Microsoft

SZYBKIE PROJEKTOWANIE

Zapanuj nad chaosem zadań i presją czasu

Steve McConnell

Helion

Tytuł oryginału: Rapid Development: Taming Wild Software Schedules

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-3270-6

Authorized translation from the English language edition: RAPID DEVELOPMENT:
TAMING WILD SOFTWARE SCHEDULES; ISBN 1556159005; by Steve McConnell;
published by Microsoft Press, a division of Microsoft Corporation, Inc.

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

Spis treści

Rozdział 3.

Klasyczne błędy

Zawartość rozdziału

- 3.1. Przykład popełniania klasycznych błędów
- 3.2. Wpływ błędów na harmonogram projektowania
- 3.3. Lista klasycznych błędów
- 3.4. Ucieczka z „wyspy Gilligana”

Tematy pokrewne

- „Zarządzanie ryzykiem”: rozdział 5.
- „Strategia szybkiego projektowania”: rozdział 2.

PROJEKTOWANIE OPROGRAMOWANIA JEST SKOMPLIKOWANYM DZIAŁANIEM. Typowy projekt daje więcej okazji do wyciągania wniosków z błędów, niż zdarza się to niektórym ludziom przez całe życie. W tym rozdziale przyjrzymy się niektórym z klasycznych błędów, które przytrafiają się w trakcie szybkiego projektowania oprogramowania.

3.1. Przykład popełniania klasycznych błędów

Poniższy przykład przypomina nieco dziecinną układankę, w której należy znaleźć wszystkie przedmioty o nazwach rozpoczynających się na literę „B”. Jak wiele klasycznych błędów jesteś w stanie wyłapać w opisie?

Przykład 3.1. Klasyczne błędy

W pewien kwietniowy, ciepły poranek Mike, lider techniczny w firmie Giga Safe, firmie zajmującej się ubezpieczeniami zdrowotnymi, jadł śniadanie w swoim biurze i spoglądał przez okno.

„Mike, dostałeś dotację na aplikację Giga-Quote! Gratulacje! — zawołał Bill, szef Mike’a w Giga Safe. — Kierownictwu bardzo spodobał się pomysł zautomatyzowania kosztów ubezpieczenia. Popiera również ideę codziennego przesyłania kosztów ubezpieczenia do centrali, dzięki czemu będziemy od razu mieli potencjalnych klientów w bazie danych. Muszę pędzić na spotkanie, ale później omówimy szczegóły. Dobrze się spisałeś z tą propozycją!”.

Mike rozpisał ofertę aplikacji Giga-Quote kilka miesięcy wcześniej, ale dotyczyła ona programu pozbawionego funkcji łączenia się z centralą. No cóż… Teraz otrzymał szansę poprowadzenia projektu aplikacji typu klient – serwer oprawionej w nowoczesny interfejs użytkownika — marzył o tym od zawsze. Otrzymał niemal rok na wykonanie projektu, a to mnóstwo czasu, żeby dodać nową funkcjonalność. Mike podniósł słuchawkę telefonu i zadzwonił do żony: „Kochanie, chodźmy gdzieś dzisiaj na kolację. Mamy powód do świętowania…”.

Mike i Bill spotkali się następnego dnia rano w celu omówienia projektu. „No dobra, Bill, o co tu chodzi? To nie jest do końca ta oferta, nad którą pracowałem".

Bill poczuł ukłucie niepokoju. Mike nie brał udziału w poprawkach, ale nie było czasu, żeby go wtajemniczyć. Gdy tylko kierownictwo zapoznało się z koncepcją aplikacji Giga-Quote, przejęło inicjatywę. „Szefostwu bardzo spodobał się pomysł stworzenia oprogramowania automatyzującego proces zakupu ubezpieczenia zdrowotnego. Dyrektorzy wymagają jednak również automatycznego przesyłania zawartych transakcji do głównego serwera. Chcą też, aby system był gotowy przed wprowadzeniem nowych stawek w dniu 1 stycznia. Przesunęli proponowany przez ciebie termin ukończenia projektu z 1 marca na 1 listopada, co skraca harmonogram do sześciu miesięcy".

Mike przewidywał, że projekt zajmie mu 12 miesięcy. Nie wierzył w szansę powodzenia przy sześciomiesięcznym harmonogramie i powiedział to Billowi. „Czegoś tu nie rozumiem — stwierdził. —Kierownictwo zażądało dodania rozbudowanej funkcjonalności sieciowej i skróciło harmonogram z dwunastu do sześciu miesięcy?".

Bill wzruszył ramionami. „Mam świadomość, że to będzie duże wyzwanie, ale jesteś kreatywny i wierzę, że sobie poradzisz. Szefowie zatwierdzili zaproponowany przez ciebie budżet, a dodanie łączy komunikacyjnych nie powinno być wcale takie trudne. Prosiłeś o trzydziestosześciokrotność wynagrodzenia zespołu i ją dostałeś. Możesz zatrudnić, kogo tylko zechcesz, by powiększyć zespół". Następnie polecił Mike'owi, żeby się spotkał z innymi programistami i ustalił skuteczny sposób terminowego dostarczenia produktu.

Mike spotkał się z Carlem, innym liderem technicznym, i razem zastanawiali się nad sposobami skrócenia czasu projektowania. „Skorzystaj może z języka C++ i programowania obiektowego — zaproponował Carl. — Dzięki temu będziesz wydajniejszy, co powinno pozwolić na zredukowanie harmonogramu o miesiąc bądź dwa". Nasz bohater uznał to za dobry pomysł. Carl wspomniał również o istnieniu narzędzia raportującego, które powinno skrócić czas projektowania o połowę. Projekt zawierał mnóstwo raportów, dlatego obydwa rozwiązania pozwoliłyby ustanowić dziewięciomiesięczny harmonogram. Zakup nowego, szybkiego sprzętu pozwoliłby zyskać dodatkowe kilka tygodni. Gdyby Mike zatrudnił najlepszych programistów w branży, zjechałby z terminem ukończenia projektu do siedmiu miesięcy. To był już całkiem niezły plan. Mike podzielił się tymi koncepcjami z Billem.

„Słuchaj — powiedział Bill — dobrze, że udało się wam skrócić termin do siedmiu miesięcy, ale to wciąż za długo. Zarząd dał jasno do zrozumienia, że zależy im na półrocznym harmonogramie. Nie dali mi wyboru. Mogę wam załatwić nowy sprzęt, ale musisz z zespołem w jakiś sposób zmieścić się z projektem w sześciu miesiącach lub pracować w nadgodzinach, jeśli to pomoże".

Mike uznał, że być może jego pierwotne oszacowanie harmonogramu było wyolbrzymione i że jednak uda mu się zakończyć projekt w pół roku. „W porządku, Bill, wynajmę dwóch podwykonawców do tego projektu. Może uda nam się znaleźć specjalistów w dziedzinie komunikacji, którzy pomogą nam w przesyłaniu danych z komputerów do serwera".

Do 1 maja Mike zebrał zespół. Jill, Sue i Thomas byli solidnymi programistami zatrudnionymi w tej samej firmie, a tak się składało, że nie przydzielono ich w tym czasie do żadnego projektu. Dodał do zespołu dwoje wolnych strzelców — Keiko i Chipa. Keiko miała doświadczenie zarówno w komputerach stacjonarnych, jak i w typie serwera, na którym zespół opierał strukturę systemu. Jill z Thomasem przeprowadzili rozmowę kwalifikacyjną z Chipem i odradzali przyjęcie go do zespołu, ale Mike'owi zaimponowała jego wiedza. Znał się na komunikacji i był dostępny, więc pomimo sprzeciwów zatrudniono go.

Na pierwszym wspólnym spotkaniu Bill powiedział zespołowi, że aplikacja Giga-Quote ma strategiczne znaczenie dla firmy. Pewni wysocy szczeblem pracownicy będą im się uważnie przyglądać. Jeżeli zespołowi się powiedzie, zarząd nie będzie skąpił nagród. Przyznał, że wierzy w możliwości poszczególnych członków.

Po przemowie motywacyjnej Billa Mike zasiadł z zespołem i wprowadził wszystkich w szczegóły harmonogramu. Zarząd dał im coś na kształt specyfikacji i mieli dwa tygodnie na wypełnienie wszelkich luk. Następne sześć tygodni mieli poświęcić na stworzenie schematu aplikacji, po czym zostałyby im cztery miesiące na jej stworzenie i testowanie. Intuicja podpowiadała Mike'owi, że program w ostatecznej postaci zawierałby ok. 30 tys. wierszy kodu napisanego w języku C++. Pozostali członkowie zgodzili się z tym. Wyzwanie było ambitne, ale wszyscy od początku byli tego świadomi.

W następnym tygodniu Mike spotkał się ze Stacy, główną testerką. Wyjaśniła, że powinni przysłać prototypy aplikacji do testowania nie później niż do 1 września, a gotową wersję przygotować do 1 października. Mike wyraził zgodę.

Zespół szybko uwinął się ze specyfikacją wymagań i zabrał się za tworzenie schematu aplikacji. Przygotowali zarys programu, który mógł we właściwy sposób korzystać z możliwości języka C++.

Do 15 czerwca mieli gotowy zarys aplikacji, co oznaczało, że skończyli ten etap przed czasem i jak szaleni zabrali się za programowanie, gdyż chcieli zdążyć z wypuszczeniem pierwszej wersji testowej przed 1 września. Nie wszystko szło tak gładko, jak należy. Jill i Thomas nie przepadali za Chipem, Sue również narzekała, że chłopak nie dopuszcza nikogo do swojej partii kodu. Mike uważał, że programiści spędzali ze sobą zbyt wiele czasu w pracy, co stanowiło przyczynę konfliktu charakterów. Bez względu na animozje na początku lipca stwierdzili, że mają przygotowane już 85 – 90% aplikacji.

