Testowo Zorientowane Metodyki Wytwarzania Oprogramowania

Marcin Baliński

21 grudnia 2010

Spis treści

Rozdział 1

Wstęp

1.1 Problemy napotykane w procesie wytwarzania oprogramowania

Na każdy projekt informatyczny można patrzeć z różnych perspektyw, w tej pracy chciałby jednak skupić się na technicznym aspekcie, jakim jest faza jego rozwoju w wybranym języku programowania, zaczynając od opisania częstych problemów podczas tej fazy.

Rozwój oprogramowania nie jest rzeczą trywialną, po dogłębnej analizie potrzeb, wybraniu narzędzi, które posłużą do budowy systemu następuje faza implementacji. Jej trudność zależy od wielu czynników takich jak:

Rodzaj wybranych narzędzi Czy wybrane środki techniczne takie jak język programowania lub zestaw zewnętrznych bibliotek nadają się do rozwiązania tego typu problemu?

Stopień skomplikowania systemu

Stopień integracji systemu Czy łatwo oddzielić od siebie poszczególne wewnętrzne funkcje systemu, czy konieczna jest integracja z zewnętrznym oprogramowaniem?

Wielkość zespołu programistów

Stopień technicznej świadomości ludzi odpowiedzialnych za prowadzenie projektu Wpływa na jakość komunikacji z programistami

Powyższe czynniki wpływają bezpośrednio na konkretne problemy, jakie pojawiają się podczas implementacji systemu. Rozwojowi oprogramowania najczęściej towarzyszą problemy takie jak:

Wzrost stopnia skomplikowania bazy kodu Kod staje się coraz bardziej skomplikowany i trudniejszy w utrzymaniu, wynika to często z braku ustalonych konwencji, polityki włączania do projektu zewnętrznych rozwiązań lub słabej komunikacji w zespole programistów.

Niepotrzebny wzrost stopnia integracji Łamanie zasady modułowego tworzenia oprogramowania, poszczególne części systemu są ze sobą coraz bardziej związane i znacząco na siebie wpływają, powoduje to sytuację, w której usterka w jednym module powoduję awarię w kilku innych częściach systemu.

Trudność w utrzymaniu systemu zgodnie z dostarczoną specyfikacją Wynikająca z niewłaściwej komunikacji lub z wcześniejszych błędów.

Te i wiele innych problemów sprawiają, że koniecznym staję się wprowadzenie narzędzia kontroli, dzięki któremu można by upewnić się co do jakości dostarczonych rozwiązań a także zminimalizować ryzyko pojawienia się podobnych problemów w przyszłości. Jednym z takich narzędzi są testy oprogramowania.

1.2 Testowanie oprogramowania

1.2.1 Wady manualnego testowania

Testowanie oprogramowania może odbywać się w sposób manualny lub automatyczny. Testy manualne przeprowadzane są przez żywego testera, który korzystając z oprogramowania, krok po kroku sprawdza jego zgodność ze specyfikacją, następnie wskazuje i opisuje ewentualne braki lub błędy. Każda nowa funkcjonalność lub poprawka wprowadzona do oprogramowania wymaga osobnej sesji z udziałem testera.

Podejście manualne ma wiele wad, wśród których do najważniejszych należą:

- Konieczność dogłębnego zrozumienia założeń projektu przez osobę odpowiedzialną za testowanie
- Trudność związana z koniecznością zidentyfikowania i przetestowania jak największej liczby możliwych przypadków użycia oprogramowania
- Czasochłonność: każda nowa funkcjonalność lub poprawka wymaga osobnej sesji testowania
- Wysokie koszty pracy testera

- Ogromna trudność zastosowania w wysoce specjalistycznych projektach
- Brak możliwości dokładnego przetestowania szczegółów implementacji danej funkcjonalności

Waga powyższych niedogodności rośnie wykładniczo wraz ze wzrostem poziomu skomplikowania oprogramowania, dlatego też manualne testowanie sprawdza się w zasadzie tylko w projektach o małej złożoności, w innych przypadkach istnieje potrzeba uzupełnienia lub zastąpienia go przez testy automatyczne.

1.2.2 Testy automatyczne

Automatyzacja procesu testowania odbywa się poprzez zastąpienie testera oprogramowaniem, które przejmie jego rolę. Automatyczne metody testowania umożliwiają sprawdzenie działania kodu programu, jak również graficznego interfejsu użytkownika.

Testowanie kodu

W procesie tym testujemy szczegóły implementacji systemu. Oprócz kodu potrzebnego do zrealizowania danej funkcjonalności programiści piszą również testy weryfikujące jej implementację. Testy takie mogą mieć różne funkcje, wymienię tylko niektóre z nich:

Testy jednostkowe Sprawdzają pojedynczy, niepodzielny element implementacji taki jak metoda lub funkcja

Testu integracyjne Sprawdzają interakcję między składowymi elementami systemu

Celem testu może być wynik działania danej części kodu, może być nim również chęć upewnienia się, że wynik działania osiągnięty jest w konkretny sposób. Dla przykładu testując funkcję, której zadaniem jest wyświetlić na ekranie monitora napis "Witaj Świecie!", możliwe jest, że prócz samego faktu pojawienia się treści na ekranie, chcemy również upewnić się, że do jej wyświetlenia użyta została jakaś konkretna metoda pochodząca z biblioteki standardowej. Jest to duża przewaga w stosunku to manualnego testowania oprogramowania, które nie daje nam takiej możliwości kontrolowania procesów prowadzących do widzialnych rezultatów.

