Inżynieria Programowania - Testowanie oprogramowania cz.2

Arkadiusz Chrobot

Katedra Systemów Informatycznych, Politechnika Świętokrzyska w Kielcach

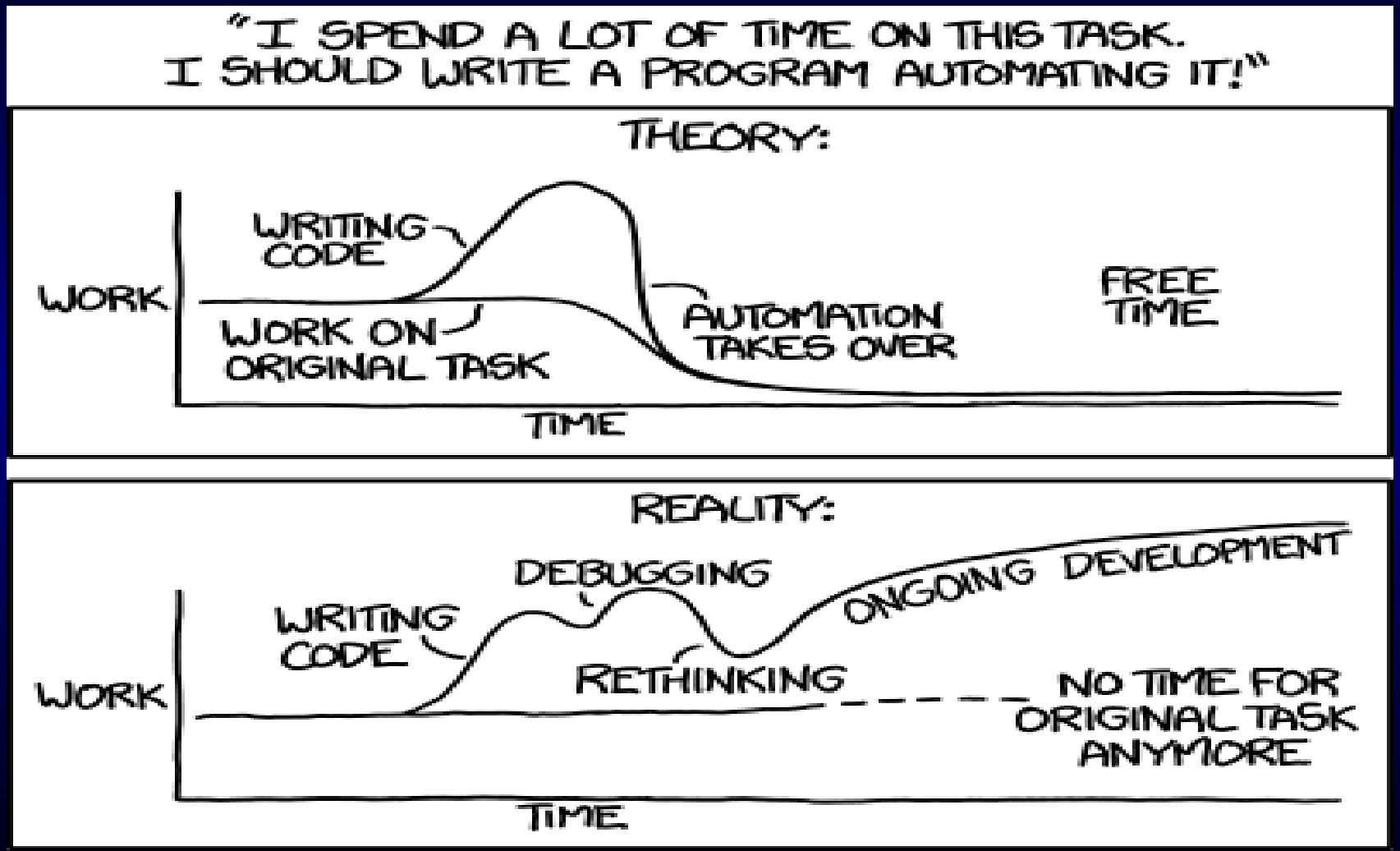
Kielce, 29 maja 2020

Plan wykładu

Motto

"Why program by hand in five days what you can spend five years of your life automating?"
Motto życiowe Terenca Parra

Motto 2



Źródło: http://xkcd.com/1319/

Wprowadzenie

Testowanie ma bezpośrednie przełożenie na jakość opracowywanego oprogramowania. Jest przy tym trudnym i obciążającym zasoby projektu procesem. Konieczne zatem stało się opracowanie pewnych narzędzi, które by usprawniły ten proces i pozwoliły określić jego efektywność.

