Notatki - Testowanie - TDD

Spis treści

TDD

~~To Done Do?~~ **Test Driven Development**. Jest podejściem do programowania, w którym definiujemy przypadki testowe określające co ma robić i jak ma się zachowywać kod, który piszemy, zanim go napiszemy. W praktyce sprowadza się to do tego, że zanim napiszemy kod, piszemy testy. Początkowo żaden z takich testów nie zakończy się sukcesem, ale stopniowo jak będziemy dopisywać kolejne funkcjonalności, testy zaczynają świecić się na zielono, aż osiągniemy stan, gdzie wszystkie testy zakończą się sukcesem.

W tym podejściu development zaczyna się od wymyślenia jakie funkcjonalności mają być dostarczone z perspektywy podziału na metody testowe. Inaczej na to patrząc, patrzymy jakie testy chcielibyśmy mieć w swoim kodzie, które potem mają działać. Jeszcze inaczej mówiąc, pisząc testy możemy zdefiniować wymagania, jakie ma spełniać nasz kod. Następnie implementujemy takie wymagania faktycznie ten kod pisząc. Inaczej używa się czasem określenia **Test First Development**, z racji, że testy są pisane najpierw.

Obraz 1. Cykl Test Driven Development

Czyli podchodząc do tematu krokowo, pisalibyśmy program w taki sposób:

- Dodaj test,
- Uruchom wszystkie testy i zobacz czy któryś nie przechodzi,
- Dopisz kod, aby test przechodził,
- Uruchom testy i jeżeli wszystkie przechodzą możesz refactorować swój kod,
- Powtórz.

Jeżeli chodzi o filozofię stosowania takiego podejścia, wyrób ją samodzielnie. Aby było prościej, postaraj się napisać projekt zaczynając od testów i zobacz jak będzie Ci to szło.

Poniżej umieszczam zalety i wady stosowania **TDD**, natomiast wiele z poniższych kwestii można poruszyć przy zadaniu sobie pytania: "Czy w ogóle pisać testy?" - Moja jednoznaczna odpowiedź brzmi **TAK**.

Zalety stosowania podejścia TDD

- z czasem jesteśmy w stanie osiągnąć stan, w którym mamy kod wraz z zestawem testów, które są w stanie automatycznie ten kod sprawdzać, oczywiście pisząc w sposób odwrotny (czyli najpierw kod, potem testy) w pewnym momencie również osiągniemy ten stan,
- pomaga zrozumieć kod zanim go napiszemy, gdyż myślimy o tym jak różne fragmenty kodu mają ze sobą rozmawiać,
- pomaga rozbić kod na mniejsze części, gdyż zależy nam na tym, aby metody testowe były jak najmniejsze. Wtedy każda metoda faktycznie realizuje swoją funkcję,
- daje nam pewność siebie w momencie gdy zaczynamy coś w kodzie zmieniać. W praktyce często zmiana kodu, modyfikacja czy refactor powoduje, że możemy bardzo łatwo wprowadzić błędy. Często jest niemożliwe sprawdzenie wszystkich przypadków, które mogliśmy popsuć. Dlatego lepiej jest mieć automat, który sprawdza popełniane przez nas błędy,
- w przypadku pracy zespołowej, jeżeli najpierw napiszemy wspólnie testy to łatwiej jest podzielić

pracę, ustalić kto potem ma implementować różne fragmenty systemu,

Wady stosowania podejścia TDD

- stosując podejście TDD zaczniemy mieć problem jeżeli często zmieniane są wymagania dotyczące tego co faktycznie kod przez nas pisany ma robić. Wyobraź sobie, napisaliśmy już kilka testów, podzieliliśmy to w głowie, zaczynamy pisać kod i przychodzi mail, że zaraz jest spotkanie, na którym się dowiesz, że połowa z tego co zostało przez Ciebie wymyślone jest do wyrzucenia. Więc znowu poprawiasz testy, chcesz zacząć pisać kod i dowiadujesz się, że zmienione zostały wymagania. W takim przypadku znacznie wydłuża to czas developmentu,
- możemy również wprowadzić błędy w samych testach, wtedy napiszemy błędny kod, który powoduje, że błędne testy przechodzą,
- takie podejście jest często ciężko obronić przez biznesem (mówię z praktyki). Bo z perspektywy biznesu, dowozimy funkcjonalność o wiele dłużej, niż byśmy to robili nie pisząc testów wcale. Natomiast jest to perspektywa bardzo krótkoterminowa, bo w długiej perspektywie zapewniamy sobie możliwość popełnienia mniejszej ilości błędów. Ten punkt w sumie jest niezależny od tego czy piszemy najpierw testy, czy kod. Pomaga tutaj jak cały zespół ma spójne podejście i wszyscy mocno naciskają na pisanie testów uzasadniając dlaczego ma to sens.

