Notatki do 4. tygodnia kursu Dotneta

Maciek Mielczarek

25 sierpnia 2020

Spis treści

1 Stan aplikacji

2 Projekt testowy

Dla C# mamy 3 popularne biblioteki do testowania aplikacji, będące bazami dla 3 typów projektów testowych:

- MSTest najstarszy, niemrawo rozwijany przez Microsoft,
- NUnit, xunit tworzone przez społeczność dotnetową,
- xUnit ma z tych 3 najprostszą składnię testów, więc to dobry wybór na początek.

Pojawiają się anotacje - słowa w nawiasach kwadratowych tuż nad metodami. Anotacja "Fact" mówi, że zaznaczona metoda to metodą testową.

Wciśnięcie PPM na projekcie testowym w eksploratorze rozwiązania -> run tests, jak nazwa wskazuje, odpala testy.

Luźno latające okienka w IDE (przynajmniej niektóre), takie jak to z testami można, przez przeciągnięcie gdzieś do rogu, "zadokować", czyli sprawić, żeby zostały w wyznaczonym miejscu i nie znikały po użyciu.

Zielone i czerwone kółka przy metodach testowych oznaczają wynik ostatniego testu i są skrótami do odpalania tego testu.

3 Testy jednostkowe

Test jednostkowy powinien zachowywać tak zwaną strukturę "AAA", co jest skrótem od słów:

- Arrange (Przygotowanie). Tu przygotowujemy dane testowe.
- Act (Działanie). Tu wykonujemy to, czego działanie chcemy przetestować.
- Assert (Weryfikacja). Tu sprawdzamy poprawność tego, co stało się w poprzedniej części. Używamy do tego (metod) klasy Assert obecnej w praktycznie każdej bibliotece testowej.

Te części to po prostu kawałki kodu wykonywane po kolei i tworzące logiczne całości.

Testy jednostkowe są do sprawdzania czy najmniejsze części aplikacji (metody klas) działają poprawnie. Ten typ testów zawsze pisze programista. Testerzy są od testów obejmujących szerszy kontekst.

Istnienie testów oszczędza nam debugowania.

Metody nie powinny być zbyt duże, żeby łatwiej było się upewnić że działają poprawnie. Te części które aktualnie nie są testowane, zakładamy że działają i symulujemy ich poprawne działanie przy użyciu biblioteki Moq.

4 Biblioteka Moq

Do testów nie używamy danych z bazy danych, tylko danych testowych.

Biblioteka Moq jest do tworzenia danych potrzebnych do testów i symulowania zachowania części aplikacji związanej z danymi. Te dane testowe i symulacje są po to, żeby trzymać bazę danych z daleka od testów jednostkowych, bo to nie ona jest tam testowana i nie wiadomo w jakim jest stanie. Stan bazy danych zapewne sprawdzany jest albo w ramach jakiegoś innego typu testów albo ręcznie.

Opcję zainstalowania biblioteki znajdziemy przez PPM na projekt testowy -> Manage NuGet Packages.

Obiekt klasy Mock<SymulowanaKlasa> (tym razem Mock, nie Moq) służy do symulowania działania klasy, która nie jest testowana w aktualnym teście, ale jest używana przez testowaną metodę. Jak zachowuje się obiekt "mock" definiujemy np. poprzez mock.Setup(...).Returns(...). Jeśli chcemy użyć zasymulowanego obiektu zamiast prawdziwego, to zamiast nazwy zmiennej typu SymulowanaKlasa pisemy mock.Object.

5 Biblioteka FluentAssertions

Metody z tej biblioteki wołamy tak, jakbyśmy wołali metody sprawdzanego obiektu-odpowiedzi, a metody klasy Assert wołamy jako Assert.Metoda(...). Możemy je wołać również na obiektach typów Mock<...>.

Kolejność Assertów ma znaczenie, bo funkcja testowa kończy się na pierwszym oblanym teście. Najpierw sprawdzamy bardziej ogólne warunki, np. jeśli wołamy metodę Dodaj(2,2), to najpierw sprawdzamy, czy wynik dodawania jest liczbą, a potem czy jest równy 4.

