**1 Testowanie - Teoria**

**Czym jest testowanie?**

Zanim zagłębimy się w teorię i zasady dotyczące pisania testów jednostkowych, podejścia TDD, czy zasad testowania oprogramowania, chciałbym sparafrazować (słyszałem dawno, a nie pamiętam dokładnie treści) pewien cytat odnoszący się do pisania testów oprogramowania.

Robię to, głównie po to, aby uświadomić Cię, jak ważny temat właśnie w tej chwili zaczynamy omawiać...

*Gdybyś Ty, programisto, miał do dyspozycji dwa katalogi plików i jeden z nich przechowywałby działający i w pełni funkcjonalny kod produkcyjny, a drugi tylko i wyłącznie kod testujący funkcjonalność stworzonego programu i pewnego razu zostałbyś zmuszony, aby permanentnie usunąć jeden z wymienionych katalogów, to który* ***byś wybrał, chcąc ponieść jak najmniejsze straty****?*

Przyznaj, że intuicja podpowiada Ci, aby usunąć katalog z testami. Okazuje się jednak, że lepszym rozwiązaniem byłoby usunięcie katalogu z kodem produkcyjnym.

A to dlatego, że dobrze napisane testy, **umożliwiają łatwe napisanie na nowo programu, który testują**.

Co więcej, jestem w stanie stwierdzić, że testy są **najlepszą dokumentacją** stworzonego przez Nas programu! Szybki wgląd w dany moduł testujący i od razu widzimy, jak powinien zachować się Nasz program dla danego przypadku (tzw. **case'u testowanego**).

Testy mają ogromne znaczenie w pisaniu efektywnego oprogramowania, dlatego też konieczne jest, abyś poznał **techniki testowania** i **rodzaje testów**, z jakimi programiści mogą mieć styczność...

**Typy i poziomy testów oprogramowania**

Uwierz mi, że jest naprawdę dużo rodzajów testów oprogramowania. Dzielimy je zarówno ze względu na ich **typ, poziom** (etap wytwarzania oprogramowania), jak i **sposób przeprowadzania testu**.

**Testy manualne i automatyczne (sposób przeprowadzania testu)**

Na sam początek warto rozróżnić dwa rodzaje testów - **testy manualne** i **automatyczne**. Testy manualne polegają na **ręcznym przeklikiwaniu** się przez program i sprawdzaniu jego poprawności. Do tego celu, testerzy manualni mogą wykorzystywać liczne **interfejsy API** i **programy zewnętrzne**. Przykładem testerów manualnych są choćby testerzy gier komputerowych, którzy po prostu w nią grają, szukając bugów.

Natomiast umiejętność przeprowadzania testów automatycznych to punkt wymagany przez jakieś **80% ofert pracy na stanowisko testera oprogramowania**. A to dlatego, że testy automatyczne są o wiele **mniej kosztowne i bardziej wydajne** niż testy manualne. Przeprowadzanie testu automatycznego polega na przygotowaniu skryptu, który będzie uruchamiany przez maszynę i będzie automatycznie wykonywał określoną sekwencję czynności, sprawdzając, czy dla każdej z nich testowany program zachowuje się w oczekiwany sposób.

Testy automatyczne stanowią kluczowy element praktyk takich jak **ciągła integracja** oraz **ciągłe dostarczanie** (**CI - Continuous Integration, CD - Continuous Delivery**). Umożliwiają również efektywne i sprawne skalowanie aplikacji do potrzeb klienta. Zaznaczę jednak, że mimo jakościowej przewagi testów automatycznych nad testami manualnymi, warto przeprowadzać ten drugi rodzaj testowania, korzystając z praktyki tak zwanego **testowania eksploracyjnego** (**testowanie eksploracyjne** to nieformalna technika projektowania testów, w której tester projektuje testy w czasie, gdy są one wykonywane i wykorzystuje informacje zdobyte podczas testowania do projektowania nowych i lepszych testów).

**Poziomy testowania**

**Testy jednostkowe**

Są przeprowadzane na bardzo niskim poziomie aplikacji (w zasadzie kodzie źródłowym) i polegają na testowaniu poszczególnych (jednostkowych) składowych aplikacji, czyli metody, klasy, komponenty czy moduły. Tym rodzajem testowania zajmiemy się w kursie PyTest.

**Testy integracyjne**

Tutaj testy wykonywane są już poziom wyżej niż testy jednostkowe. Mogą one testować, np. **połączenie aplikacji z bazą danych**. Wykonywanie tego poziomu testów jest droższe, ponieważ wymagają one uruchomienia wielu elementów aplikacji.

