

wymagań i dokumentacja implementacji z diagramami modułów. Trochę mniej programistów zdaje sobie sprawę z tego jak ważne jest, aby ta dokumentacja była napisana prostym i zrozumiałym językiem. Nie zmienia to faktu, że świadomość wagi dobrej dokumentacji jest powszechna.

Dużo mniej powszechna jest świadomość, co tak naprawdę oznacza dobre, porządne wykonanie danego dokumentu. I jakie są związane koszty ze zrobieniem tego naprawdę dobrze. Oraz jakie są skutki zrobienia tego byle jak.

Chcę tu zwrócić uwagę na kilka niebezpieczeństw.

Ostrożnie z diagramami

Doczesne miejsce w dokumentacji technicznej zajmują diagramy. Diagramy modułów w systemie jako całości, diagramy interakcji między obiektami, diagramy hierarchii klas, scenariusze itd.

W niektórych sytuacjach kilka czytelnych diagramów potrafi przekazać więcej niż kilkanaście stron tekstu. Dlatego, przy tworzeniu czytelnej i zrozumiałej dokumentacji, diagramy są niezastąpione. Istnieje co najmniej kilka programów ułatwiających tworzenie prostych i czytelnych diagramów (włączając w to narzędzia do rysowania dostępne w Wordzie). Warto wybrać sobie narzędzie, które najbardziej nam pasuje i go używać. Podkreślam – nic nie zastąpi czytelnego diagramu.

Przy podejmowaniu decyzji o tworzeniu diagramu powinniśmy kierować się zdrowym rozsądkiem. Przestrzegam przed postanowieniami typu „diagram dla każdego scenariusza”, „diagram dla każdego obiektu”. Przyjęcie takich zasad spowoduje, że będą tracone godziny lub dni na diagramy ilustrujące rzeczy na tyle proste, że można je opisać w kilku liniach tekstu. Diagram nie jest celem samym w sobie. Celem rysowania diagramu jest ilustracja rzeczy trudniejszych do opisanie słowami. Nowoczesne narzędzia wychodzą nam tu naprzeciw, automatyzując tworzenie niektórych diagramów na podstawie definicji obiektów. Jednak opierając stworzoną dokumentację na dużej ilości diagramów (a będzie ich dużo, jeśli będziemy je tworzyć, dla np. każdego obiektu) musimy być świadomi kosztów ich wykonania oraz kosztów dbania o to, by były aktualne. Możemy skorzystać z programu, który bardzo pomaga przy rysowaniu takich obiektów. Takie programy nie tylko same rysują „klocki” z odpowiednimi atrybutami. Dają one możliwości tworzenia repozytorium obiektów i odwoływania się do diagramów danych obiektów z różnych dokumentów. Mówiąc ogólnie - pomagają zarządzać zagadnieniami i obiektami, które występują w projekcie. Musimy mieć jednak na uwadze kilka czynników. Trzeba pamiętać, że repozytorium obiektów w dużym projekcie często się zmienia. W związku z tym, ktoś musi tym administrować. Musi też istnieć jakiś sposób określania, które repozytorium odpowiada danej wersji oprogramowania.

Wyobraźmy sobie następujący przykład. Mamy obiekt, który jest obecny na wielu diagramach. I ten obiekt się zmienia. Dochodzi nowa ważna metoda, bądź usuniętych jest 5 atrybutów. W związku z tym każdy dokument, który ma w sobie diagram z tym obiektem, musi zostać uaktualniony lub powinna zostać stworzona nowa wersja tego dokumentu. Już samo to zajmuje czas. A jak jeszcze są diagramy, które importują inne diagramy (np. za pomocą OLE) to uchowaj Boże. Używałem takiego narzędzia ze 4 lata temu. Zmiana głupiego obiektu zajmowała mi kilka dni, bo program działał wolno i się sypał. Ale 4 lata to dużo czasu. Dzisiaj takie rzeczy są bardziej stabilne. Jednakże narzut administracyjny pozostaje. I zależności między wersjami też. Pamiętajmy, że jeżeli „bawimy się” w repozytoria obiektów, to musimy się też „bawić” w administrowanie wersjami tych obiektów. W przeciwnym razie, będzie to bezwartościowe, bo każdy programista czy analityk, będzie się głowił, czy diagram, który widzi to jest akurat aktualna

wersja. Oczywiście, są obiekty, których ilustracje, ze względu na ważność tych obiektów, muszą pojawić się w wielu dokumentach. Jeżeli zmieni się obiekt, który jest bardzo ważny w architekturze systemu, to wtedy nie ma wyjścia – dokumentacja musi być uaktualniona. Jeżeli jednak w dokumentacji umieścimy szczegółowe diagramy obiektów poślednich, to skazujemy się na pracochłonne uaktualnianie prawie wszystkich dokumentów po każdym cyklu rozwoju oprogramowania.

