu w dużej mierze oparto się na rozwiązaniach bazujących na dotychczasowym doświadczeniach. Ze względu na to, że nie uzgodniono faktycznych potrzeb, w trakcie trwania projektu zaczęły pojawiać się problemy wynikające z wadliwej, według klientów, pracy programu. O ile mniej czasu zostałoby zmarnowane, gdyby zapytano klientów, czego tak naprawdę oczekują.

W każdym z wymienionych przypadków wystarczyłoby zdobyć prawdziwe wymagania klientów, chociażby stosując proste techniki szkiców ujęć, czy burzy mózgów[74]. Nie zrobiono tego, lub zrobiono to w taki sposób, że prawdziwe wymagania nie zostały jednoznacznie zidentyfikowane i nazwane.

3.2 Złudne prototypy

Wspomniano już o prototypie jako narzędziu zdobywania wymagań. Mechanizm ten został zastosowany w ramach przedsięwzięcia **C**. Przygotowana została aplikacja mająca na celu wizualizację możliwości programu. Została ona w pełni zaakceptowana przez drugą stronę, a na dodatek przyjęto ją jako bazę produktu. Początkowe korzyści wynikające z oszczędności czasu zastąpione zostały przez pracę nad nieefektywnym

[74] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 129-154

i niemożliwym do zarządzania kodem programu w późniejszych fazach projektu. Na tym samym organizmie prowadzone były zarówno prace doświadczalne, jak i rzeczywiste prace programistyczne. Nagminne ignorowanie zasady DRY[75] doprowadziło do powstania tak wielu powiązań w kodzie programu, że programiści przestali panować nad tym, co stworzyli. Wprowadzenie kolejnej zmiany do programu prowadziło do znacznego multiplikowania istniejącej liczby powiązań. Wprowadzenie poprawki potrafiło generować kilka kolejnych błędów. Praca z kodem źródłowym zaczęła przypominać syzyfową pracę. Gdy program był już prawie gotowy okazywało się, że nowe błędy uniemożliwiają jego poprawną pracę. Nikt więc się nie dziwił, gdy zaczęły pojawiać się głosy proponujące wyrzucenie dotychczasowego kodu aplikacji i rozpoczęcie prac od nowa. Wszystko było wynikiem jednej błędnej decyzji - zachowania prototypu jako bazy dla dalszych prac. Jedyną rozsądną decyzją w przypadku tego przedsięwzięcia byłoby porzucenie dotychczasowego kodu zaraz po prezentacji oraz przystąpienie projektantów do pracy. Tak jednak się nie stało.

W projekcie **B** prezentacja prototypu miała trochę inny przebieg. Firma wdrażająca dysponowała istniejącym już rozwiązaniem. Produkt został przedstawiony w taki sposób, że nie wykazywał wad w trakcie prezentacji. W trakcie prac zaczęły pojawiać się „małe niedogodności", o których wcześniej nie było mowy. Wdrożenie produktu zakończyło się oczywiście „powodzeniem". Pytanie tylko, co mogą powiedzieć o tym użytkownicy systemu. Zatem: czy to, że wdrożono produkt, samo w sobie jest sukcesem? O końcowych użytkownikach zapomniano, a prezentacja produktu w określony sposób sprawiła, że byli oni pewni, iż kupują co innego niż zostało ostatecznie wdrożone[76].

3.3 Pomijanie projektowania

Jak już zostało wspomniane w rozdziale 1.4 zaniechanie projektowania prowadzi do wprowadzenia bałaganu w przedsięwzięciu. Rozpoczyna się pracę, ale nie wie się do czego się dąży. Projekt **A** był projektem o małym rozmiarze, więc można wytłumaczyć tym brak projektowania za pomocą zaawansowanych narzędzi. Zaskakujące jest jednak to, że nie stosowano nawet prostych technik projektowych. Projektantem był zwykle programista, a nie osoba odpowiedzialna za produkt. Prowadziło to do takiej konstrukcji

[75] A.Hunt, D.Thomas, *Keep It DRY...* wyd. cyt., s. 101
[76] A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 31-32

produktu, która była wygodna dla programisty a nie dla użytkownika[77]. Wystarczyło zastosować proste sposoby prezentacji idei, chociażby rysunki na kartce papieru przy pomocy ołówka. Ważne jednak było to, aby końcowy odbiorca w pełni określił swoje oczekiwania. W przeciwnym wypadku dochodzi do głosu syndrom „tak, ale[78]".