W połowie sierpnia dział aktuarialny wydał stawki obowiązujące na przyszły rok i zespół stwierdził, że musi dostosować system do zupełnie nowej struktury stawek. Nowa metoda obliczania kosztów wprowadzała pytania o takie czynniki jak częstotliwość aktywności fizycznej, uzależnienie od tytoniu i alkoholu, zainteresowania oraz inne elementy, których nie było wcześniej we wzorach na stawkę ubezpieczeniową. Zespół Mike'a pomyślał, że struktura języka C++ powinna ich uchronić przed wpływem takich zmian. Zamierzali po prostu dodać nowe współczynniki do tabeli kosztów. Musieli jednak pozmieniać wygląd okien dialogowych, projekt bazy danych, dostęp do niej, a także obiekty komunikacyjne, żeby przystosować całość do nowej struktury. Zespół zabrał się za wprowadzanie poprawek, a Mike powiedział Stacy, że mogą zaliczyć kilka dni poślizgu, zanim przygotują wersję przeznaczoną do testowania.

Wersja testowa nie była gotowa 1 września, a Mike zapewnił Stacy, że zostanie dostarczona w ciągu maksymalnie dwóch dni.

Dni zmieniły się w tygodnie. Termin oddania gotowego projektu do testowania (1 października) nadszedł i przeminął. Programiści ciągle nie przygotowali pierwszej wersji testowej. Stacy umówiła się z Billem na przedyskutowanie harmonogramu. „Nie otrzymaliśmy jeszcze żadnej wersji do testowania — powiedziała. — Mieliśmy ją dostać do 1 września, ale nic takiego nie nastąpiło. Wygląda na to, że mają przynajmniej jeden miesiąc poślizgu. Obawiam się, że trapią ich jakieś kłopoty".

„Żebyś wiedziała, że mają kłopoty — odparł Bill. — Muszę porozmawiać z zespołem. Obiecałem sześciuset agentom, że do pierwszego listopada otrzymają tę aplikację. Musimy ją wypuścić przed zmianą stawek".

Bill zwołał spotkanie zespołu. „Jesteście świetną ekipą i powinniście wywiązywać się ze swoich zobowiązań — powiedział im. — Nie wiem, co poszło źle, ale oczekuję od was wszystkich ciężkiej pracy oraz że dostarczycie oprogramowanie na czas. Ciągle możecie otrzymać nagrody, ale musicie sobie na nie zapracować. Od teraz aż do zakończenia projektu każde z was jest zobowiązane do dziesięciogodzinnego dnia pracy przez sześć dni w tygodniu". Po spotkaniu Jill i Thomas burknęli Mike'owi, że nie trzeba traktować ich jak dzieci, ale zgodzili się pracować w wyznaczony sposób.

Zespół przesunął termin zakończenia projektu o dwa tygodnie, gwarantując, że przygotują gotowy produkt do 15 listopada. Testerzy mieliby sześć tygodni na jego sprawdzenie, zanim w styczniu zostałyby wprowadzone nowe stawki.

Pierwsza wersja testowa aplikacji pojawiła się cztery tygodnie później — 1 listopada. Członkowie zespołu zebrali się, aby omówić kilka istniejących problemów.

Tomas pracował nad generowaniem raportów i natrafił na przeszkodę. „Strona z podsumowaniem zawiera prosty wykres kolumnowy. Korzystam z generatora raportów, który jest w stanie go wyświetlać, ale przekształca całą stronę w taki wykres. Jednym z wymogów jest umieszczenie wykresu oraz tekstu na tej samej stronie. Odkryłem, że mogę oszukać system, umieszczając tekst raportu jako legendę w obiekcie wykresu. Jest to oszustwo, ale po wydaniu pierwszej wersji mogę spróbować wprowadzić jakieś bardziej formalne rozwiązanie".

Mike odparł: „Nie wiem, w czym widzisz problem. Mamy wydać produkt i nie mamy czasu, żeby doszlifowywać kod. Bill wyraził się wystarczająco jasno, że nie możemy sobie pozwolić na kolejne poślizgi. Oszukaj system".

Chip poinformował, że jego moduł komunikacyjny jest gotowy w 95%, ale jeszcze musi przeprowadzić kilka testów. Mike zauważył, jak Jill i Tomas przewracają oczami, ale zignorował to.

Zespół pracował ciężko do 15 listopada, w tym niemal przez całe dwie noce tuż przed końcem terminu, ciągle jednak nie był w stanie wydać ostatecznej wersji. Ludzie byli wycieńczeni, ale to Bill poczuł się bardzo źle rankiem 16 listopada. Stacy poinformowała go telefonicznie, że zespół programistów nie przesłał jej poprzedniego dnia aplikacji do przetestowania. Tydzień wcześniej Bill zapewnił kierownictwo, że wszystko idzie zgodnie z planem. Inna kierowniczka projektu, Claire, przyjrzała się postępom czynionym przez zespół i stwierdziła, że doszły ją słuchy o poślizgach w terminowym dostarczaniu aplikacji do testowania. Bill nie lubił Claire, ponieważ uważał, że jest zbyt nerwowa. Zapewnił ją, że jego zespół nie ma problemu z harmonogramem.

Bill kazał Mike'owi zebrać zespół, a gdy przyjrzał się ludziom, stwierdził, że wyglądają na pokonanych. Półtora miesiąca 60-godzinnych tygodni pracy zebrało swoje żniwo. Mike zapytał, o której godzinie tego dnia można się spodziewać gotowego produktu, ale odpowiedziała mu cisza. „Co chcecie mi przez to powiedzieć? — zapytał. — Skończymy dzisiaj tę ostateczną wersję, prawda?".

„Słuchaj, Mike — zaczął Tomas — mogę oddać dzisiaj swój moduł i powiedzieć, że jest »gotowy«, ale prawdopodobnie czekałyby mnie po tym trzy tygodnie jego poprawiania". Mike zapytał, co Tomas ma na myśli, mówiąc „poprawianie". „Nie wstawiłem logo firmy na każdej stronie, nie zdążyłem też wstawić danych kontaktowych agenta w stopce wydruku. To nic wielkiego. Wszystkie najważniejsze funkcje działają jak należy. Mój moduł jest gotowy w dziewięćdziesięciu dziewięciu procentach".

„Ja też jeszcze nie przygotowałam wszystkiego do końca — dodała Jill. — Moja poprzednia grupa wydzwaniała ostatnio często do mnie z prośbą o wsparcie techniczne i codziennie pomagałam im przez bite dwie godziny. Do tego Tomas właśnie mi przypomniał, że mieliśmy udostępnić agentom możliwość dodawania swoich danych osobowych do raportów. Jeszcze nie zaimplementowałam okien dialogowych odpowiedzialnych za tę funkcję, a także kilku innych elementów interfejsu. Nie sądziłam, że będą one potrzebne w tym momencie".

Teraz z kolei Mike'owi zrobiło się słabo. „Jeżeli nic mi się nie stało z uszami i dobrze słyszę, to chcecie mi powiedzieć, że będziemy mieli gotowy, funkcjonalny produkt za trzy tygodnie. Zgadza się?".

„*Najwcześniej* za trzy tygodnie" — sprecyzowała Jill. Pozostali programiści przytaknęli. Mike obszedł stół i zapytał z osobna każdego członka zespołu, czy skończy swoje zadania w ciągu trzech tygodni. Wszyscy stwierdzili, że przy odpowiednio ciężkiej pracy jest to osiągalne.

Tego samego dnia po długiej, nieprzyjemnej rozmowie Mike i Bill zgodzili się przedłużyć harmonogram o trzy tygodnie, do 5 grudnia, pod warunkiem że wszyscy członkowie zespołu będą pracować nie po 10, a po 12 godzin dziennie. Bill wyjaśnił, że szefostwo chce widzieć presję wywieraną na zespół. Zmodyfikowany harmonogram oznaczał, że firma musiała jednocześnie testować aplikację i szkolić agentów ubezpieczeniowych — był to jedyny sposób, aby 1 stycznia wypuścić aplikację na rynek. Stacy narzekała, że dział kontroli jakości nie zdąży w tym czasie przetestować produktu, ale została przegłosowana.

Zgodnie z ustaleniami 5 grudnia twórcy aplikacji Giga-Quote oddali przed południem produkt do testowania i wyszli z pracy wcześniej na zasłużony odpoczynek. Pracowali niemal bez przerwy od 1 września.

Dwa dni później Stacy opublikowała pierwszą listę błędów i rozpętało się piekło. W ciągu tych dwóch dni zespół testerów wykrył ponad 200 usterek, w tym 23 błędy zaliczane do najwyższej kategorii — „konieczne do naprawienia". „Nie wyobrażam sobie, żeby to oprogramowanie było gotowe do przekazania agentom pierwszego stycznia — podsumowała. — Prawdopodobnie tyle czasu zajmie mojemu zespołowi napisanie testów regresyjnych dla już odkrytych usterek, a z każdą chwilą znajdujemy kolejne".

Mike zwołał zebranie zespołu na ósmą rano następnego dnia. Programiści byli rozdrażnieni. Twierdzili, że pomimo kilku poważnych problemów wiele zgłoszonych błędów wcale nie było błędami, a wynikiem niewłaściwego zrozumienia działania programu. Tomas jako przykład podał błąd nr 143: „W raporcie dotyczącym tego błędu stwierdzono, że na stronie podsumowania wykres słupkowy powinien się znajdować po prawej stronie, a nie po lewej. Tego nie można nazwać błędem najwyższej kategorii. To klasyczny przykład przewrażliwienia działu kontroli jakości".

Mike przekazał każdemu członkowi zespołu kopię raportu. Programiści mieli przeanalizować zawarte tam problemy i oszacować czas potrzebny na ich wyeliminowanie.

Na popołudniowym spotkaniu nie usłyszał dobrych wieści. „Jeżeli mam być realistką, to usunięcie zgłoszonych do tej pory błędów zajmie mi około dwóch tygodni — powiedziała Sue. — Do tego muszę dokończyć algorytmy sprawdzania poprawności bazy danych. Łącznie czekają mnie cztery tygodnie pracy".

