Testowanie Oprogramowania.

C#

Wydział Matematyki i Informatyki Stosowanej
Ćwiczenia: Michał Mnich

Wstęp.

Jak istotnym jest testowanie oprogramowania może dowiedzieć się każdy programista który kiedykolwiek pisał program choć trochę większy niż 300 linijek. Już oprogramowania głupiej gry w ping ponga “https://pl.wikipedia.org/wiki/Pong” wymaga kilku testów badajacyh zachowanie piłki w przypadkach brzegowych. Wobec powyższego dobre oraz optymalne (nie nazbyt przesadne) testowanie aplikacji jest ważne. To nie ulega wątpliwości. Gdy tworzymy program i go wydajemy lub oddajemy dla klienta, to aplikacja powinna działać jak najlepiej. A jak sprawdzić, czy tak jest, jeśli nie jest testowana? Podejścia do testowania są różne. Niektóre firmy zatrudniają testerów manualnych, którzy przeklikują się przez aplikację. Inni piszą test automatyczne, w różnych formach. Jeszcze inni sprawdzają aplikację ręcznie tylko podczas pisania danej funkcjonalności. Które podejście jest właściwe? Tak naprawdę – wszystkie. Po prostu zależy to od danej sytuacji. Tworząc poważny system, powinniśmy go dokładnie testować. Jednak pisząc jedynie małą aplikację pomocniczą, której użyjemy tylko kilka razy i potem wyrzucimy, być może będzie to strata czasu.

Czemu testowanie jest ważne?

Odpowiedź jest prosta – żeby zapewnić wysoką jakość aplikacji. Ale testować można też ręcznie. A dlaczego powinniśmy pisać testy automatyczne (na których się tutaj skupiamy)? Korzyści jest kilka.

Dobrze napisane testy powinny przyspieszyć prace nad aplikacją. Pozwalają one szybciej (niż manualne testy) potwierdzić, że nowa funkcjonalność działa, a także że zmiany w istniejącym kodzie nie popsuły działających funkcjonalności.

Testy zmuszają nas także do pisania kodu w bardziej przejrzysty sposób. Aplikacja musi być stworzona bardziej modułowo, gdzie każda część zajmuje się jedną rzeczą – tak, żebyśmy dali radę to przetestować. Wielu programistów, którzy nie mieli do czynienia z testami, lubi wrzucać wiele różnych, niezależnych od siebie funkcji do jednej klasy czy metody. Przez to kod jest później ciężki do czytania, debugowania i trudno wprowadzać w nim zmiany. Testy automatyczne pozwalają nam więc na stworzenie kodu lepszej jakości. Kodu, z którym się wygodniej pracuje i jest mniej podatny na błędy.

Wprowadzenie nowego programisty do projektu również będzie łatwiejsze, gdy mamy do dyspozycji testy. Będzie mógł on dowiedzieć się z nich, jak powinny działać dane fragmenty kodu, bez zajmowania czasu innym osobom.

Rodzaje testów

Testy automatyczne możemy podzielić na kilka kategorii ze względu na to, jaką część systemu testujemy:

Testy jednostkowe – jest to rodzaj testów, w które służą do walidacji, jak i weryfikacji działania klas i ich metod a zatem sprawdzamy działanie tylko pojedynczego elementu aplikacji, który nie posiada żadnych zależności, czym na przykład jest metoda w klasie, która nie wykorzystuje niczego z zewnątrz.

Testy integracyjne – jest to poziom wyżej niż testy jednostkowe. Sprawdzamy tutaj elementy systemu, które od siebie zależą i czy powiązania między nimi działają prawidłowo.

Testy UI – są to testy interfejsu użytkownika. Jest to zupełnie oddzielna kategoria, ponieważ na warstwie widoku ciężko jest tworzyć testy w taki sam sposób jak w kategoriach powyżej. Tutaj zazwyczaj do dyspozycji mamy pewien framework, który pozwala nam na interakcję z włączoną aplikacją i podczas testu wykonuje zakodowane wcześniej akcje, podobnie jakby robił to zwykły użytkownik, a więc np. klika na przycisk, wpisuje tekst, itp.

Narzędzia

Do pisania testów przyda się jakiś framework. W świecie .Neta mamy 3 popularne opcje: **NUnit**, **xUnit**, **MSTest**

Na potrzeby tych ćwiczeń zajmiemy się Nunit-em ale nie bronię państwu użyć innych frameworków nawet nie figurujących na liście.