Dokumentacja użytkownika

W bardzo wielu projektach dokumentację użytkownika piszą programiści. Nie piszą jej dlatego, że mają żylkę do znajdowania najbardziej prostych opisów złożonych zjawisk. Ani nie piszą jej dlatego, że rozkoszują się zabawą słowem. Najczęściej, programiści piszą dokumentację, bo muszą. Jeżeli oczarowuje nas jakość dokumentacji tworzonej przez renomowane firmy, to proszę pozwolić, że opiszę jak wygląda proces jej wytwarzania. Zajmuje się tym osobny dział, składający się z ludzi, którzy potrafią dobrze pisać (pisać teksty do czytania a nie oprogramowanie). Ponieważ takim ludziom zdarza się nie znać od środka zagadnienia, o którym piszą, robią tak: na podstawie wymagań piszą pierwszą wersję. Potem dają tę wersję do przejrzenia programistom i testerom, którzy opisują funkcjonalność implementowali i testowali. Nanoszą oni uwagi i dokument jest poprawiany. Jeżeli trzeba, cykl jest powtarzany. A pod sam koniec, dokumentacja jest testowana (tzn. próbuje się wykonywać poszczególne funkcje programu kierując się instrukcjami z dokumentacji). Sami musimy ocenić czy nasz projekt na to stać czy nie (dotyczy to zwłaszcza niedużych projektów tworzonych przez małe i średnie firmy).

3. Narzędzia

Po pierwsze narzędzia wspomagające proces testowania i zarządzania nim są drogie. Istnieją oczywiście darmowe narzędzia (Open Source) i wiele z nich ma bardzo dobrą reputację. Jednak wciąż pokutuje mit (moim zdaniem - fałszywy), że dla narzędzi Open Source nie ma wsparcia. Mit wzmocniony jeszcze propagandą firm komercyjnych (patrz FUD, <http://en.wikipedia.org/wiki/FUD>). Odstrasza to skutecznie wielu managerów.

Po drugie musimy mieć na uwadze, że wbrew temu, co mówią często materiały marketingowe, korzystanie z narzędzi komercyjnych wcale nie jest proste. Nie jest proste skonfigurowanie ich w sposób w pełni odpowiadający naszym potrzebom. Nie jest także proste nauczyć się nimi *efektywnie* posługiwać. Każdy projekt ma swoje niuansiki i przystosowanie narzędzia do tych niuansików wymaga dużej wiedzy i czasu (albo pieniędzy na konsultanta, który to zrobi, (jeżeli to możliwe) i wystawi rachunek nie za rozwiązanie, lecz za spędzony w projekcie czas).

Pamiętajmy, że znane pakiety dają nie tylko oprogramowanie narzędziowe. Dają one też metodologię, którą do jakiegoś stopnia będziemy musieli wdrożyć w projekcie. Uwzględnijmy to i dodajmy do kosztów licencji i wsparcia technicznego. Weźmy też pod uwagę, że wdrożenie nowego narzędzia w jednym departamencie, może za sobą pociągnąć konieczność wdrożenia tego narzędzia w departamentach współpracujących. Wtedy suma cen licencji zaczyna gwałtownie rosnąć.

Komercyjne narzędzia nęcą też bardzo ciekawymi funkcjami, które zaimplementowane w projekcie, mogą rzeczywiście wnieść nową jakość do procesu. Bądźmy jednak czujni. Często się okazuje, że skorzystanie z tej funkcjonalności wymaga podporządkowania się pewnym regułom, które mogą być trudne lub prawie niemożliwe do spełnienia (przynajmniej w naszym projekcie). Ale tak nie musi być zawsze. Jeżeli uznamy jakąś funkcjonalność za pociągającą, sprawdźmy, co dokładnie musimy zrobić, aby móc z niej skorzystać. Może akurat w przypadku naszego projektu

nie będzie trzeba zrobić wiele a zyski będą znaczące. Musimy to sami ocenić.

4. Zapewnianie jakości kodu

Poza dbaniem o dobrą dokumentację, jakość można też podnieść wdrażając procedury polepszające proces kodowania.

Chyba we wszystkich poważnych tekstach na temat tworzenia dobrego kodu jest jasno powiedziane, że najlepszą metodą jego weryfikacji, jest przejrzenie go przez innych. Nazywa się to „code walkthrough” lub „code review” (code walkthrough to analiza programu przez wykonanie go w myśli linia po linii). Bez wątpienia jest to metoda skuteczna, ale bardzo droga. Kod nie przejrzy się sam. Muszą na to poświęcić czas inni programiści. Żeby ich komentarze były coś warte, muszą mieć oni czas, aby "wgrzyźć" się w kod. W przeciwnym razie nie będą mieli do powiedzenia nic bardziej odkrywczego ponad to, że wcięcia im się nie podobają lub, że nazwa zmiennej jest myląca.