**Testy poziomu E2E (end-to-end)**

Testują złożone zachowania aplikacji przeprowadzane w jej środowisku, np. poprawność procesu logowania się do serwisu, zapisywania się na newsletter i otrzymywania powiadomienia itd. Testy end-to-end są bardzo kosztowne, dlatego zaleca się przeprowadzenie kilku z nich i poleganiu głównie na testach niższego poziomu (wykrycie bowiem defektu na wcześniejszym - niższym - etapie produkcji, łatwo skorygować, aniżeli przebudowywać całe środowisko po znalezieniu błędu na etapie testowania E2E).

**Testy akceptacyjne**

Obejmują szeroki zakres testowania zachowań aplikacji. Do ich zakresu może również dojść testowanie wydajności systemu. Ze względu na to, że jest to najwyższy poziom tworzenia oprogramowania testującego, przewidują jak najmniej zmian, choćby z względu na ich dużą kosztowność i opóźnienie w dostarczaniu produktu do klienta (testy akceptacyjne przeprowadza się już na etapie finalny dostarczania oprogramowania).

**Blackbox, whitebox. Typy testów**

Zanim przejdziemy do konkretów, chciałbym zwrócić uwagę na pewną cechę testów - otóż mogą one być **blackbox'owe** lub **whitebox'owe**. Różnice ograniczają się do tego, czy jako testerzy, mamy wgląd do kodu, który testujemy (wtedy mówimy o teście typu **whitebox**), czy też nie i bazujemy jedynie na dokumentacji i specyfikacji **(blackbox)**. Jeżeli więc na przykład, jesteś równocześnie programistą tworzącym kod produkcyjny i samodzielnie dostarczasz do niego testy, to masz wtedy do czynienia z whitebox'em.

A teraz wracając do meritum. Czterema najbardziej znanymi **typami** testów są:

* **strukturalne**
* **funkcjonalne**
* **niefunkcjonalne**
* **regresywne, retesty**
* **inne rodzaje testów (obciążeniowe, stres-testy, alfa i beta testy)**

**Testy strukturalne -** rodzaj testów whitebox, dążymy w nich do 100% pokrycia kodu testowanego (dążymy do tzw. **100 % UT coverage**), czyli chcemy przetestować każdy możliwy scenariusz w aplikacji.

**Testy funkcjonalne -** testy blackbox'owe. Najczęściej wykonują je osoby spoza zespołu autorów kodu produkcyjnego, często są to osoby nieposiadające wiedzy z zakresu programowania, np. zwykli potencjalni użytkownicy. Głównym założeniem testów funkcjonalnych jest testowanie nie struktury programu, ale jego funkcjonalności *(pamiętasz przykład odnoszący się do testów manualnych? :)*

Testy funkcjonalne możemy podzielić na:

**Testy regresywne** - testy przeprowadzane po dokonaniu zmian w kodzie, sprawdzają, czy modyfikacje i rozszerzenia nie zmieniły prawidłowego zachowania oprogramowania.

**Testy potwierdzające** - testy przeprowadzane po naprawieniu danego błędu.

**Retesty** - testy przeprowadzane po wykryciu błędu, naprawienia go, w celu sprawdzenia funkcjonalności programu na nowo.

**Testy niefunkcjonalne** - w przeciwieństwie do testów funkcjonalnych, testują program pod względem choćby: niezawodności, wydajności, możliwości uruchomienia na różnych platformach itp.

Testy performance - dzielą się na:

**Testy obciążeniowe** - testy, które sprawdzają jak zachowuje się Nasze oprogramowanie, gdy zwiększamy, np. liczbę użytkowników z niego korzystających, liczbę transakcji, jakie są w nim przeprowadzane.

**Testy wydajnościowe** - są to najczęściej prowadzone pod kątem badań biznesowych mających oszacować jakie są wymagania dla działania aplikacji, porównanie czasów wysyłanych oraz odbieranych requestów dla grupy użytkowników lub tego czy zostały spełnione wyznaczone dla działania aplikacji kryteria pod kątem wykonywania w akceptowalnym czasie.

**Testy przeciążeniowe (stres-testy)** - testy przeprowadzane w niekorzystnych warunkach, np. przy wykorzystaniu maksymalnej ilości dostępnych zasobów. W ten sposób badamy, jak zachowa się napisana aplikacja w skrajnie złych warunkach.

**Alfa testy** - testy, których celem jest walidacja oprogramowania. Wykonywane są zanim produkt zostanie dostarczony do klienta i najczęściej przeprowadzane są przez dział QA (Quality Assurance).

**Beta testy** - testy przeprowadzane przez kilku wybranych klientów, sprawdzają jak zachowuje się aplikacja w docelowym środowisku do uruchomienia.