Metryki

Testowanie jest czasochłonnym i kosztownym procesem. Opracowano więc pewne miary, które pozwalają ocenić jego stopień zaawansowania, oraz jakość przeprowadzonych testów. Te miary nazywane są metrykami. Istnieje kilka rodzajów metryk, które stosowane są z różnymi metodami testowania.

Metryki pokrycia kodu

Stosowane w strukturalnej metodzie testowania. Pozwalają ocenić kompletność testów, rozumianą jako ilość kodu testowanego oprogramowania, która objęta jest przypadkami testowymi. Dzięki nim można ustalić, czy całość kodu jest testowana oraz czy istnieją nadmiarowe przypadki testowe, które nic nie wnoszą do procesu testowania.

Metryka pokrycia bloków instrukcji

Blok kodu w tej metryce definiowany jest jako ciąg instrukcji sekwencyjnych, który nie zawiera instrukcji zmieniających przepływ sterowania (np. instrukcji warunkowych). Metryka ta pozwala ocenić stosunek liczby bloków przetestowanych do liczby wszystkich bloków. Podobną do niej jest metryka pokrycia instrukcji kodu - zamiast bloku brane są w niej pod uwagę pojedyncze instrukcje. Do ich zalet zalicza się zrozumiałość, prosta metoda pomiaru oraz możliwość zastosowania zarówno dla kodu źródłowego, jak i wynikowego. Wadą są problemy z określeniem jakości testów sprawdzających wyrażenie warunkowe w instrukcjach sterujących.

Metryka pokrycia bloków instrukcji - przykład

```
if(a==3) {
        funkcja1(...);
} else {
        funkcja2(...);
}
```

Ten fragment kodu do pełnego pokrycia wymaga co najmniej dwóch przypadków testowych - po jednym dla każdej gałęzi instrukcji warunkowej.

Metryka pokrycia bloków instrukcji - przykład

```
a=-2;
if(b>0)
        a=1;
x=sqrt(a);
```

Ten fragment kodu do pełnego pokrycia wymaga tylko jednego przypadku testowego. Jednakże jeśli ten przypadek zostanie źle dobrany to nie wykryje problemów z obliczaniem pierwiastka z liczby ujemnej.

Metryka pokrycia decyzji

Metryka ta pozwala sprawdzić na ile dobrze przetestowano instrukcje zmieniające przepływ sterowania. Jej wartością jest stosunek przetestowanych wyjść instrukcji sterującej do wszystkich jej wyjść. Do zalet tej metryki należy równie dobre sprawdzenie bloków jak to ma miejsce w przypadku poprzedniej metryki i lepsze sprawdzenie warunków w instrukcjach decyzyjnych. Do wad zaliczana jest konieczność opracowania większej liczby przypadków testowych oraz nieuwzględnianie specyfiki testowania warunków w językach programowania stosujących tzw. skracanie obliczeń.

Metryka pokrycia decyzji - przykład

```
if(a>0&&(b<0 || funkcja(a,3)>0)) {
        instr1;
} else {
        instr2;
}
```

Zgodnie z metryką decyzji ten fragment kodu może być w pełni przetestowany przy użyciu tylko dwóch przypadków testowych. Niestety nie muszą one gwarantować, że zostanie uruchomiona funkcja, której wywołanie umieszczono w warunku.

Metryka pokrycia ścieżek

Temat tej metryki był częściowo poruszany na poprzednim wykładzie. Jej zaletą w porównaniu z poprzednio opisanymi metrykami jest uwzględnienie skracania obliczeń. Za wady należy uznać konieczność opracowania dużej liczby przypadków, szczególnie jeśli testowany program zawiera pętle. W takich przypadkach testuje się oprogramowanie tylko dla ograniczonej iteracji tych pętli.

Metryki pokrycia wymagań

Te metryki stosowane są w testowaniu z użyciem metody funkcjonalnej. Ich celem jest ocena stopnia sprawdzenia wymagań opisanych w specyfikacji testowanego oprogramowania. Podstawowa metryka tego typu wyznacza stosunek liczby przetestowanych wymagań do liczby wszystkich wymagań. Definicja wymagania zależna jest od użytego sposobu określania wymagań. Liczba przypadków testowych niezbędnych do przetestowania pojedynczego wymagania zależna jest od jego złożoności.