Filozofia

To może na koniec skupmy się jeszcze na filozofii dotyczącej testów. Zapoznaj się z poniższymi dywagacjami ☺.

To co nam dają te testy

Refactor

Bezpieczeństwo - to przede wszystkim. Gdy masz już napisane testy do swojej aplikacji (tutaj nie ma znaczenia, czy w podejściu TDD, czy inaczej) możesz bezpiecznie podejść do jakichkolwiek modyfikacji kodu źródłowego. Czemu bezpiecznie? **Bo "czuwa" nad Tobą magiczny sprawdzacz**, który weryfikuje czy zmiana jednej linijki, nie popsuje Ci zachowania całej aplikacji. Testy w ten sposób pomagają w refactoringu kodu. Jeżeli jakiś fragment kodu jest dobrze przetestowany i pokryte są przypadki użycia (najlepiej wszystkie), możesz spokojnie przejść do refactorowania. Co więcej, jeżeli czytasz jakiś fragment kodu i dochodzisz do wniosku, że przydałby się refactor, warto jest zacząć od napisania testów, jeżeli dany fragment kodu takich testów nie posiada. Ponownie podkreślę - daje Ci to **bezpieczeństwo w Twoich działaniach**.

Dokumentacja

Wyobraź sobie, że przychodzisz do nowego projektu i chcesz się zapoznać z kodem źródłowym. Czytasz i czytasz, ale dalej nie rozumiesz, jak to działa, bo nie wiesz jakiego rodzaju dane będą przepływały przez dany fragment kodu. Co w takiej sytuacji? **Szukasz testów, one pomogą Ci zrozumieć pełny obraz wywołania danego fragmentu kodu.** Dlatego właśnie jednym z najlepszych rodzajów dokumentacji są dobre testy. Z nich można wyczytać, jakie są dane wejściowe dla danego przypadku i jakie są oczekiwane rezultaty. Co więcej, możesz takie testy debuggować i zobaczyć jak dana linijka kodu

zachowuje się dla różnych danych, jakie wartości będzie przyjmowała zmienna itp.

Jeżeli chcesz pomóc innym i *sobie z przyszłości*, pisz dobre testy. Uwierz mi, że po czasie zapominasz, jak napisany przez Ciebie kod działa i co tak naprawdę miałeś/miałaś na myśli. Dobre testy są wtedy jak skarb.

Poprawność

Ten podpunkt zrozumiesz solidnie dopiero po zaliczeniu kilku takich sytuacji, ale i tak go napiszę ☺. Jeżeli zaczniesz pisać testy do istniejącego już kodu, to bardzo szybko nastąpi weryfikacja, czy dany fragment kodu jest dobrze zaprojektowany. Co mam na myśli? Jeżeli kod jest źle zaprojektowany, to może się okazać, że nie da się go łatwo przetestować, albo czasami w ogóle nie da się go przetestować. Wtedy najczęściej trzeba robić jakieś fikołki, żeby najpierw zmienić ten kod (tu jest ryzyko, że zmienisz logikę zawartą w tym kodzie, bo nie masz przecież testów), a dopiero potem dopisujesz testy. Opcja druga to zaczynasz od TDD, najpierw piszesz testy, a potem usuwasz całą implementację i piszesz ją na nowo. Takie kwestie to już jest kwestia przypadku, ale można podsumować ten podpunkt w taki sposób, że testowanie kodu pomaga nam trzymać się pewnych wzorców (o wzorcach będzie w dedykowanym warsztacie), bo testy w pewnym sensie to na nas wymuszają.

Metody void

A co z testami jednostkowymi metod zwracających `void`? Co wtedy?

Metody `void` (jak już wiesz) nic nie zwracają. Mogą natomiast robić inne rzeczy:

- zmienić stan obiektu przekazywanego jako argument metody,
- wyrzucić wyjątek w trakcie wykonania takiej metody,
- logować przebieg działania aplikacji - to już jest trochę bardziej skomplikowane, ale można w testach weryfikować, czy jakaś informacja została zalogowana na ekranie.

W takich przypadkach również można przeprowadzać testy takich metod.

Podsumowanie

Dodam jeszcze na koniec, że materiał jest tak podzielony, żeby ten warsztat dawał namiastkę testowania i był tylko wprowadzeniem. Do tematyki testów będziemy wracać jeszcze wielokrotnie w kolejnych warsztatach. Jeżeli chcesz sprawdzić gdzie, zapoznaj się z zakładką 'Lista warsztatów'.