Projekt **B** posiadał specjalnie powołany zespół projektowy, ale wydaje się, że zespołowi brakowało podstaw technicznych do opracowania w pełni satysfakcjonującego produktu. Mam tutaj na myśli jednolity i wspólny dla wszystkich stron język. Dla wszystkich jest jasne, że aby ludzie się porozumieli muszą mówić tym samym językiem. Zaskakujące jest to, że w projektach informatycznych nie dość, że nie posługuje się wspólnym językiem, to nie korzysta się z pomocy „tłumaczy". Efektem jest taka sytuacja, w której wszyscy myślą, że się rozumieją, a *de facto* tak nie jest. Wydaje się, że wykorzystanie jednolitego narzędzia komunikacji oraz narzędzi pozwalających na znaczną automatyzację gromadzenia wymagań znacznie wpłynęłyby na poprawę prac projektowych.

Próbę wprowadzenia do harmonogramu fazy projektowej w projekcie **C** należy niestety nazwać tylko próbą. Śladowe działania projektowe nie miały szansy wpłynąć na produkt, gdyż zostały podjęte w trakcie trwania prac programowych[79]. Projekt taki wprowadzał tylko dodatkowe zamieszanie. Z informacji projektowych wynikało co innego niż z programu oraz oczekiwań analityków. Efektem była całkowita rozbieżność pomiędzy projektem a programem. Ratunkiem w tej sytuacji mogło być tylko i wyłącznie odrzucenie prototypu i podjęcie pełnych prac projektowych. Niestety nie podjęto takiej decyzji.

3.4 Ratowanie terminu

Ratowanie terminu ma zwykle na celu uniknięcie konsekwencji wynikających z jego nie dotrzymania. Można dyskutować co jest lepsze: oddanie produktu o wyższej jakości ale przy tym poniesienie konsekwencji z niedotrzymania terminu, czy oddanie produktu o niższej jakości w ramach wyznaczonego terminu. Zarówno jedna, jak i druga opcja, nie stanowi komfortowej sytuacji. Niemniej jednak dotrzymywanie terminów za wszelką cenę może prowadzić do tragicznych w skutkach działań.

[77] A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 76
[78] D.Leffingwell, D.Widrig, *Zarządzanie wymaganiami* wyd. cyt., s. 94
[79] A.Cooper, *Wariaci rządzą...* wyd. cyt., s. 123

Zarówno w przedsięwzięciu **B**, jak i **C**, szacunek do terminu wynikał z upolitycznienia przedsięwzięć. Niedotrzymanie terminów wiązało się z konsekwencjami zapisanymi w kontraktach. Prowadziło to jednak do absurdalnych wręcz sytuacji, gdy kolejna wersja produktu była przygotowywana, mimo swej niedoskonałości, po to tylko, aby zmieścić się w terminie. To samo tyczyło się opracowywanych dokumentów. Waga terminu przysłaniała sens dokumentu czy aplikacji. Efektem były programy, które robiły nie to co trzeba i dokumenty, które opisywały nie to co miały opisywać.

Wydaje się, że w takiej sytuacji jedynym rozsądnym rozwiązaniem jest rozmowa obu stron na temat terminów oraz ustalenie co jest przyczyną opóźnień. Może to zabrzmieć jak utopia, ale celem obu stron powinno być dążenie do uzyskania produktu wysokiej jakości. Można się zastanawiać czy odstąpienie od ustalonej kary nie będzie czasami lepszym rozwiązaniem, niż przypieranie drugiej strony do muru. W końcu to klient najbardziej odczuje wszelkie braki w produkcie. Być może zostanie zachowany termin, ale jakim kosztem?