Tomas przesłał błąd numer 143 do ponownej analizy, domagając się zmiany jego priorytetu z najwyższej kategorii na priorytet trzeciego poziomu — „zmiany kosmetyczne". Dział kontroli jakości wyjaśnił, że raporty podsumowań w aplikacji Giga-Quote powinny wyglądać tak samo, jak raporty generowane przez aplikację odpowiedzialną za przedłużanie polis, które z kolei

miały taki sam styl jak stosowane od lat materiały promocyjne. Agenci ubezpieczeniowi byliprzyzwyczajeni do materiałów zawierających wykres słupkowy po prawej stronie, musiał więc on pozostać w tym miejscu. Utrzymano najwyższy priorytet błędu, co spowodowało dalsze problemy.

„Pamiętasz to oszustwo z wyświetlaniem wykresu i raportu na tej samej stronie? — zapytał Tomas. — Żeby umieścić wykres po prawej stronie, muszę całkowicie przeredagować ten konkretny raport, co oznacza, że muszę stworzyć moduł odpowiedzialny za formatowanie tekstu i grafikę". Mike się wzdrygnął i poprosił o przybliżony czas tworzenia modułu. Tomas odpowiedział, że zajmie mu to przynajmniej 10 dni, ale musi jeszcze dokładniej się zastanowić nad problemem, żeby się upewnić.

Przed pójściem do domu Mike oznajmił Stacy i Billowi, że zespół będzie pracował w trakcie przerwy świątecznej i usunie wszystkie zgłoszone błędy do 7 stycznia. Bill powiedział, że spodziewał się tego i zatwierdził czterotygodniowy poślizg przed wyruszeniem na trwającą miesiąc wycieczkę na Karaiby, którą planował już od poprzednich wakacji.

Mike spędził cały miesiąc, trzymając swoich żołnierzy w ryzach. Przez cztery miesiące pracowali tak ciężko, jak mogli, i sądził, że już się nie da mocniej ich przycisnąć. Przebywali w biurze po 12 godzin dziennie, ale dużo czasu spędzali na czytaniu gazet, opłacaniu rachunków i rozmowach telefonicznych. Wydawało się, że do furii doprowadza ich każde pytanie dotyczące czasu potrzebnego do usunięcia kolejnych błędów. Na jeden wyeliminowany błąd przypadały dwa nowe. Usterki, których usunięcie powinno zająć kilka minut, miały wpływ na cały projekt, przez co programiści głowili się nad nimi przez wiele dni. Po niedługim czasie zrozumieli, że wyeliminowanie wszystkich błędów do 7 stycznia jest nierealne.

Tego dnia Bill wrócił z urlopu i Mike przekazał mu informację, że zespół będzie potrzebował dodatkowe cztery tygodnie na usunięcie wszystkich usterek. „Chyba żartujesz — odparł Bill. — Mam na głowie sześciuset agentów, którzy mają dość wodzenia za nos przez bandę komputerowców. Zarząd rozważa możliwość anulowania projektu. Musisz w jakiś sposób dostarczyć ten produkt w ciągu dwóch tygodni, bez względu na koszty".

Mike zwołał zebranie zespołu, żeby rozważyć możliwości. Przekazał im ultimatum Billa, a następnie poprosił, żeby oszacowali przybliżoną datę wydania produktu, najpierw w tygodniach, a później — miesiącach. Nikt się nie odezwał. Żaden członek zespołu nie chciał ryzykować podania terminu, którego prawdopodobnie i tak nie mogliby dotrzymać. Mike nie wiedział, co powiedzieć Billowi.

Po spotkaniu Chip oznajmił Mike'owi, że podpisał kontrakt z inną firmą, który zacznie obowiązywać od 3 lutego. Mike po raz pierwszy poczuł, że może byłoby lepiej, gdyby projekt rzeczywiście został anulowany.

Mike zlecił Kipowi, programiście odpowiedzialnemu za architekturę serwera, pomoc w pozostałych modułach i przydzielił go do eliminacji błędów występujących po stronie komunikacyjnej komputerów klienckich. Kip walczył przez tydzień z kodem napisanym przez Chipa i zrozumiał, że zawiera on pewne podstawowe wady koncepcyjne, przez które moduł nie ma prawa właściwie działać. Został zmuszony do ponownego zaprojektowania i zaimplementowania klienckiej części łącza komunikacyjnego.

W połowie lutego odbyło się spotkanie zarządu, na którym Claire stwierdziła, że usłyszała już wystarczająco wiele i zaproponowała „zaprzestanie" projektu Giga-Quote. Spotkała się w piątek z Mike'em. „Ten projekt wymknął się spod kontroli — zaczęła. — Od miesięcy nie

dostałam od Billa żadnych rzetelnych wytycznych do harmonogramu. Miał to być projekt trwający pół roku, a już zaliczyliście trzymiesięczny poślizg bez perspektywy ukończenia go w najbliższym czasie. Przyjrzałam się statystykom błędów i twój zespół nie radzi sobie z ich usuwaniem. Pracujecie już tak długo, że teraz nie czynicie żadnych postępów. Daję wam wszystkim wolny weekend; natomiast po powrocie masz sporządzić szczegółowy raport, który ma zawierać wszystkie, powtarzam: *wszystkie*, elementy pozostałe do ukończenia projektu. Nie chcę, abyś na siłę wymyślał jakiś sztuczny harmonogram. Jeżeli tworzenie aplikacji zajmie kolejne dziewięć miesięcy, muszę o tym wiedzieć. Przygotuj ten raport do środowego popołudnia. Nie musi być elegancki, ale ma być kompletny".

Zespół był bardzo zadowolony z wolnego weekendu i członkowie w ciągu następnego tygodnia przystąpili do sporządzania raportu ze zdwojoną mocą. W środę wylądował na biurku Claire. Dała go do przeanalizowania Charlesowi — konsultantowi do spraw inżynierii oprogramowania, który również przejrzał raport błędów. Zalecił, aby zespół skupił się na kilku najwrażliwszych modułach, natychmiast przygotował analizę wszystkich błędów projektowych i programistycznych, a także żeby programiści przestali pracować w nadgodzinach, dzięki czemu mogliby dokładnie oszacować, ile wysiłku zostało włożonego w projekt oraz ile jeszcze zostało czasu do jego zakończenia.

Trzy tygodnie później, w pierwszym tygodniu marca, lista błędów po raz pierwszy zaczęła się kurczyć. Morale zespołu nieco się podniosło, a konsultant na podstawie jednostajnych postępów oszacował termin dostarczenia gotowego i przetestowanego produktu do 15 maja. Kolejny wzrost stawek miał wejść w życie 1 lipca, dlatego Claire ustaliła 1 czerwca jako oficjalną datę wydania aplikacji.

Epilog

Program Giga-Quote dostarczono agentom zgodnie z planem, 1 czerwca. Został przyjęty ciepło, chociaż z pewną dozą sceptycyzmu.

Korporacja Giga Safe doceniła ciężką pracę włożoną w projekt przez członków zespołu, dając każdemu z nich premię w wysokości 250 dolarów. Kilka tygodni później Tomas wziął długi urlop, a Jill przeniosła się do innej firmy.

Stworzenie aplikacji Giga-Quote zajęło ostatecznie nie sześć, a 13 miesięcy — poślizg wyniósł ponad 100% pierwotnego planu. Liczba osobomiesięcy (wliczając nadgodziny) wyniosła 98 miesięcy, czyli została przekroczona o 170% w porównaniu do pierwotnie zakładanych 36 osobomiesięcy.

Odnośniki: Więcej informacji na temat orientacyjnego szacowania terminu wykonania projektów o różnych rozmiarach znajdziesz w podrozdziale 8.6. „Orientacyjne szacowanie harmonogramu".

Wyliczono, że ostateczny produkt składał się z ok. 40 tys. niepustych, niebędących komentarzami wierszy kodu napisanego w języku C++, co stanowiło wartość o 33% większą od rozmiaru zakładanego przez Mike'a. Aplikacja stanowiła hybrydę produktu biznesowego i „celofanowego", ponieważ została rozprowadzona do 600 punktów w kraju. Produkt tego typu i wielkości jest tworzony przeciętnie przez 11,5 miesiąca oraz wymaga włożenia wysiłku równego 71 miesięcy pracy zespołowej. Jak widać, zostały przekroczone obydwie wartości nominalne.

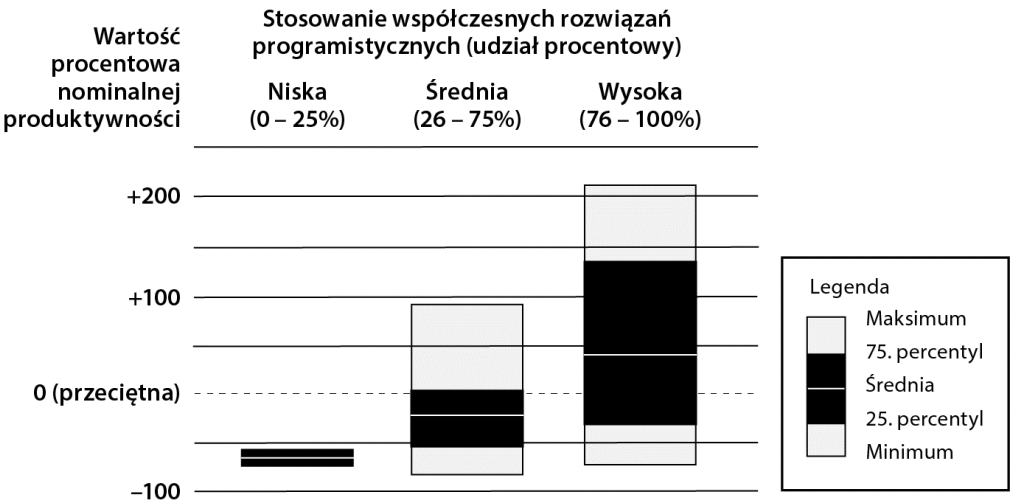
3.2. Wpływ błędów na harmonogram projektowania

Michael Jackson (piosenkarz, nie informatyk) śpiewał, że „jedno zgniłe jabłko nie zepsuje pozostałych, maleńka”[1]. Być może jest to prawda w przypadku jabłek, ale nie dla oprogramowania. Jedno zgniłe jabłko *może* zepsuć cały projekt.