Metryka pokrycia błędów

Wartością tej metryki jest stosunek liczby błędów, które wykryło oprogramowanie podczas testów, do liczby błędów na które powinno reagować według specyfikacji. Przez błąd należy tu rozumieć sytuacje wyjątkowe, jak błędy format danych wejściowych, niemożność nawiązania połączenia z serwerem, czy awaria urządzenia peryferyjnego.

Metryki pokrycia przypadków użycia

Przypadek użycia jest zbiorem scenariuszy określających przebieg typowej interakcji z systemem. Wyróżnia się biznesowe przypadki użycia, które związane są z wysokopoziomowymi wymaganiami względem systemu oraz przypadki systemowe, które opisują procesy w systemie z uwzględniemiem szczegółów technicznych. Można zatem określić metryki, które wyznaczą jakość pokrycia testami obu rodzajów przypadków użycia oraz wszystkich związanych z nimi scenariuszy.

Inne metryki

Sprawny przebieg testów zależy nie tylko od ich starannego zaprojektowania, ale także od jakości kodu, który nim podlega. Opracowano metryki, które pozwalają tę jakość określić, a także zbadać efektywność testowania i usuwania defektów.

Liczba wykrytych defektów

Ta metryka pozwala ocenić początkową jakość wytworzonego kodu, a następnie jej przyrost po fazach testowania i usuwania defektów oraz efektywność przyjętych metod testowania. Wymaga ona rejestrowania zmian liczby wykrywanych defektów w miarę upływu czasu realizacji projektu. Stabilizacja tej liczby jest sygnałem do zakończenia testów.

Liczba defektów komponentów

Ta metryka pozwala ocenić, które komponenty tworzonego oprogramowania posiadają największą liczbę defektów, a co za tym idzie mogą mieć słabej jakości kod, lub być bardzo złożonymi elementami. Innym powodem dużej liczby defektów może być przeciążenie pracą zespołu przygotowującego komponent. Wartością tej metryki jest całkowita liczba defektów liczona dla każdego komponentu z osobna. Ta metryka, pozwala także określić, które testy mają największą skuteczność wykrywania defektów. Mogą one być stosowane na etapie konserwacji systemu, kiedy trzeba sprawdzić jego działanie, po wprowadzonych modyfikacjach.

Częstość defektów

Wartością tej metryki jest liczba wykrytych defektów do liczby wierszy kodu. Pozwala ona ocenić przyrost jakości oprogramowania w stosunku do przyrostu liczby jego wierszy. Mimo niejednoznacznej definicji wiersza kodu jest ona dosyć powszechnie stosowaną metryką.

Procent defektów poprawionych

Jest to metryka, której wartością jest stosunek liczby wykrytych defektów do szacowanej całkowitej liczby defektów. Szacowanie to wykonuje się poprzez *zasiewanie defektów*, czyli celowe wprowadzanie sztucznych defektów. Całkowitą liczbę błędów wyznacza się ze wzoru $N_p = N_s \cdot \frac{n_p}{n_s}$, gdzie N_s jest całkowitą liczbą zasianych defektów, n_s jest liczbą zasianych defektów wykrytych podczas testów, a n_p jest liczbą faktycznych defektów wykrytych podczas testów. Im większa zgodność typów zasianych defektów i typów rzeczywistych defektów, tym większa jest wiarygodność tej miary.

Automatyzacja testów

Automatyzacja testów ma na celu usprawnienie tego procesu. Może ona dotyczyć różnych czynności i różnych etapów testowania. Tworzone są całe warsztaty testowe, które wspomagają pracę zespołów odpowiedzialnych za testowanie integracyjne. Istnieje także oprogramowanie, które ułatwia przeprowadzanie testów jednostkowych.

Automatyzacja testów jednostkowych - motywacja

Poprawnie wykonane testy jednostkowe pozwalają skrócić trwanie i zmniejszyć koszty kolejnych etapów testowania. W metodykach zwinnych stosowane jest podejście do tworzenia oprogramowania, nazywane Tworzeniem Sterowanym Testami (ang. Test Driven Development), w którym testy jednostek odgrywają kluczową rolę. Oprogramowanie wspomagające testowanie jednostkowe pozwala zmniejszyć nakład pracy związany z przygotowywaniem testów oraz ułatwia testowanie regresyjne.