Ważny link:

http://getistqb.com/docs/sylabus-istqb-poziom-podstawowy/

Ćwiczenia:

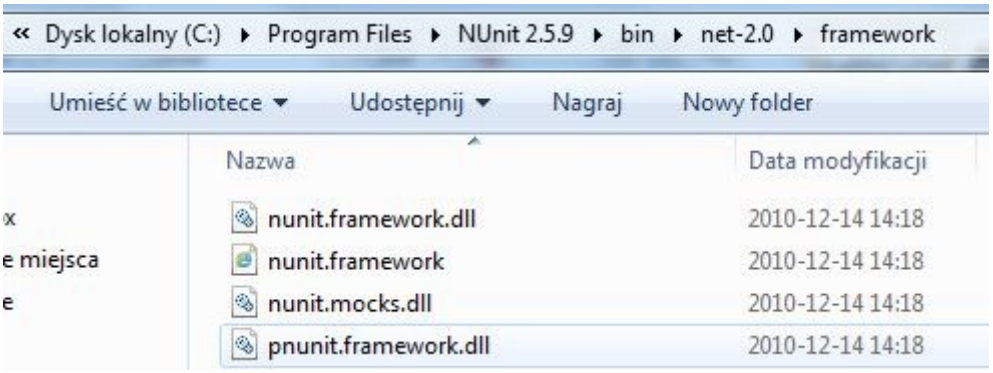
Na sam początek pobierz najnowszą wersję NUnit ze strony. http://www.nunit.org/

Proszę także wspomagać się przy tym ćwiczeniu manualem ze strony microsoftu.
https://docs.microsoft.com/pl-pl/visualstudio/test/getting-started-with-unit-testing?view=vs-2022&tabs=dotnet%2Cnunit

Załóżmy, że chcemy przetestować poniższą klasę:

```
public class SimpleCalculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

Wybierz z menu Visual Studio “File/Add New Project” i stwórz nowy “Class Project”. Do tego nowego projektu trzeba dodać referencje NUnit (nunity.framework.dll) . Poniższy obrazek pokazuje domyślną ścieżkę instalacyjną NUnit. Niech projekt nazywa się na przykład “**SimpleCalculator.Test**”



Do projektu trzeba też dodać referencje naszego programu, który ma być testowany w tym wypadku “**SimpleCalculator**”. Naturalnie by użyć klas wewnątrz tych bibliotek musimy dodać także klauzulę using z przestrzeniami nazw.

```
using NUnit.Framework;
using SimpleCalculator;
```

Po czym możemy już zacząć pisać test. Najprostszy możliwy test za pomocą biblioteki NUnit wyglądałby następująco:

```
[TestFixture]
public class MyUnitTest
{

[Test]
    public void TestRegularUseCaseWithSmallNumbers()
    {
        //Arrange:
        SimpleCalculator simpleCalculator = new SimpleCalculator();

        //Act:
        int result = simpleCalculator.adder(1,2);

      //Assert:
      Assert.AreEqual(3, result);
    }
}
```

Zadanie 1

Proszę napisać klasę łącząca dwa łańcuchy znaków. Jeżeli co najmniej jednym z łańcuchów znaków jest null funkcja ma również zwracać null. Następnie za pomocą biblioteki NUnit i mechanizmu asercji proszę napisać testy, które sprawdzą jej zachowanie dla kilku typowych wartości oraz warunków brzegowych. “Wartość brzegowa ISTQB”

Zadanie 2

Proszę napisać drugą wersję klasy łączącej dwa łańcuchy znaków. Jeżeli, któryś z argumentów jest nullem, chcielibyśmy, że metoda rzuciła wyjątkiem – proszę to sprawdzić przy pomocy asercji.

Zadanie 3

```
/**
* Interfejs obiektu który sprawdza czy dane słowa są anagramami. * Anagram
  jest słowem lub frazą, która powstała
* przez zmianę kolejności liter w oryginalnym słowie lub frazie.
* Zobacz kilka przykładów na http://www.wordsmith.org/anagram/hof.html
*/
public interface IAnagramChecker
{
     /*Sprawdza czy jedno slowo jest anagramem drugiego.
     * Wszystkie niealfanumeryczne znaki są ignorowane.
     * Wielkość liter nie ma znaczenia.
     * word1 - dowolny niepusty string różny od null.
     * word2 - dowolny niepusty string różny od null.
     * Zwraca true wtedy i tylko wtedy gdy word1 jest anagramem word2.
     */
     bool IsAnagram(string word1, string word2);
}
```

A. Proszę napisać jak najlepszy zestaw testów jednostkowych dla powyższego.
B. Proszę zaimplementować klasę **AnagramChecker.**
C. Jeżeli podczas implementacji przyjdą Państwu do głowy nowe przypadki testowe, to proszę je dopisać to zestawu testowego(proszę jakoś oznaczyć takie przypadki).
D. Proszę usunąć błędy i stworzyć dla nich nowe przypadki testowe(proszę je oznaczyć w jakiś specjalny sposób).

Zadanie 4

„Formularz zawiera pole "PESEL", system udziela zniżki osobom poniżej 18 i powyżej 65 lat."

Proszę w zadaniu korzystać z interface'u:

```
public interface IDiscountFromPeselComputer
{
    bool HasDiscount(String pesel)
}
```

oraz z klasy wyjątku:

```
public class InvalidPeselException : Exception
{
}
```

A. Proszę napisać za pomocą TDD testy jednostkowe dla interfejsu IDiscountFromPeselComputer
B. Proszę sprawdzić jaki odsetek Państwa testów nie przechodzi dla przykładowej implementacji umieszczonej na Pegazie.
C. Proszę wykorzystując stworzone przez Państwa testy jednostkowe napisać jak najlepszą implementację metody "HasDiscount".