Grupa naukowców z instytutu technicznego ITT przeanalizowała 44 projekty z dziewięciu krajów w celu ustalenia wpływu 13 czynników na produktywność (Vosburgh i in. 1984). Obserwowano takie czynniki, jak wykorzystanie współczesnych rozwiązań programistycznych, poziom zaawansowania kodu, wymagania dotyczące wydajności, udział klienta w ustalaniu specyfikacji produktu, doświadczenie pracowników i kilka innych. Badacze podzielili czynniki na kategorie związane z małą, średnią i dużą wydajnością. Przykładowo: rozdzielili czynnik *współczesne rozwiązania programistyczne* na *niewielkie*, *średnie* i *częste* zastosowanie. Na rysunku 3.1 widzimy wyniki badań przeprowadzonych na czynniku *współczesne rozwiązania programistyczne*.

Im dłużej przyglądasz się wykresowi z rysunku 3.1, tym robi się on bardziej interesujący. Widoczny na nim wzór odpowiada zasadniczo wszystkim analizowanym czynnikom. Naukowcy odkryli, że projekty należące do kategorii składających się na niską produktywność rzeczywiście nie były zbyt wydajne — słupek ukazany w kategorii *niska* na rysunku 3.1. Jednakże produktywność projektów o wysokiej wydajności cechowała olbrzymia rozbieżność — słupek w kategorii *wysoka*. W tej kategorii wydajność projektów plasowała się pomiędzy *niską* a *znakomitą*.

Odnośniki: Więcej informacji na temat omawianego wykresu znajdziesz w podrozdziale 4.2. „Podstawy techniczne”.



Rysunek 3.1. Wnioski dotyczące czynnika „wykorzystanie współczesnych rozwiązań programistycznych” (Vosburgh i in. 1984). Właściwe wdrożenie kilku elementów wcale nie gwarantuje szybkiego projektowania. Trzeba również unikać popełniania błędów

[1] W oryginale: „One bad apple don`t spoil the whole bunch, baby” z piosenki *One Bad Apple* — *przyp. tłum.*

Odnośniki: Więcej informacji dotyczących wpływu błędów na proces szybkiego projektowania znajdziesz w podrozdziale 2.1. „Ogólna strategia szybkiego projektowania".

Nie powinno dziwić, że projekty o niskiej produktywności osiągnęły takie a nie inne wyniki. Jednak zaskakujący jest fakt, że wiele projektów zorientowanych na wysoką produktywność w rzeczywistości kiepsko sobie radzi. Celem powyższego i innych wykresów umieszczonych w tej książce jest ukazanie, że skuteczne rozwiązania są warunkiem koniecznym, ale niewystarczającym do osiągnięcia maksymalnej szybkości projektowania. Nawet jeżeli stosujesz pewne elementy właściwie (np. wydajnie korzystasz z nowoczesnych rozwiązań programistycznych), wciąż możesz popełnić błąd, który zniweluje korzystny wpływ danego czynnika.

Przy zastosowaniu strategii szybkiego projektowania pojawia się kusząca myśl, że wystarczy zidentyfikować i wyeliminować przyczyny powolnego tworzenia, a produkt powstanie znacznie szybciej. Problem polega na tym, że istnieją również inne czynniki wpływające na szybkość projektowania i ostatecznie wyeliminowanie podstawowych przyczyn na wiele się nie zda. Przypomina to trochę pytanie: „Dlaczego nie jestem w stanie przebiec jednego kilometra w trzy minuty?". Jestem za stary. Za dużo ważę. Nie mam dobrej kondycji. Nie chce mi się trenować. Nie mam światowej klasy trenera ani odpowiedniego obiektu treningowego. Nawet za młodu nie byłem wystarczająco szybki. Przyczyny można wymieniać w nieskończoność.

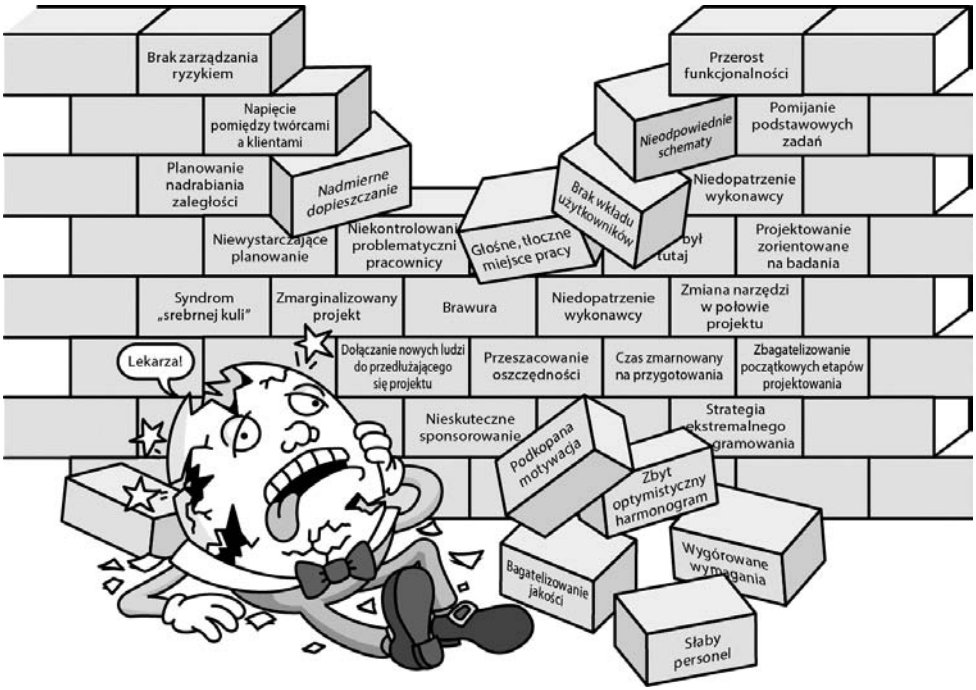
Gdy mówimy o wyjątkowych osiągnięciach, pojawia się zbyt wiele przyczyn, dla których ludzie nie osiągają najwyższej formy. Omawiany w przykładzie 3.1 zespół twórców aplikacji GigaQuote popełnił wiele błędów trapiących świat programistów od zarania dziejów informatyki. Ścieżka inżynierii oprogramowania jest bardzo wyboista, a od tych wybojów w dużej mierze zależy, jak szybko jesteś w stanie zaprojektować oprogramowanie.

W przypadku oprogramowania jedno zgniłe jabłko może zepsuć pozostałe, maleńka. Wystarczy popełnić jeden naprawdę poważny błąd, aby zlecieć do czeluści powolnego programowania; w celu osiągnięcia szybkiego projektowania musisz wystrzegać się popełniania *jakichkolwiek* większych pomyłek. W następnym podrozdziale przyjrzymy się najczęściej popełnianym dużym błędom.

3.3. Lista klasycznych błędów


Klasyczny błąd

Pewne nieskuteczne rozwiązania projektowe są wybierane tak często, przez tak wiele osób i zawsze dają tak samo marne wyniki, że zasługują na nazwanie ich „klasycznymi błędami". Większość błędów przybrana jest kuszące szaty. Czy musisz uratować spóźniający się projekt? Zatrudnij więcej osób! Czy chcesz skrócić harmonogram? Wymuś większy wysiłek! Czy jeden z pracowników irytuje pozostałych? Poczekaj do końca projektu i wylej go! Czy musisz się śpieszyć z wydaniem projektu? Zatrudnij wszelkich dostępnych programistów i zacznij działać od zaraz!



Rysunek 3.2. Projekt był najeżony błędami i żaden z kierowników ani liderów nie był w stanie go uratować

Zazwyczaj programiści, kierownicy i klienci mają dobre powody, aby podejmować określone decyzje, a kuszący wygląd klasycznych pułapek stanowi jedną z przyczyn tak częstego ich popełniania. Jednak z powodu tak dużej częstości występowania takich błędów można bardzo łatwo przewidzieć ich skutki. Klasyczne błędy bardzo rzadko przynoszą zamierzone efekty.

Opisuję w tym podrozdziale niemal 40 rodzajów klasycznych błędów. W swojej karierze natrafiłem przynajmniej raz na każdy z nich, a sam mam kilka na sumieniu. Wiele z nich pojawia się w przykładzie 3.1. Wspólnym mianownikiem wszystkich wymienionych tu pomyłek jest fakt, że ich uniknięcie niekoniecznie powoduje skrócenie czasu projektowania, ale ich popełnianie z pewnością go wydłuża.

Jeżeli niektóre z tych błędów wydają się podejrzanie znajome, nie martw się — popełnia je również wiele innych osób. Gdy zrozumiesz ich wpływ na szybkość projektowania, będziesz mogła/mógł wykorzystać tę listę przy sporządzaniu planu projektu oraz zarządzaniu ryzykiem.

Poważniejsze błędy zostały dokładniej omówione w innych częściach książki, innym z kolei nie poświęcam tak wielkiej uwagi. W celu zwiększenia przejrzystości podzieliłem ich przynależność do poszczególnych wymiarów decydujących o szybkości projektowania: czynnika ludzkiego, procesu, produktu i technologii.

Czynnik ludzki

Poniżej prezentuję niektóre klasyczne błędy związane z ludźmi.

Odnośniki: Więcej informacji na temat prawidłowego i nieprawidłowego stosowania motywacji znajdziesz w rozdziale 11. „Motywacja".