W chwili obecnej istnieją dziesiątki gotowych narzędzi pozwalających testować kod napisany w każdym szerzej używanym języku programowania. Ich używanie należy do podstaw każdej nowoczesnej metodyki prowadzenia projektów informatycznych.

Testowanie interfejsu użytkownika

Istnieje szereg narzędzi pozwalających testować zachowanie, oraz wygląd interfejsów użytkownika, ich działanie opiera się najczęściej na nagrywaniu i późniejszym odtwarzaniu testowanych interakcji oraz porównywaniu ich rezultatów z oczekiwanymi. W taki sposób można testować tradycyjne aplikacje, jak również aplikacje www, działające w przeglądarce[1]

W kwestii testowania interfejsu użytkownika przewaga automatycznych testów nie jest już tak druzgocąca jak w przypadku testowania kodu, jednak i tutaj jesteśmy w stanie znacząco skorzystać na automatyzacji, zyskiem jest przede wszystkim czas oraz zerowy koszt powtórzenia testu.

Decydując się na automatyzacje procesu testowania oprogramowania należy pamiętać, że nadal kluczowym elementem jest konieczność dogłębnego zrozumienia specyfikacji oprogramowania przez osobę odpowiedzialną za pisanie testów. Równie ważnym wymogiem jest to, że pakiet testów powinien być kompletny, to znaczy pokrywać wszystkie kluczowe elementy systemu. Im większy procent kodu pokryty jest testami, tym lepiej, oznacza to również, że każdy nowy kod musi być dostarczony wraz z odpowiednimi testami.

Jeśli spełnimy te warunki proces utrzymania oprogramowania stanie się dużo łatwiejszy, oto niektóre z korzyści:

- Mamy pewność, że system działa zgodnie z założeniami
- Proces modyfikacji oprogramowania staje się łatwiejszy i bezpieczniejszy: jeśli nowy kod spowoduje defekt w którejś z bieżących funkcjonalności zostaniemy o tym niezwłocznie poinformowani przez nie przechodzący test

[1] w tym przypadku szczegóły działania narzędzi testujących są inne, interfejs użytkownika jest bowiem najczęściej zdefiniowany przez znaczniki HTML.

Rozdział 2

Test Driven Development

2.1 Czym jest TDD?

Test Driven Development jest praktyką, według założeń której każda modyfikacja systemu poprzedzona jest stworzeniem odpowiedniego testu opisującego tą modyfikację. Programista zaczyna od napisania testu, który z naturalnych przyczyn (testowany kod nie istnieje a tym etapie) daje wynik negatywny. Następnie napisany zostaje właściwy kod, którego zachowanie zgodne jest z testowanym. Kiedy testy przechodzą można wprowadzić ewentualne poprawki. Proces rozwoju oprogramowania w zgodzie z filozofią TDD składa się z wielu takich cyklów, które zobrazować można diagramem:

[diagram cyklu TDD]

1. Napisz test
2. Uruchom testy, upewnij się, że nowe testy nie przechodzą
3. Napisz kod
4. Uruchom testy, upewnij się, że przechodzą
5. Jeśli jest to potrzebne, zmodyfikuj kod

2.1.1 Główne zasady TDD

Zacznij od testu Test powinien być napisany zanim zacznie się implementacja funkcjonalności. Takie podejście gwarantuje, że będziemy mieli pełen zestaw testów, opisujących każdą funkcję systemu. Inną zaletą jest konieczność dokładnego przemyślenia szczegółów implementacji, jeszcze przed jej rozpoczęciem.

Zaraz po napisaniu nowe testy powinny dawać negatywny wynik Daje to pewność, że testy faktycznie spełniają swoją funkcję oraz, że każda degradacja funkcjonalności będzie sygnalizowana nieprzechodzącym testem.

2.1.2 Budowa testu

TBD

2.1.3 Zalety TDD

TBD

- Wzrost produktywności
- Wzrost jakości kodu
- Minimalizacja liczby defektów
- Możliwość wczesnego wykrycia defektów
- Modularyzacja kodu jako pozytywny skutek uboczny

2.2 Narzędzia wspierające TDD dostępne dla języka Ruby

TBD

Rozdział 3

Behavior Driven Development

3.1 Czym jest BDD?

3.2 Narzędzia BDD dostępne dla języka Ruby

Rozdział 4

Studium przypadku: Dynamicznie generowany panel administracyjny

4.1 Założenia projektu

4.2 Proces implementacji

4.3 Wnioski