Dodatkowym efektem ubocznym ratowania terminu jest pojawianie się nadgodzin. Na podstawie doświadczeń autora można z całą pewnością stwierdzić, że nadgodziny są nieodzownym elementem przedsięwzięć informatycznych, w których menedżerowie starają się niedopuścić do przekroczenia terminu. Nadgodziny są w głównej mierze wynikiem nierealnych harmonogramów, które od samego początku skazane są na niepowodzenie. Nie ma znaczenia, jaki projekt jest rozpatrywany. Zaskakujące jest jednak to, że w przytoczonych projektach każdy z harmonogramów był mniej więcej dwa razy krótszy niż rzeczywisty czas potrzebny na realizacje przedsięwzięcia[80]. Te niezwykle optymistyczne harmonogramy doprowadzały do tego, że pracownicy musieli nadrabiać stracony czas pracą w godzinach nadliczbowych. Niestety nie ma dobrego rozwiązania dla źle zaplanowanego harmonogramu. Chyba, że założenia zostaną zrewidowane, a pierwotny harmonogram wydłużony.

Kolejnym aspektem ratowania terminu jest obniżanie jakości. Zaskakujące z jaką łatwością podejmuje się decyzje o obniżeniu jakości produktu kosztem dotrzymania terminu. Nagminnym wydaje się redukowanie możliwości, zamienianie funkcjonalności cechami, pomijanie oczekiwań użytkowników.

Obniżenie jakości wydaje się na pierwszy rzut oka cechą pozytywną. W końcu

[80]Fr.P.Brooks, Jr., *Mityczny osobomiesiąc* wyd. cyt., s. 17

pracownicy nie będą musieli robić tego wszystkiego co byłoby konieczne w przypadku oprogramowania o wyższym standardzie. Jest to jednak złudne. W przedsięwzięciu **C** zdecydowano się na obniżenie jakości poprzez wprowadzanie do oprogramowania zmian *ad hoc*. Poprzez takie potencjalnie pozytywne działanie nagminnie niszczono architekturę systemu. W systemie pojawiły się elementy wcześniej nie przewidziane. Prowadziło to do niszczenia logiki systemu i mniejszej identyfikacji programistów z oprogramowaniem (zasada „*wybitego okna*") oraz nakręcania spirali braku jakości.

Jedynym ratunkiem w takiej sytuacji wydaje się „zatrzymanie" się na chwilę i doprowadzenie programu do porządku. Wymaga to gruntownych porządków i zastanowienia się czy obrany kierunek jest tym pożądanym. Niestety wymaga to czasu oraz pewności po stronie programistów, że jakość jest ważniejsza od terminu.

Zakończenie

W niniejszej pracy zostały zaprezentowane błędy jakie popełniane są w trakcie trwania projektów informatycznych. Mimo tego, że błędy te są stosunkowo łatwe do zidentyfikowania, nadal są popełniane przez menedżerów projektów informatycznych. To co należy stwierdzić z całą stanowczością, to fakt braku kompetencji oraz doświadczenia kierowników przedsięwzięć informatycznych. Niedostrzeganie problemów w trakcie trwania procesu można tłumaczyć albo ignorancją, albo niewiedzą szefów zespołów programistycznych. Niestety, statystyki, które zostały przytoczone, a dotyczyły czytelnictwa wśród programistów, są zastraszające. Brak doświadczenia nie jest rekompensowany wiedzą teoretyczną.

W pracy zwrócono uwagę na konieczność dokształcania pracowników. Jest to praktyka stosowana, aczkolwiek z małym rozmachem. Od pracowników raczej się wymaga, żeby się rozwijali, niż proponuje formy rozwoju.

Niestety również sami pracownicy są winni upadających projektów. Ignorowanie podstawowych zasad dobrego stylu w programowaniu nie pozwala mówić o nich jako ofiarach menedżerów, a raczej jak o współwinnych katastrof.

Zadziwiające jest jednak to, że nie stara się przeciwdziałać tym czynnikom, które zostały wymienione. Sprawia to wrażenie braku zainteresowania projektem zarówno ze strony pracodawców jak i pracowników, a przecież zarówno jednym jak i drugim powinno zależeć na powodzeniu przedsięwzięcia. Pytanie o to, kto powinien zadbać o jakość, pozostanie zapewne otwarte. Pracodawca dysponuje możliwościami określania harmonogramów, negocjowania umów i określania rozmiaru prac. Pracownicy natomiast są bezpośrednio odpowiedzialni za kształt produktu. To im powinno zależeć na tym by był to produkt wysokiej jakości. W końcu jest to ich dzieło. Być może inicjatywa oddolna byłaby w tego typu przedsięwzięciach czymś jak najbardziej pożądanym?