JUnit 4 - wprowadzenie

Jednym z najczęściej stosowanych narzędzi do automatyzacji testów jednostkowych są biblioteki dla różnych języków programowania znane pod wspólną nazwą xUnit. Pierwszą wśród nich była biblioteka SUnit dla języka SmallTalk. Największą popularność zdobyła wersja tej biblioteki dla języka Java, czyli JUnit. Jednymi z jej głównych autorów są Kent Beck i Erich Gamma. Najnowszą wersją tej biblioteki jest wersja piąta. Wykład dotyczy wersji czwartej. Oprócz niej istnieją także: CUnit, dla języka C, NUnit (dla środowiska .NET), PyUnit dla języka Python i wiele innych.

JUnit 4 - implementacja testów

Biblioteka JUnit pozwala na odseparowanie kodu służącego do testowania od kodu testowanego. Testy są implementowane przy użyciu osobnej klasy, której nazwa zwyczajowo jest tworzona poprzez dodanie do nazwy klasy testowanej słowa `Test`. Wewnątrz tej klasy należy utworzyć i oznaczyć odpowiednimi adnotacjami metody przy pomocy których przeprowadzane są pojedyncze lub zbiorcze testy. Każda z tych metod korzysta z lub tworzy obiekt testowanej klasy, wywołuje jego metody i sprawdza wytworzone przez nie wyniki za pomocą jednej z metod `assert...()` zdefiniowanych w bibliotece JUnit.

JUnit 4 - metody assert...()

Metoda	Opis
`fail(`*message*`)`	Metoda powoduje zawsze pozytywny wynik testu.
`assertTrue(`*message*`, condition)`	Metoda daje negatywny wynik testu, jeśli warunek jest spełniony.
`assertFalse(`*message*`, condition)`	Metoda daje negatywny wynik testu, jeśli warunek nie jest spełniony.
`assertEquals(`*message*`,expected,actual)`	Metoda daje negatywny wynik testu, jeśli `expected` i `actual` są sobie równe. **Uwaga:** W przypadku tablic porównuje referencje!
`assertEquals(`*message*`, expected,actual,tolerance)`	Jak wyżej, ale porównuje wartości z dokładnością zadaną parametrem `tolarance` - używana do testowania wartości zmiennoprzecinkowych.
`assertNull(`*message*`, object)`	Metoda daje negatywny wynik testu, jeśli referencja ma wartość `null`.
`assertNotNull(`*message*`,object)`	Metoda daje negatywny wynik testu, jeśli referencja ma wartość różną od `null`.
`assertSame(`*message*`,expected,actual)`	Metoda daje negatywny wynik testu, jeśli obie referencje mają tą samą wartość.
`assertNotSame(`*message*`,expected,actual)`	Metoda daje negatywny wynik testu, jeśli referencje nie mają takiej samej wartości.

JUnit 4 - adnotacje

Adnotacja	Opis
@Test public void method()	Metoda jest metodą testującą.
@Test(expected=Exception.class) public void method()	Daje wynik pozytywny testu, jeśli metoda nie wyrzuci określonego wyjątku.
@Test(timeout=200) public void method()	Daje wynik pozytywny testu, jeśli test trwa więcej niż 200 milisekund.
@Before public void method()	Metoda jest wykonywana przed każdym testem.
@After public void method()	Metoda jest wykonywana po każdym teście.
@BeforeClass public static void method()	Metoda jest wykonywana jednokrotnie, przed wszystkimi testami.
@AfterClass public static void method()	Metoda jest wykonywana jednokrotnie, po wszystkich testach.
@Ignore public void method()	Metoda jest pomijana w trakcie testów.

JUnit 4 - parametryzacja testów

Aby uniknąć tworzenia osobnej metody dla każdego z testów osobno można skonstruować specjalną klasę testującą oznaczoną adnotacją @RunWith(Parameterized.class). Taka klasa może zawierać tylko jedną metodę testującą, która będzie wywoływana z różnymi parametrami będącymi danymi testowymi. Zbioru tych danych będzie jej dostarczała inna metoda klasy testującej. Ta metoda musi być metodą statyczną oznaczoną przez adnotację @Parameters i zwracać dane testowe w postaci kolekcji tablic. Dodatkowo klasa testująca wymaga konstruktora, który będzie odpowiedzialny za inicjację atrybutów przechowujących dane testowe.

JUnit 4 - zbiór testów

Duże projekty informatyczne składają się z dużej liczby klas. Przeprowadzeni zbiorczych testów dla wszystkich tych klas wymagałoby osobnego uruchamiania metod klas testowych. Biblioteka JUnit oferuje rozwiązanie tego problemu w postaci specjalnej klasy, która implementuje zbiór testów *test suite*. Taka klasa jest oznaczana adnotacjami @RunWith(Suite.class) oraz @SuiteClasses() do której w parametrze, przekazywany jest wykaz wszystkich klas testowych, które mają w tym zestawie być uruchomione.