6 Co testować?

Jak coś nie zawiera logiki, np. konstruktor, standardowy getter czy setter albo proste operacje CRUD (które zwykle możemy znaleźć w Serwisie), to nie ma sensu tego testować. Jeśli jednak Serwisy zawierają trochę logiki (prawdopodobnie tak w aplikacji konsolowej, prawdopodobnie nie w aplikacji sieciowej), to też je testujemy. Natomiast np. klasy ze słowem Manager w nazwie zwykle mają metody które reagują na działania użytkownika i te metody na pewno chcemy testować.

Testy jednostkowe są do metod (funkcji). Nie testujemy testami jednostkowymi np. czy (klasa) Koń jest (czyli dziedziczy po klasie) Zwierzęciem i czy ma (pole typu lista, długości) 4 (zawierające obiekty typu) Nogi. Testujemy natomiast, np. czy metoda Koń.Idź(...) zwraca sensowne nowe położenie konia.

Nie testujemy też czy zewnętrzne biblioteki dobrze działają i generują sensowny kod. Tym zajmują się twórcy tych bibliotek. Polegamy tu na ich reputacji.

7 TDD

TDD, czyli Test Driven Development, polega na tym, że najpierw piszemy test, a potem kod wymagany do jego zaliczenia. Następnie przechodzimy do kolejnego testu, potem kodu wymaganego do jego zaliczenia, i tak dalej.

Oczywiście, zaraz po napisaniu testu do nieistniejącego jeszcze kodu, oblewamy ten test. Chodzi o to, żeby zaraz po oblaniu testu dopisać do aplikacji tylko tyle kodu ile potrzeba żeby zaliczyć test.

Jeśli natrafiamy podczas pisania testu na sytuację w której test nie kompiluje się, bo brakuje jakiegoś kodu w aplikacji, to wtedy dopisujemy ten kod, np. pustą metodę. Ciało tej metody dopisujemy dopiero wtedy, gdy test zaczyna tego wymagać.

Jeśli nowa metoda będzie potrzebowała skorzystać np. z metod serwisu, to przy pomocy mock.Setup() ustawiamy jak te metody powinny się zachowywać.

8 Testy integracyjne

Sprawdzają czy różne warstwy aplikacji, np. Kontrolery i Menadżery, dobrze ze sobą współpracują.

Tutaj nie używamy Mocków, tylko prawdziwych obiektów. Zapisujemy, odczytujemy, usuwamy dane z bazy używanej też przez samą aplikację. Zwracamy więc uwagę, żeby test zostawił bazę w takim stanie w jakim ją zastał. Na koniec testu integracyjnego możemy więc mieć dodatkowy krok Czyszczenia (Clear), gdzie właśnie przywracamy bazę do stanu sprzed wykonania testu.

Na etapie czyszczenia, jeśli wcześniej dodawaliśmy do bazy jakieś obiekty, to konieczne może być nie tylko ich usunięcie, ale też posprzątanie innych śmieci, które mogły po tych obiektach pozostać w bazie. Np. zwrócić identyfikatory obiektów zużyte do testów, żeby potem w normalnym działaniu aplikacji nie powodować przerw w numerowaniu obiektów. Także najpierw uczymy się przywracać stan bazy danych do stanu sprzed testu, a dopiero potem zabieramy się za testy integracyjne.

9 Typowe błędy

Każda aplikacja potrzebuje testów. Każda. Jeśli jakaś funkcjonalność już jest i masz pewność że działa poprawnie, to nie znaczy że masz pomijać pisanie dla niej testów. Mogą Cię one zabezpieczyć przed zepsuciem tej działającej części w przyszłości.

Przemyśl co testujesz. Czy na pewno jest to napisana przez Ciebie (względnie Twój zespół) logika aplikacji, a nie proste operacje i zewnętrzne biblioteki? Patrz rozdział 6.