1. Podkopana motywacja. Poszczególne badania wskazywały, że czynnikiem mającym prawdopodobnie największy wpływ na produktywność i jakość jest motywacja (Boehm 1981). W przykładzie 3.1 kierownictwo podjęło czynności systematycznie podkopujące motywację w czasie trwania projektu — rozpoczęło się od płomiennej przemowy, następnie w środku zmuszono programistów do pracy w nadgodzinach, jeden z dyrektorów wyjechał na urlop w ciężkim dla zespołu momencie, na koniec zaś członkowie zespołu otrzymali premię równowartości nieprzekraczającej jednego dolara za każdą nadgodzinę.

2. Słaby personel. Zaraz po motywacji prawdopodobnie największy wpływ na produktywność mają indywidualne umiejętności poszczególnych członków oraz zdolność do pracy zespołowej (Boehm 1981; Lakhanpal 1993). Zatrudnianie pracowników o niskich kompetencjach zagraża wszelkim próbom wprowadzenia strategii szybkiego projektowania. W przykładzie 3.1 pracownicy byli dobierani pod kątem dostępności, a nie pracowitości. Początek tworzenia produktu może przebiegać zrazu dość szybko, ale zazwyczaj takie tempo nie utrzymuje się do końca.

Odnośniki: Więcej informacji na temat tworzenia wydajnych zespołów znajdziesz w rozdziale 12. „Praca zespołowa".

3. Niekontrolowani, stwarzający problemy pracownicy. Brak kontroli nad sprawiającymi problemy pracownikami również zagraża szybkości projektowania. Jest to dość powszechny kłopot, dogłębnie przeanalizowany od czasu wydania w 1971 r. książki *Psychology of Computer Programming* autorstwa Geralda Weinberga. Najczęstszym zarzutem, jakim są obarczani liderzy przez członków zespołu, jest nieumiejętne radzenie sobie z krnąbrnym pracownikiem (Larson i LaFasto 1989). W przykładzie 3.1 zespół wiedział, że Chip jest „zgniłym jabłkiem", ale lidera to nie obchodziło. Rezultat — całkowite przerabianie modułu Chipa — był do przewidzenia.

Odnośniki: Więcej informacji na temat brawury i projektów opartych na zaangażowaniu znajdziesz w podrozdziale 2.5. „Alternatywna strategia szybkiego projektowania", w punkcie„Sporządzanie harmonogramu zorientowane na poświęcenie" (podrozdział 8.5) oraz w rozdziale 34. „Wspólny cel".

4. Brawura. Niektórzy inżynierowie oprogramowania kładą duży nacisk na indywidualne osiągnięcia w projekcie (Bach 1995). Uważam, że takie brawurowe podejście przynosi więcej szkody niż pożytku. W przykładzie 3.1 kierownik projektu skupił się na wprowadzaniu pozytywnego nastawienia, a nie na systematycznych, stabilnych postępach i tworzeniu konstruktywnych raportów. W wyniku tego tworzono harmonogramy będące na krawędzi możliwości realizacji, przez co poślizgi nie były zauważane, potwierdzane i przekazywane zarządowi aż do ostatniej chwili. Niewielki zespół programistów oraz jego bezpośredni przełożony blokowali całą firmę, ponieważ nie potrafili przyznać, że mają problem z dotrzymaniem terminów. Nacisk kładziony na brawurę zmusza do podejmowania olbrzymiego ryzyka i osłabia współpracę pomiędzy różnymi oddziałami pracującymi nad projektem.

Niektórzy kierownicy dążą do brawurowych zachowań, gdy polegają na zbyt pozytywnym nastawieniu. Jeżeli w wyniku tego nastawienia zespół przestanie tworzyć rzetelne, czasem negatywne raporty, kierownicy sami sobie uniemożliwiają korekcję kierunku, w jakim dąży cały projekt. Zasadniczo nie wiedzą nawet, że muszą podjąć jakieś kroki, dopóki szkoda nie zostanie

już wyrządzona. Tom DeMarco stwierdził, że nazbyt pozytywne nastawienie może przekształcić małe utrudnienia w prawdziwą katastrofę (DeMarco 1995).

Odnośniki: Alternatywne sposoby ratowania przedłużającego się projektu znajdziesz w rozdziale 16. „Ratowanie projektu".

5. Dołączanie ludzi do przedłużającego się projektu. To jest prawdopodobnie najbardziej klasyczny z klasycznych błędów. Gdy grozi poślizg, włączenie nowych ludzi do projektu może spowodować zredukowanie produktywności dotychczas zaangażowanych pracowników zamiast jej zwiększenia dzięki obecności „świeżej krwi". Fred Brooks porównał ten błąd do gaszenia pożaru za pomocą benzyny (Brooks 1975).

Odnośniki: Więcej informacji na temat wpływu otoczenia na produktywność znajdziesz w rozdziale 30. „Środowisko pracy".

6. Hałaśliwe, zatłoczone biuro. Większość pracowników ocenia swoje warunki pracy jako niezadowalające. W ok. 60% przypadków są one niewystarczająco ciche lub nie zapewniają prywatności (DeMarco i Lister 1987). Pracownicy przebywający w cichych, prywatnych pomieszczeniach są znacznie bardziej produktywni od kolegów po fachu zajmujących głośne, zatłoczone boksy lub wspólne pomieszczenia.

Odnośniki: Więcej informacji na temat dobrych relacji z klientami znajdziesz w rozdziale 10. „Projektowanie zorientowane na klienta".

7. Napięcie pomiędzy programistami a klientami. Napięcie występujące pomiędzy programistami a klientami może się pojawić z kilku powodów. Klienci mogą uznać, że programiści nie chcą współpracować, gdyż nie zamierzają się zgodzić na proponowany harmonogram lub nie są w stanie spełnić danych obietnic. Z kolei programiści mogą stwierdzić, że klienci nierozsądnie żądają zmieszczenia się w nierealnym terminie lub żądają wprowadzenia zmian już po ustaleniu wymagań projektu. Czasami mogą się pojawiać tarcia następujące w wyniku różnic charakteru.

Podstawowym skutkiem tego napięcia jest słaba komunikacja, a w wyniku słabej komunikacji zespół nie rozumie wymogów, projektuje niewłaściwy interfejs użytkownika, w najgorszym przypadku klienci odrzucają produkt. Często napięcie pomiędzy programistami a klientami sięga takiego poziomu, że obydwie strony są gotowe zrezygnować z projektu (Jones 1994). Zniwelowanie napięcia zajmuje bardzo dużo czasu i nie pozwala się skupić na właściwej pracy.

Odnośniki: Więcej informacji na temat definiowania oczekiwań znajdziesz w podrozdziale 10.3. „Zarządzanie oczekiwaniami klienta".

8. Wygórowane oczekiwania. Jednym z podstawowych źródeł napięcia powstającego pomiędzy programistami a klientami są wygórowane oczekiwania tych drugich. W przykładzie 3.1 Bill nie miał żadnych podstaw, aby oczekiwać stworzenia produktu w ciągu sześciu miesięcy, ale kierownictwo firmy zażądało takiego terminu. To, że Mike nie był w stanie przekonać zarządu do bardziej realnego harmonogramu, stanowi główne źródło wszystkich pojawiających się później kłopotów.

W innych przypadkach kierownicy projektów lub programiści sami proszą się o kłopoty, otrzymując dotację na podstawie nazbyt optymistycznego harmonogramu. Czasami obiecują niemożliwy do zrealizowania zestaw funkcji.

Chociaż wygórowane założenia same w sobie nie wydłużają procesu twórczego, mogą sprawiać wrażenie, że harmonogram jest zbyt luźny, a to niemal równie szkodliwe zjawisko. Organi-

zacja Standish Group opublikowała listę pięciu najważniejszych czynników mających wpływ na sukces projektów biznesowych, wśród których znalazły się realistyczne oczekiwania (Standish Group 1994).

9. Brak sponsora z prawdziwego zdarzenia. Obecność bogatego sponsora jest niezbędna do rozwoju wielu aspektów szybkiego projektowania, takich jak realistyczne planowanie, kontrola zmian, a także wprowadzanie nowych rozwiązań projektowych. Bez wpływowego patronatu inni członkowie zarządu mogą wymusić niemożliwy do realizacji harmonogram lub skłonić zespół do wprowadzenia zmian psujących cały projekt. Pochodzący z Australii konsultant Rob Thomsett twierdzi, że brak patrona na poziomie zarządu grozi porażką projektu (Thomsett 1995).

10. Brak zaangażowania ze strony wszystkich zainteresowanych. Wszystkie najważniejsze osoby biorące udział w tworzeniu projektu muszą czuć wiarę w sens jego realizacji. Dotyczy to sponsora, lidera zespołu, członków zespołu, personelu z działu promocji, użytkowników ostatecznych, klientów oraz każdego, który ma w danym projekcie jakiś interes. Bliska współpraca pojawiająca się w przypadku zaangażowania wszystkich zainteresowanych stron umożliwia precyzyjną koordynację strategii szybkiego projektowania.

11. Brak wpływu użytkowników. Zgodnie z badaniem przeprowadzonym przez organizację Standish Group głównym powodem sukcesu systemów informatycznych jest wkład użytkowników w powstawanie tych produktów (Standish Group 1994). Projekty, w których opinie użytkowników nie są brane pod uwagę na wczesnych etapach, grożą niewłaściwym zrozumieniem wymogów projektowych oraz są narażone na przerost funkcjonalności w dalszych etapach.

Odnośniki: Więcej informacji na temat prawidłowej polityki znajdziesz w podrozdziale 10.3. „Zarządzanie oczekiwaniami klienta".