Są też narzędzia typu Bounds Checker, które pozwalają badać kod w trakcie uruchomienia. Sprawdzają na przykład czy nie ma jakichś dziwnych alokacji lub dealokacji pamięci. Z cenami takich narzędzi jest różnie. BoundsChecker CompuWare kosztuje 800\$. Nie mam doświadczenia z tym narzędziem, ale opinie, które znalazłem na jego temat są w przeważającej mierze pozytywne. Ja korzystałem z Rational Purify. Są też narzędzia Open Sourcowe – tu też nie mam doświadczenia. Z tego co wiem nowe wersje kompilatorów i środowisk deweloperskich mają wbudowane tego typu narzędzia. Ogólnie oceniam, że nie jest to zła rzecz. Uruchomienie tego nie zajmuje znowuż tak dużo czasu, a uzyskane informacje są cenne. Należy jednak pamiętać o tym, że kod uruchamiany w takim trybie potrzebuje więcej zasobów komputera. Te jednak są tanie, więc warto się tym tematem zainteresować.

5. Planujmy uwzględniając nasze realne możliwości

Zanim zaplanujemy co będzie robione w projekcie, aby polepszyć jakość, zastanówmy się, na co nas stać. Weźmy pod uwagę budżet, ilość ludzi i terminy. Musimy też sobie uczciwie odpowiedzieć na pytanie czy implementujemy procedury zapewnienia jakości, bo chcemy być kryci przed zwierzchnikami lub czy dlatego że chcemy zwiększyć prawdopodobieństwo produkcji lepszego kodu. Jeżeli tylko chcemy być kryci, to właściwie nie widzę jakiś konkretnych barier dla naszej kreatywności. Możemy wymyślać jakie tylko chcemy procedury. Programiści zapewnią, że to wszystko będzie działać. Będą tworzone na czas odpowiednie dokumenty, rysowane diagramiki, dostarczane będą rezultaty testów. To nic, że dokumentacja będzie bezwartościowa i niezrozumiała. To nic, że diagramy będą nieaktualne. To nic, że jak się przyjrzymy testom, to stwierdzimy, że są napisane tak, żeby zawsze przechodziły. I to nic, że oprogramowanie nie będzie działać. Przecież to nie o to w tym chodzi. Chodzi o możliwość wykazania, że trzymaliśmy się procedur. A programiści chętnie pomogą, bo zawsze znajdą sposób, żeby spełnić wymogi procedury. Każdej.

Pamiętajmy: W obliczu deadline-u, wszystkie procedury mogą zostać ominięte i oszukane. Programista musi być tylko wystarczająco inteligentny i chcieć. A chcieć znaczy móc. A w obliczy perspektywy pracy w wariackich nadgodzinach, chęć pojawi się sama.

Jeżeli jednak chcemy zrobić coś żeby produkt był lepszy to starajmy się unikać tworzenia zasad, które raczej na pewno będą łamane, bo będą nierealistyczne. Prawo, które musi być łamane jest demoralizujące. Starajmy się utworzyć zbiór zasad, które będą możliwe do spełnienia. Stawiamy realistyczne wymagania i wtedy egzekwujemy je z całą bezwzględnością.

6. Sugerowane minimum

Oto moje realistyczne minimum, będące wynikiem moich obserwacji w trakcie 8 lat pracy przy wytwarzaniu oprogramowania:

Dobrze spisane wymagania

Z biznesowego punktu widzenia, jest to jeden z najważniejszych dokumentów. To on opisuje, jaką funkcjonalność ma program zawierać. Powyżej przekonywałem, że niektóre rzeczy można sobie darować, bo są za drogie i nie wnoszą aż tyle, aby warto było ponosić koszty z nimi związane. Ale dobrze spisanych wymagań nie możemy sobie darować. I jeżeli zamierzamy oszczędzać na tworzeniu dokumentacji, to na tworzeniu wymagań powinniśmy starać się oszczędzić jak najmniej. Dobrze spisane wymagania mogą nas obronić przed żądaniem klienta, który twierdzi, że zrozumiał, że funkcjonalności będzie dwa razy więcej od tej, którą mu dostarczono. Źle spisane - są w statystykach źródłem największych strat finansowych w informatyce. Pisanie dobrych wymagań to bezcenna umiejętność i wielka sztuka, którą się zgłębia przez całe życie.

Komentarze w kodzie

Każda funkcja w kodzie powinna zawierać krótki opis co dana funkcja robi. Każdy moduł powinien mieć nagłówek, z krótkim opisem co dany moduł robi. (Wiem, wiem - to oczywiste. Wciąż jednak ta reguła nie jest przestrzegana, nawet w firmach przez duże F).