Automatyzacja testów GUI - motywacja

Jeśli testowane oprogramowanie wyposażone jest w tekstowy interfejs użytkownika, to narzędzie automatyzujące testy komunikuje się z nim za pomocą przekierowania strumienia wejściowego i strumienia wyjściowego danych. Graficzne interfejsy użytkownika (GUI) stanowią większe wyzwanie. Testy takie mogą być przeprowadzone manualnie, ale zwiększa to ich koszty oraz nie gwarantuje powtarzalności. Oprogramowanie automatyzujące taki proces powinno mieć możliwość generowania zdarzeń wejściowych i rejestrowania reakcji testowanego programu.

Automatyzacja testów GUI - rozwiązania

Istnieją dwa typy rozwiązań programowych w zakresie automatyzacji testowania GUI.
Pierwszy to osobne oprogramowanie, które pozwala nagrać (zapisać) pojedynczą sesję testową przeprowadzoną ręcznie przez użytkownika, a następnie wielokrotnie ją odtwarzać. Przykładami takich rozwiązań są takie programy jak Marathon, który pozwala nagrać, odtworzyć, a także zmodyfikować zapis sesji testowania.
Drugim rodzajem rozwiązania są specjalne biblioteki operacji graficznych, które pozwalają zaprogramować sesje testowe programów wyposażonych w GUI. Przykładami takich bibliotek są Jemmy i UISpec4j.

Biblioteka Jemmy 2

Biblioteka Jemmy 2 służy do tworzenia oprogramowania, które symuluje pracę użytkownika z interfejsem graficznym aplikacji. Ponieważ sama nie posiada mechanizmów pozwalających zaimplementować testowanie, to często stosowana jest w połączeniu z biblioteką JUnit. Podobnie jak ona pozwala także oddzielić kod testujący od kodu testowanego.

Biblioteka Jemmy 2 - użycie

Klasa imitująca zachowanie użytkownika aplikacji musi implementować metodę `runIt()` z interfejsu `Scenario` wchodzącego w skład biblioteki Jemmy. W tej metodzie, należy uruchomić testowaną aplikację za pomocą metody `startApplication()` klasy `ClassReference`. Konstruktor tej klasy przyjmuje argument będący pełną nazwą (łącznie z pakietem) głównej klasy testowanej aplikacji. Następnie, za pomocą obiektu klasy `JFrameOperator` należy uzyskać referencję do obiektu głównej formatki aplikacji (klasa operatora identyfikuje ją po tytule) i za pomocą klas innych operatorów do poszczególnych elementów składowych GUI aplikacji. Każda z klas operatorów oferuje metody do aktywowania tych elementów zarówno w sposób blokujący, jak i nieblokujący. Przykładowo klasa `JButtonOperator` oferuje metody `push()` i `pushNoBlock()` służące do automatycznego „naciskania” przycisków.

Inne rodzaje testowania

Znaczna część współczesnych aplikacji, ma interfejs użytkownika będący stroną WWW. Testowanie tego typu interfejsów również podlega automatyzacji. Oprogramowanie, które przeprowadza takie testy korzysta z metod protokołu HTTP, aby imitować zachowanie użytkownika.
Testy obciążeniowe wykonywane są z użyciem oprogramowania, zwanego robotem, które generuje sztuczny ruch sieciowy dla testowanej aplikacji. Wyniki takich testów mogą być rejestrowane przez to oprogramowanie (np. czas odpowiedzi na żądania) lub przez osobne narzędzia (dzienniki zdarzeń systemu, monitor zużycia zasobów itd.). Testy obciążeniowe można połączyć z testami funkcjonalnymi. W takim wypadku robot generuje obciążenie, a osobne oprogramowanie testujące sprawdza, czy system poprawnie realizuje w takich warunkach określone dla niego zadania.
Do testowania akceptacyjnego aplikacji internetowych można użyć narzędzia o nazwie FitNesse.

Źródła

Materiał do wykładu powstał na bazie następujących źródeł:

- Krzysztof Sacha, „*Inżynieria oprogramowania*”, PWN, Warszawa, 2010
- Lars Vogel,“Unit Testing with JUnit - Tutorial”, http://www.vogella.com/tutorials/JUnit/article.html, dostęp:20.01.2014
- “Jemmy tutorial”, http://wiki.netbeans.org/Jemmy_Tutorial, dostęp:20.01.2014

Pytania

?

KONIEC

Dziękuję Państwu za uwagę.