12. Przedkładanie polityki nad treść. Larry Constantine opisał cztery zespoły mające cztery różne orientacje polityczne (Constantine 1995a). „Politycy" skupiali się na „zarządzaniu" — utrzymywali dobre stosunki z kierownikami. „Badacze" zajmowali się wyszukiwaniem i gromadzeniem informacji. „Izolacjoniści" byli bardzo skryci i tworzyli granice projektu, poza które nie wpuszczali nikogo spoza zespołu. „Generaliści" robili wszystko po trochu: utrzymywali kontakty z zarządem, przeprowadzali badania i zbierali dane, a także koordynowali swoje działania z pozostałymi zespołami. Zgodnie z wynikami badania początkowo najbardziej doceniane przez zarząd były zespoły „polityków" i „generalistów", ale po półtora roku ten pierwszy został najszybciej rozwiązany. Przedkładanie polityki ponad wyniki jest zabójcze dla projektowania zorientowanego na szybkość.

13. Pobożne życzenia. Zawsze mnie zdumiewało, jak wiele problemów w dziedzinie inżynierii oprogramowania wynika z pobożnych życzeń. Ile razy słyszałaś/słyszałeś podobne stwierdzenia od różnych osób:

> „Żaden członek zespołu nie wierzył w stworzenie projektu w wyznaczonym terminie, ale uznali, że jeśli każdy będzie ciężko pracował i wszystko pójdzie zgodnie z planem, a do tego jeśli jeszcze uśmiechnie się do nich los, to może zdołają zdążyć".
>
> „Nie przyłożyliśmy się do skoordynowania interfejsów pomiędzy różnymi modułami aplikacji, ale dobrze się dogadywaliśmy w innych aspektach produktu, a kwestia interfejsów jest w miarę nieskomplikowana, zatem w kilka dni usuniemy wszystkie błędy".

> „Wiemy, że wynajęliśmy najtańszego podwykonawcę do stworzenia podsystemu bazodanowego i aż w oczy kłuł brak doświadczenia poszczególnych pracowników. Nie mieli tak wielkiej wprawy, jak inni podwykonawcy, ale może brak doświadczenia nadrobią entuzjazmem. Prawdopodobnie zdążą z podsystemem na czas".
>
> „Nie musimy pokazywać klientowi najnowszych zmian wprowadzonych do prototypu. Jestem pewien, że już wiem, czego klient od nas oczekuje".
>
> „Zespół twierdzi, że zmieszczenie się w terminie będzie wymagało nieprawdopodobnego wprost wysiłku i już zaliczył kilkudniowy poślizg z niektórymi celami, ale myślę, że chłopaki jeszcze zdążą".

Pobożne życzenia to nie tylko optymizm. To zasłanianie oczu i liczenie na łut szczęścia w momencie gdy nie powinno się opierać na przypadku. Wprowadzone na początku projektu pobożne życzenia wywołują na jego końcu olbrzymi chaos. Dewastują skuteczne planowanie i być może stanowią źródło większej liczby problemów niż wszystkie pozostałe pomyłki razem wzięte.

Proces

Błędy związane z procesem spowalniają projektowanie, ponieważ powodują marnowanie wysiłków i talentu pracowników. Poniżej przedstawiam najgorsze błędy dotyczące procesu.

Odnośniki: Więcej informacji na temat nadmiernie optymistycznych harmonogramów znajdziesz w podrozdziale 9.1. „Tworzenie zbyt optymistycznych harmonogramów".

14. Nadmiernie optymistyczne harmonogramy. Wyzwania stojące przed zespołem tworzącym aplikację w ciągu trzech miesięcy są zupełnie inne od wyzwań dotyczących projektu trwającego cały rok. Przygotowanie zbyt optymistycznego harmonogramu skazuje projekt na porażkę z powodu niedocenienia jego rozmiarów, niewłaściwego planowania, i skrócenia czynności początkowych, takich jak analiza wymagań i przygotowanie szkieletu aplikacji. Ponadto taki harmonogram wywiera olbrzymią presję na programistów, co powoduje długoterminowy uszczerbek na morale i produktywności pracownika. Było to podstawowe źródło problemów w przykładzie 3.1.

Odnośniki: Więcej informacji na temat zarządzania ryzykiem znajdziesz w rozdziale 5. „Zarządzanie ryzykiem".

15. Niewystarczające zarządzanie ryzykiem. Pewne błędy nie występują na tyle często, żeby można je było uznać za klasyczne. Noszą one nazwę „ryzyko" lub „zagrożenie". Podobnie jak w przypadku z klasycznymi błędami, jeżeli nie będziesz aktywnie zarządzać ryzykiem, wystarczy, że tylko jedna rzecz pójdzie nie tak, żeby przekształcić szybkie projektowanie w powolne. Nieumiejętność zarządzania ryzykiem jest klasycznym błędem.

Odnośniki: Więcej informacji na temat podwykonawców znajdziesz w rozdziale 28. „Zewnętrzni podwykonawcy".

16. Niewywiązanie się podwykonawcy ze zlecenia. Jeżeli firma bardzo się śpieszy z jakimś projektem, czasami zleca jego część zewnętrznemu podwykonawcy. Podwykonawcy często jednak spóźniają się z dostarczeniem swojego produktu, charakteryzuje się on niską jakością lub nie zawiera wymaganych funkcji (Boehm 1989). Takie ryzyko jak niska wydajność lub nieprawidłowy interfejs zawsze staje się bardziej prawdopodobne, gdy jest wynajmowany zewnętrzny podwykonawca. Przy nieodpowiedniej komunikacji taki podwykonawca zamiast przyśpieszyć projekt, może go tylko spowolnić.

Odnośniki: Więcej informacji na temat planowania znajdziesz w punkcie „Planowanie" (podrozdział 4.1).

17. Niedokładne planowanie. Jeżeli nie zaplanujesz dobrze strategii szybkiego projektowania, nigdy jej nie wdrożysz.

Odnośniki: Więcej informacji na temat planowania pod presją znajdziesz w podrozdziale 9.2. „Harmonogram pod presją" oraz w rozdziale 16. „Ratowanie projektu".

18. Zaprzestanie planowania pod wpływem presji. Zespoły projektowe często sporządzają plany, a następnie je porzucają, gdy tylko rozpoczną się problemy z harmonogramem (Humphrey 1989). Problemem nie jest samo porzucenie planu, ale niezdolność do stworzenia jego zamiennika, co powoduje przejście w tryb nieprzemyślanego pisania kodu. W przykładzie 3.1 zespół porzucił pierwotny plan tuż po pierwszym poślizgu — jest to typowe zachowanie. Od tego momentu praca stała się niezorganizowana i uciążliwa do tego stopnia, że Jill zaczęła pracować dla swojego poprzedniego zespołu i nikt tego nie zauważył.

19. Czas zmarnowany na przygotowania. „Przygotowaniami" nazywam czas jeszcze przed rozpoczęciem projektu, który standardowo jest przeznaczony na zatwierdzenie i zbieranie budżetu. Dość powszechną praktyką jest spędzanie miesięcy, a nawet lat na przygotowaniach, po czym zaczyna się wyścig z napiętym harmonogramem. O wiele łatwiej, taniej i bezpieczniej jest dobrze wykorzystać te kilka tygodni lub miesięcy, niż zostawić sobie czynności przygotowawcze na później przy harmonogramie trwającym równie długo co faza przedwstępna.

Odnośniki: Więcej informacji na temat bagatelizowania początkowych etapów znajdziesz w podrozdziale 9.1. „Tworzenie zbyt optymistycznych harmonogramów".

20. Zbagatelizowanie początkowych etapów projektowania. Wykonywane w pośpiechu projekty zazwyczaj mają usuwane pozornie najmniej istotne elementy projektowe, a takimi łatwymi celami są analiza, tworzenie architektury i układu projektu, ponieważ w tym czasie nie jest bezpośrednio pisany kod. Wziąłem się kiedyś za ratowanie katastrofalnego projektu; gdy poprosiłem o pokazanie schematów, odpowiedziano mi, że „nie mieli czasu na ich przygotowanie".

Skutki tego błędu — zwanego również „niezaplanowanym programowaniem" — są bardzo przewidywalne. W przykładzie 3.1 wysokiej jakości projekt modułu został zastąpiony oszustwem programistycznym (umieszczenie na jednej stronie wykresu i treści raportu). Przed wypuszczeniem produktu na rynek, oszukany kod tak czy siak musiał zostać usunięty i należało poświęcić czas na opracowanie właściwego rozwiązania. Projekty, w których pracownicy starają się zaoszczędzić w początkowych etapach, zazwyczaj generują na późniejszych etapach 10 – 100 razy większe koszty w porównaniu do podobnego projektu, w którym należycie przeanalizowano poszczególne moduły i sporządzono odpowiednie schematy (Fagan 1976; Boehm i Papaccio 1988). Jeżeli nie potrafisz na początku projektu poświęcić pięciu godzin na wykonanie niezbędnej pracy, skąd weźmiesz 50 godzin na poprawianie w dalszych etapach?

21. Nieodpowiednie przygotowanie schematów. Szczególnym przypadkiem zbagatelizowania początkowych etapów jest nieodpowiednie sporządzenie schematów. W projektach tworzonych pod presją czasu często nie poświęca się wystarczającej uwagi na dobór odpowiedniego środowiska, w którym można w szybki sposób przygotować alternatywne, przemyślane schematy tworzenia aplikacji. Często podczas sporządzania schematu kładziony jest nacisk na doraźne potrzeby, a nie na jakość, dlatego zazwyczaj kończy się to na konieczności przeprowadzenia kilku czasochłonnych cyklów projektowych, zanim cały system zostanie całkowicie ukończony.

Odnośniki: Więcej informacji na temat kontroli jakości znajdziesz w podrozdziale 4.3. „Podstawy kontroli jakości".