Zrozumiała, krótka dokumentacja techniczna

Do każdego większego modułu powinien istnieć odpowiedni, co najwyżej kilkustronicowy dokument. Powinien on przedstawiać - możliwie najprostszym językiem i używając klarownych ilustracji - funkcjonalność modułu. Stylem powinien bardziej przypominać opowiadanie niż specyfikację techniczną.

Celem tego dokumentu jest danie osobie czytającej wyobrażenia do czego moduł tak naprawdę służy i na jakiej zasadzie działa. Jeżeli dokumentację czyta osoba nietechniczna, to ten poziom szczegółowości jest z zasady wystarczający. Jeżeli czyta ją programista i potrzebuje więcej szczegółów, może ich poszukać w kodzie źródłowym modułu. Jeżeli kod będzie miał te minimum komentarzy, o których mówię powyżej i programista będzie znał przeznaczenie i ogólne zasady działania modułu – wtedy poruszanie się po kodzie będzie dużo łatwiejsze.

System kontroli wersji.

Istnieją co najmniej 2 darmowe, sprawdzone przez rzesze informatyków, narzędzia do kontroli wersji – RCS i CVS. Nie znam żadnego racjonalnego powodu, żeby systemowi kontroli wersji nie stosować. Znam za to szereg kataklizmów, jakie mogą być efektem nieużywania takich systemów. Stosujmy kontrolę wersji zarówno do kodu programu jak i do dokumentacji.

Tylko po wdrożeniu tego minimum możemy zacząć myśleć o czymś bardziej zaawansowanym jak na przykład tworzenie szczegółowej dokumentacji technicznej czy też wdrożenie systemu zarządzania testami, automatyzacji testów etc.

Jeżeli nie mamy zasobów do wdrożenia tego minimum to nasz projekt można porównać do spaceru po linie na przepaści. Widzę 3 wytłumaczenia takiej sytuacji:

- Znaczny rozrost wymagań w trakcie projektu (scope creep)
- Ktoś planujący budżet projektu nie za bardzo się na planowaniu znał
- Ktoś planujący budżet projektu znał się na tym, ale planowane zyski ze zrealizowania

projektu są tak wysokie, że zaakceptowano tak wysokie ryzyko. Warto mieć świadomość, że brak tego minimum jakości, czyni projekt misją wysoce ryzykowną wtedy, gdy w projekcie pracuje 4-5 osób. Dla projektów 10 i więcej osobowych jest to już po prostu misją samobójczą.

Na koniec jeszcze drobna sugestia:

Preferujemy komunikację ustną nad pisemną przy przekazywaniu wiedzy

Nie utrudniamy testerom dostępu do programistów. Żaden dokument nie zastąpi rozmowy ze specjalistą/autorem modułu, który odpowie nam na nasze pytania. Jeżeli jest dostępna zarówno dokumentacja jak i programista modułu, nie skazujemy testera na czytanie dokumentacji pisanej przez programistów, którzy traktują dokumentację jako dopust boży. Pozwólmy testerowi porozmawiać z programistą.

7. Życie nie jest takie proste...

Dla większości uogólniających tez i rad istnieją kontrprzykłady i dziedziny gdzie te tezy są fałszywe a rady się nie stosują. Mój artykuł nie jest tu wyjątkiem. I to jest nieuniknione. Jak już zaznaczyłem, zagadnienia zapewnienia jakości są zbyt złożone, żeby je podsumować na kilku stronach. Zdaję sobie sprawę, że to co piszę może nie przystawać (przynajmniej wprost) do rzeczywistości niektórych projektów.

Niestety, nie tylko my (wytwórcy oprogramowania) decydujemy o tym, jaka ma być dokumentacja. Czasami klient narzuca nam, co ma być tworzone. Warto mieć przynajmniej świadomość, jakie koszty pociąga za sobą tworzenie dokumentacji. Może jak się to uświadomi klientowi, to zrezygnuje on z części dokumentów.

Niektórzy są na wyrafinowane procedury zapewnienia jakości skazani z definicji - systemy sterowania elektrownią jądrową, systemy wojskowe, NASA (której próby oszczędzania nie wyszły na dobre). Dla tych firm dokumentacja jest nie tylko informacją techniczną. Jest też dowodem przy wystąpieniu jakichkolwiek problemów. A koszty ewentualnej awarii są tak wysokie, że wielokrotnie przewyższają ogromne nakłady na zapewnienie jakości.

Pomimo to wciąż jest wiele projektów, gdzie takich odgórnym wymogów nie ma. I wtedy warto mieć świadomość, jakie dana procedura zapewniania jakości niesie ze sobą narzuty. Czasami są one tak duże, że lepiej ją sobie darować, bo jej wdrożenie zje czas i budżet naszego projektu.