Zaproponowane w pracy sposoby przeciwdziałania błędom menedżerów nie są oczywiście jedynymi rozwiązaniami. Są raczej wynikiem przemyśleń i konfrontacji rzeczywistych projektów z literaturą. Te drobne sugestie, z pozoru łatwe, mogą jednak okazać się niemożliwe do zastosowania, jeżeli w grę wchodzi polityka. Czasami zachowanie terminu może okazać się tak istotne, że poświęcenie jakości jest jedynym rozwiązaniem, mimo tego, że rozpoczyna marsz ku klęsce.

Literatura

[1] M.E. Bays, *Metodyka wprowadzania oprogramowania na rynek*, WNT, Warszawa 2001

[2] R.V. Binder, *Testowanie systemów obiektowych*, WNT, Warszawa 2003

[3] G. Booch, J. Rumbaugh, I. Jacobson, *UML przewodnik użytkownika*, WNT, Warszawa 2002

[4] Fr.P. Brooks, Jr., *Mityczny osobomiesiąc*, WNT, Warszawa 2000

[5] M. Cantor, *Jak kierować zespołem programistów*, WNT, Warszawa 2004

[6] P. Carpenter, *e-brands*, WIG-Press, Warszawa 2001

[7] J. Cheesman, J. Daniels, *Komponenty w UML*, WNT, Warszawa 2004

[8] R.B. Cialdini, *Wywieranie wpływu na ludzi*, GWP, Gdańsk 2001

[9] A. Cockburn, *Jak pisać efektywne przypadki użycia*, WNT, Warszawa 2004

[10] A. Cooper, *Wariaci rządzą domem wariatów*, WNT, Warszawa 2001

[11] T. DeMarco, T. Lister, *Czynnik ludzki*, WNT, Warszawa 2002

[12] M. Drummond, *Zdrajcy Imperium*, WNT, Warszawa 2002

[13] I. Graham, *Metody obiektowe w teorii i w praktyce*, WNT, Warszawa 2004

[14] D. Hamlet, J. Maybee, *Podstawy techniczne inżynierii oprogramowania*, WNT, Warszawa 2003

[15] A. Hunt, D. Thomas, *Pragmatyczny programista*, WNT, Warszawa 2002

[16] A. Hunt, D. Thomas, *Keep It DRY, Shy, and Tell the Other Guy*, IEEE Software 2004

[17] A. Hunt, D. Thomas, *Zero-Tolerance Construction,* IEEE Software Sep/Oct 2002

[18] A. Hunt, D. Thomas, *Mock Objects*, IEEE Software May/Jun 2002

[19] A. Hunt, D. Thomas, *Nurturing Requirements*, IEEE Software Mar/Apr 2004

[20] A. Hunt, D. Thomas, *Software Archaeology*, IEEE Software Mar/Apr 2002

[21] A. Hunt, D. Thomas, *Three Legs, No Wobble*, IEEE Software Jan/Feb 2004

[22] A. Hunt, D. Thomas, *The Art of Enbugging*, IEEE Software Jan/Feb 2003

[23] A. Hunt, D. Thomas, *The Art in Computer Programming*, The Pragmatic Programmers, LLC, 2001

[24] A. Jaszkiewicz, *Inżynieria oprogramowania*, Helion, Gliwice 1997

[25] K. Kelly, *Nowe reguły nowej gospodarki*, WIG-Press, Warszawa 2001

[26] B.W. Kernighan, R. Pike, *Lekcja programowania*, WNT, Warszawa 2002

[27] D. Leffingwell, D. Widrig, *Zarządzanie wymaganiami*, WNT, Warszawa 2003

[28] R. Levine, C. Locke, D. Searls, D. Weinberger, *manifest www.cluetrain.com*, WIG-Press, Warszawa 2000

[29] J. Ridderstråle, K. Nordström, *Funky biznes,* WIG-Press, Warszawa 2001

[30] C. Szyperski, *Oprogramowanie komponentowe*, WNT, Warszawa 2001

[31] J. Warmer, A. Kleppe, *OCL precyzyjne modelowanie w UML*, WNT, Warszawa 2003

[32] D.M. Weiss, C.T.R. Lai, *Asortyment produktów programowych*, WNT, Warszawa 2003

[33] S. Zi, *Sztuka wojenna*, vis-à-vis Etiuda, Kraków 2003