22. Zbagatelizowanie kontroli jakości. Wykonywane w pośpiechu projekty niejednokrotnie cierpią od spłycania takich etapów, jak analizowanie konstrukcji i kodu aplikacji, omijania planów testowania oraz przeprowadzania jedynie powierzchownych testów. W przykładzie 3.1 marginalnie potraktowano analizę konstrukcji i kodu oprogramowania w celu zmieszczenia się w wyznaczonym harmonogramie. Okazało się, że gotowy produkt ma zbyt wiele błędów, żeby można go było wydać, a poprawianie zajęło kolejne pięć miesięcy. To jest bardzo typowy rezultat. Pomięcie jednego dnia poświęconego na testowanie podczas początkowych etapów projektowania w późniejszym terminie może kosztować 3 – 10 dni dodatkowej pracy (Jones 1994). Taki „skrót" zdecydowanie przekreśla strategię szybkiego projektowania.

Odnośniki: Więcej informacji na temat mechanizmów zarządzania znajdziesz w punkcie „Śledzenie" (podrozdział 4.1) oraz w rozdziale 27. „Rozbijanie celów na podetapy".

23. Niewystarczające mechanizmy zarządzania. W przykładzie 3.1 wykorzystano zbyt mało mechanizmów zarządzania do ostrzegania w odpowiednim momencie o nieubłaganych poślizgach, a te, które były od początku stosowane, zostały porzucone, gdy pojawiły się problemy. Zanim stwierdzisz, że projekt zmierza we właściwym kierunku, musisz najpierw określić, czy w ogóle znajduje się na torach.

Odnośniki: Więcej informacji na temat przedwczesnej konwergencji znajdziesz w ustępie „Przedwczesna konwergencja (zbieżność)" (podrozdział 9.1).

24. Przedwczesna lub zbyt często występująca konwergencja. Zanim zostaną ustalone terminy wydania gotowego produktu, pojawiają się naciski, aby go odpowiednio przygotować — poprawić wydajność, wydrukować dokumentację, wdrożyć system pomocy, dopracować instalator, usunąć funkcje, które nie zostaną przygotowane na czas itd. W przypadku projektów tworzonych w pośpiechu istnieje tendencja do przedwczesnego łączenia tych elementów. Niemożliwa jest konwergencja produktu w dowolnym momencie, dlatego wiele zespołów stara się ją wymusić przynajmniej kilkakrotnie, zanim odniosą sukces. Te dodatkowe iteracje konwergencji nie wpływają dobrze na produkt. Zabierają jedynie cenny czas i niepotrzebnie wydłużają harmonogram.

Odnośniki: Listę powszechnie omijanych zadań znajdziesz w ustępie „Nie pomijaj najpowszechniejszych zadań" (podrozdział 8.3).

25. Pomijanie podstawowych zadań. Jeżeli ludzie zignorują sporządzanie dokładnych raportów z poprzednich projektów, zapomną o mniej oczywistych zadaniach, których lista będzie się wydłużać. W wyniku pomijania zadań harmonogram zostaje wydłużony o jakieś 20 – 30% (van Genuchten 1991).

Odnośniki: Więcej informacji na temat ponownego szacowania znajdziesz w punkcie „Przekalibrowanie" (podrozdział 8.7).

26. Planowanie nadrabiania zaległości. Jednym z rodzajów ponownego szacowania jest niewłaściwe reagowanie na poślizg. Jeżeli pracujesz nad projektem trwającym pół roku, a spełnienie celu wyznaczonego na dwa miesiące zajmuje Ci trzy miesiące, to co robisz? Wiele planów zakłada nadrobienie zaległości w późniejszym terminie, ale tak się nigdy nie dzieje. Wraz z tworzeniem projektu dowiadujesz się o nim coraz więcej, w tym również coraz dokładniej szacujesz czas potrzebny na jego ukończenie. Ta wiedza powinna zostać wykorzystana do ponownego oszacowania harmonogramu.

Inny rodzaj pomyłki związany z ponownym szacowaniem wynika ze zmian produktu. Jeżeli parametry tworzonego przez Ciebie produktu są modyfikowane, wydłuża się również czas potrzebny na wprowadzenie tych zmian. W przykładzie 3.1 główne założenia projektu zmieniły się jeszcze przed rozpoczęciem prac, co nie zostało odzwierciedlone w harmonogramie ani zasobach. Dodawanie nowych funkcji bez zmiany terminu ukończenia projektu musi się skończyć poślizgiem.

Odnośniki: Więcej informacji na temat strategii nieprzemyślanego pisania kodu znajdziesz w podrozdziale 7.2. „Nieprzemyślane pisanie kodu".

27. Strategia ekstremalnego programowania. Niektóre firmy uważają, że drogą do szybkiego projektowania jest błyskawiczne, swobodne i niepilnowane pisanie kodu aplikacji. Jeżeli programiści są odpowiednio zmotywowani i doświadczeni, pokonają każdą przeszkodę. Jak się przekonasz podczas dalszej lektury, nie ma bardziej mylnego założenia. Strategia taka jest często przedstawiana jako „świadcząca o przedsiębiorczości", ale w rzeczywistości stanowi tylko fasadę starego paradygmatu nieprzemyślanego tworzenia kodu połączonego z ambitnym harmonogramem — a taka kombinacja nie sprawdza się prawie nigdy. Jest to doskonały przykład na ukazanie, że dwa złe rozwiązania wcale nie przynoszą dobrego efektu.

Produkt

Poniżej przedstawiam klasyczne błędy związane z definiowaniem produktu.

28. Nadmiar wymagań. Niektóre projektu od samego początku obarczone zostają zbyt wysokimi wymaganiami. Zbyt często wydajność zostaje wymieniona jako niezbędny wymóg, co zwykle powoduje niepotrzebne wydłużenie harmonogramu. Nieraz użytkownicy są mniej zainteresowani zaawansowanymi funkcjami niż twórcy czy dział promocji, a taka złożona funkcjonalność nieproporcjonalnie spowalnia proces tworzenia aplikacji.

Odnośniki: Więcej informacji o przeroście funkcjonalności znajdziesz w rozdziale 14. „Regulowanie zestawu funkcji".

29. Przerost funkcjonalności. Nawet jeżeli uda Ci się uniknąć nadmiaru wymagań, są one modyfikowane w czasie trwania przeciętnego projektu średnio o 25% (Jones 1994). Taka zmiana może spowodować wydłużenie harmonogramu również o 25%, co stanowi katastrofę w przypadku projektów zorientowanych na szybkie tworzenie.

Odnośniki: W podpunkcie „Niejasne lub niemożliwe do spełnienia cele" (podrozdział 14.2) znajdziesz przykład opisujący zupełnie przypadkowe wystąpienie samowoli programisty w trakcie trwania projektu.

30. Samowola programistów. Programiści fascynują się nowymi technologiami i czasami dążą do tego, aby wypróbować nowe funkcje języka lub środowiska programistycznego albo stworzyć własną wersję świetnej funkcji podpatrzonej u konkurencji — chociaż w danym projekcie jest ona zupełnie niepotrzebna. Wysiłek włożony w przygotowanie, wdrożenie, przetestowanie, stworzenie dokumentacji oraz obsługę niepotrzebnych funkcji powoduje tylko zwiększenie ryzyka poślizgu.

31. „Przepychane" negocjacje. Jedną z dziwniejszych taktyk negocjacyjnych jest odwlekanie terminu przez kierownika zbyt wolno tworzonego projektu po to tylko, aby po wprowadzeniu nowego terminu dorzucić kolejne zadania. Trudno zrozumieć takie postępowanie, ponieważ kierownik zatwierdzający nowy harmonogram pośrednio przyznaje, że pierwotny termin był błędny. Jednakże ta sama osoba po zatwierdzeniu nowego harmonogramu podejmuje jawne

działania dążące do kolejnego poślizgu. Takie działanie w żaden sposób nie pomaga w dotrzymaniu harmonogramu.

32. Projektowanie zorientowane na badania. Seymour Cray, twórca superkomputerów marki Cray, stwierdził, że nie zamierzał skupiać się na więcej niż na jednoczesnym rozwoju dwóch obszarów inżynierii, ponieważ dalsze zwiększenie ich liczby niebezpiecznie przybliża prawdopodobieństwo porażki (Gilb 1988). Wielu programistów powinno czerpać naukę od pana Craya. Jeżeli w swoim projekcie musisz się zająć rozwojem informatyki poprzez stworzenie nowego algorytmu lub nowych rozwiązań obliczeniowych, nie projektujesz aplikacji, tylko skupiasz się na badaniach. Harmonogram inżynierii oprogramowania jest przewidywalny w dość dużym stopniu, natomiast harmonogramu badań nie da się w najmniejszym stopniu ustalić.

Jeżeli wśród celów projektu znajduje się rozwój nowych technologii — algorytmów, wydajności, wykorzystania pamięci itd. — musisz założyć, że harmonogram będzie bardzo niestabilny. W przypadku gdy zajmujesz się rozwojem technologii, a Twój projekt zawiera inne słabe punkty — zbyt mały personel, niewykwalifikowany personel, niesprecyzowane wymagania, niestabilny interfejs tworzony przez zewnętrznego podwykonawcę — możesz od razu wyrzucić założony harmonogram. Jeśli musisz za wszelką cenę rozwinąć się technologicznie, zrób to, ale nie oczekuj, że szybko zakończysz projekt!

Technologia

Pozostałe błędy mają związek z nieużywaniem lub nieprawidłowym stosowaniem współczesnej technologii.

Odnośniki: Więcej informacji na temat syndromu „srebrnej kuli" znajdziesz w podrozdziale 15.5. „Syndrom »srebrnej kuli«".

33. Syndrom „srebrnej kuli". W przykładzie 3.1 za bardzo zaufano potencjalnym zaletom reklamowanych, lecz nieużywanych wcześniej technologii (generator raportów, programowanie obiektowe, język C++) i nie przeanalizowano ich działania w tym konkretnym środowisku projektowym. Jeśli zespoły projektowe zdobędą pojedyncze nowe rozwiązanie, nową technologię lub rygorystyczny proces i oczekują od nich lekarstwa na wszystkie problemy z harmonogramem, to srodze się zawiodą (Jones 1994).

Odnośniki: Więcej informacji dotyczących oszczędzania na narzędziach zwiększających produktywność znajdziesz w punkcie „Określanie stopnia redukcji harmonogramu" (podrozdział 15.4).

34. Przeszacowanie oszczędności wynikających ze stosowania nowych rozwiązań lub narzędzi. Firmy sporadycznie zwiększają produktywność w znaczącym stopniu, bez względu na ilość bądź jakość wdrażanych nowych rozwiązań i narzędzi. Zalety korzystania z nowych rozwiązań są częściowo równoważone przez czas potrzebny na nauczenie się ich, a doskonałe opanowanie danej techniki wymaga wiele czasu. Z nowymi rozwiązaniami wiążą się również nowe zagrożenia, które ujawniają się dopiero podczas korzystania z danej metody. Spodziewaj się raczej powolnego, stabilnego zwiększania produktywności rzędu kilku procent na dany projekt, a nie olbrzymiego skoku. Zespół omawiany w przykładzie 3.1 powinien założyć zwiększenie produktywności najwyżej o 10% wynikające z nowej technologii, a nie jej podwojenie.

Odnośniki: Więcej informacji na temat wielokrotnego wykorzystywania zasobów znajdziesz w rozdziale 33. „Wielokrotne wykorzystywanie zasobów".

Specjalnym przypadkiem nadmiernych oszczędności jest ponowne wykorzystywanie kodu stworzonego w poprzednich projektach. Takie rozwiązanie może być bardzo skuteczne, ale oszczędność czasu rzadko bywa równie spektakularna.

35. Zmiana narzędzi w środku projektu. Jest to stare rozwiązanie ratunkowe, które bardzo rzadko daje zamierzony efekt. Czasem warto zaktualizować wykorzystywane oprogramowanie, np. z wersji 3.0 do 3.1, a nawet 4.0, jednak czas poświęcony na naukę, potrzebny na zmodyfikowanie dotychczasowego produktu, oraz nowe rodzaje pomyłek nieodmiennie towarzyszące kolejnej wersji narzędzia zazwyczaj niwelują wszelkie potencjalne oszczędności czasu w trakcie trwania projektu.

Odnośniki: Więcej informacji na temat kontrolowania kodu źródłowego znajdziesz w punkcie „Zarządzanie konfiguracją oprogramowania” (podrozdział 4.2).

36. Brak zautomatyzowanej kontroli kodu źródłowego. Pominięcie zautomatyzowanej kontroli kodu źródłowego naraża projekt na niepotrzebne ryzyko. Bez tej funkcji dwóch programistów pracujących nad tym samym modułem musi koordynować ręcznie swoje wysiłki. Mogą dojść do porozumienia i porównywać najnowsze wersje swoich plików przed umieszczeniem ich w katalogu głównym, ale zawsze może się zdarzyć, że ktoś nadpisze wyniki pracy drugiej osoby. Ludzie tworzą kod do przestarzałych interfejsów, a później muszą zmodyfikować całość, gdyż korzystali z niewłaściwej wersji interfejsu. Użytkownicy zgłaszają wady, których nie można przeanalizować, ponieważ nie da się skompilować używanej przez nich wersji aplikacji. Przeciętnie kod źródłowy jest modyfikowany o ok. 10% w ciągu miesiąca, a jego ręczna kontrola po prostu nie nadąża za zmianami (Jones 1994).

W tabeli 3.1 zawarłem pełną listę klasycznych błędów.

3.4. Ucieczka z „wyspy Gilligana”

Lista wszystkich klasycznych błędów zajęłaby znacznie więcej stron, ale wymieniłem te najpowszechniej występujące oraz najpoważniejsze. Profesor David Umphress z Seattle University zauważa, że przyglądanie się, jak różne firmy starają się unikać klasycznych błędów, przypomina oglądanie powtórek *Wyspy Gilligana*[2]. Na początku każdego odcinka Gilligan, Skipper lub Profesor opracowują nowy niedorzeczny plan ucieczki z wyspy. Najpierw wszystko zdaje się zmierzać we właściwym kierunku, ale wraz z rozwojem fabuły cała koncepcja zaczyna się rozsypywać i na końcu odcinka bohaterowie trafiają do punktu wyjścia — na wyspę.

Podobnie jest z większością firm: na końcu projektu odkrywają, że został popełniony kolejny klasyczny błąd, po którym nastąpił poślizg w terminie wydania produktu, został przekroczony budżet albo jednocześnie wystąpiły obydwa skutki.

[2] Amerykański serial komediowy emitowany w latach 1964 – 1967 — *przyp. tłum.*

Tabela 3.1. Lista klasycznych błędów

Błędy związane z czynnikiem ludzkim	Błędy związane z procesem	Błędy związane z produktem	Błędy związane z technologią
Podkopana motywacja	Nadmiernie optymistyczne harmonogramy	Nadmiar wymagań	Syndrom „srebrnej kuli”
Słaby personel	Niewystarczające zarządzanie ryzykiem	Przerost funkcjonalności	Przeszacowanie oszczędności wynikających ze stosowania nowych rozwiązań lub narzędzi
Niekontrolowani, problematyczni pracownicy	Niewywiązanie się podwykonawcy ze zlecenia	Samowola programistów	
Brawura	Niedokładne planowanie	„Przepychane” negocjacje	Zmiana narzędzi w środku projektu
Dołączanie ludzi do przedłużającego się projektu	Zaprzestanie planowania pod wpływem presji	Projektowanie zorientowane na badania	Brak zautomatyzowane j kontroli kodu źródłowego
Hałaśliwe, zatłoczone biuro	Czas zmarnowany na przygotowaniach		
Napięcie pomiędzy programistami a klientami	Zbagatelizowanie początkowych etapów projektowania		
Wygórowane oczekiwania	Nieodpowiednie przygotowanie schematów		
Brak sponsora z prawdziwego zdarzenia	Zbagatelizowanie kontroli jakości		
Brak zaangażowania ze strony wszystkich zainteresowanych	Niewystarczające mechanizmy zarządzania		
Brak wpływu użytkowników	Przedwczesna lub zbyt często występująca konwergencja		
Przedkładanie polityki nad treść	Pomijanie podstawowych zadań		
Pobożne życzenia	Planowanie nadrabiania zaległości		
	Strategia ekstremalnego programowania		

Twoja własna lista najgorszych rozwiązań

Nigdy nie zapominaj o klasycznych błędach. Stwórz listę „najgorszych rozwiązań”, których będziesz unikać w kolejnych projektach. Rozpocznij od listy zawartej w tabeli 3.1. Dodawaj do niej kolejne błędy, czytając sprawozdania z ukończenia projektu, aby poznać pomyłki popełnione przez zespół. Przekonaj zarząd, aby wprowadził sporządzanie sprawozdań również w innych projektach, dzięki czemu będziesz w stanie poznać błędy popełniane przez inne zespoły. Wymieniaj się również opowieściami i doświadczeniami z pracownikami innych firm. Pokazuj sporządzoną przez siebie listę błędów jak największej grupie osób, gdyż ktoś może wyciągnie wnioski i nie popełni po raz kolejny klasycznego błędu.

Literatura uzupełniająca

Kilka książek zajmuje się zagadnieniem pomyłek programistycznym, ale nie spotkałem się z tytułem traktującym o klasycznych błędach w dziedzinie sporządzania harmonogramów projektu. W dalszej części tej publikacji poruszam szczegółowo pewne kwestie dotyczące klasycznych błędów.

Skorowidz

I

J

K

L

M

N

O

P

R

S

Ś

T

U

W

Z

Odzyskaj kontrolę nad swoim projektem i zrealizuj go w terminie!

Zespoły projektowe borykają się z ciągłym niedostatkiem czasu. Napięte do granic możliwości terminy wymuszają na software developerach narzucenie morderczego tempa pracy. Takie podejście sprawia, że albo dostarczony produkt nie spełnia oczekiwań, albo nie udaje się dotrzymać terminu. Co gorsza, ciągła praca pod presją czasu powoduje chroniczne przemęczenie i problemy zdrowotne, nie wspominając już o braku sił i czasu na rozwój, który w branży IT ma kolosalne znaczenie.

Książka ta jest praktycznym, zdroworozsądkowym poradnikiem metod projektowania. Opisane w niej strategie pracy pozwolą na usprawnienie i przyśpieszenie procesu projektowego. Przedstawiono tu również takie zagadnienia jak zarządzanie ryzykiem, podstawy projektowania aplikacji oraz planowanie cyklu życia projektu. Mimo że nie są bezpośrednio związane z metodami szybkiego projektowania, mają kluczowe znaczenie dla produktywności zespołu. Naturalnie, nie istnieje jedna magiczna metoda przydatna w każdych warunkach — w tej książce opisano i krytycznie przeanalizowano najprzydatniejsze rozwiązania z różnych branż tworzenia oprogramowania.

- strategie szybkiego projektowania i sprawdzone rozwiązania
- rozwiązania przyśpieszające realizację projektu, takie jak prototypowanie, języki szybkiego projektowania, zasady motywowania zespołu oraz zasady wydajnej współpracy
- najczęściej popełniane błędy, ich przyczyny i konsekwencje
- studia przypadków oparte na rzeczywistych wydarzeniach
- dobieranie właściwych metod do poszczególnych projektów

Steve McConnell jest głównym inżynierem oprogramowania i dyrektorem generalnym w spółce Construx Software Builders. Jest także członkiem organizacji IEEE Computer Society oraz ACM. Aktywny programista, koncentruje się głównie na projektowaniu komercyjnego oprogramowania „celofanowego" (ang. shrink-wrap). Współpracuje z wieloma znanymi firmami, w tym z korporacją Microsoft. Wraz z żoną i z dziećmi mieszka w Bellevue w stanie Waszyngton.

ISBN 978-83-283-3270-6
9 788328 332706
cena: 99,00 zł