Zaskoczę Cię - to tylko część możliwych testów, jakie testerzy mogą przeprowadzić. Istnieje wiele innych typów, jednak powyżej znajdują się te, z którymi sam miałem do czynienia w pracy zawodowej i są najczęściej wykorzystywane. Jednak sama wiedza dotycząca tego, czego mogą dotyczyć konkretne poziomy i rodzaje testów nie wystarczy do efektywnej pracy jako programista czy tester oprogramowania. Konieczne jest zapoznanie się z **systemem i zasadami** według których możemy skutecznie testować oprogramowanie... Poruszymy to w następnych sekcjach tego szkolenia.

**Automatyczne testy jednostkowe zgodne z metodologią TDD (Test-Driven Development)**

Jak już wiesz, w kursie nauczymy się pisać automatyczne testy jednostkowe. Jednak zanim zaczniemy tworzyć pierwsze testy, musisz wiedzieć, w jakim systemie/z użyciem jakiej metodologii (a będzie to **TDD**) powinniśmy je tworzyć.

**Czego dotyczy TDD (Test-Driven Development)?**

**Zasady metodologii TDD** zakładają kilka niezwykle ważnych praw:

**1. Nie można zacząć pisać kodu produkcyjnego, zanim nie stworzymy testu jednostkowego.**

Co to znaczy? To oznacza tyle, że musimy koniecznie zadbać o to, aby **najpierw napisać test** (de facto do nieistniejącego jeszcze feature'a oprogramowania), a dopiero później zająć się implementacją danej właściwości. Czyli np:

* Zaczynamy od napisania testu, który sprawdza, czy wkrótce zaimplementowana przez Nas funkcja, obliczająca silnię podanej liczby, zwraca prawidłowy wynik dla, np. operacji 10! (wiemy, że jest to wartość równa 10 \* 9 \* 8 \* 7 \* 6 \* 5 \* 4 \* 3 \* 2 \* 1). Dopiero po stworzeniu testu, zabieramy się za implementację właściwej funkcji.

**2. Pojedynczy test jednostkowy powinien sprawdzać co najwyżej jeden case programu** (w zasadzie od tego określenie ***jednostkowe***).

To znaczy, że test jednostkowy powinien być jak najmniejszy pod względem objętościowym oraz nie powinniśmy pisać testów "na zaś".

Musimy działać zgodnie z poniższym cyklem:

1. Napisanie testu

2. Stworzenie funkcjonalności, którą sprawdza napisany wcześniej test

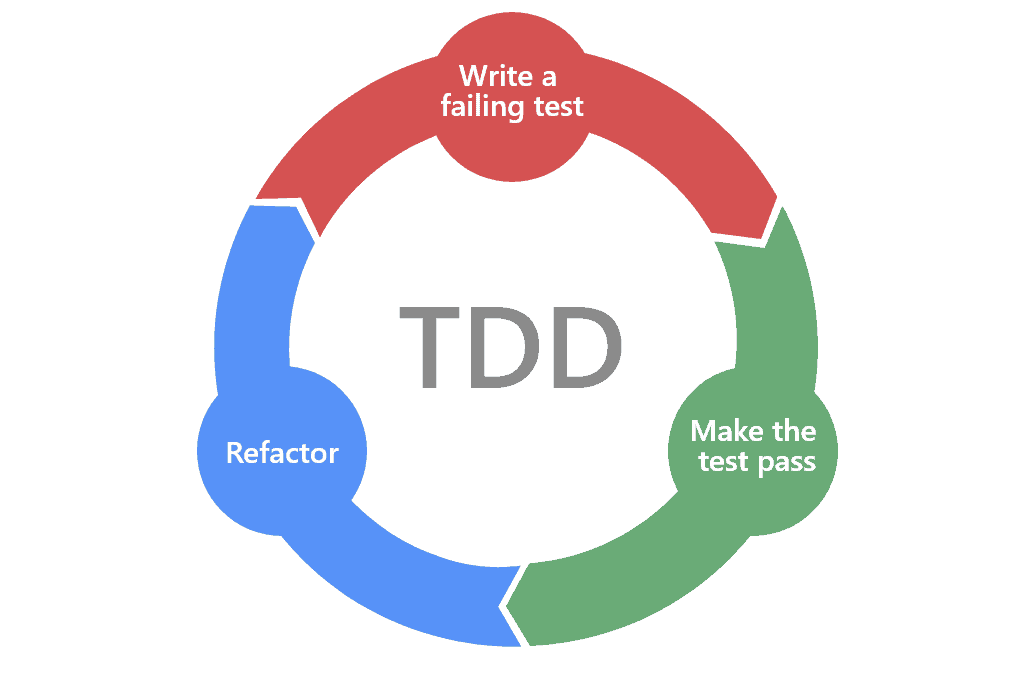
3. Ewentualna refaktoryzacja kodu.

3. Ostatnia zasada dotyczy w zasadzie wyżej przedstawionego cyklu. **Otóż należy ściśle trzymać się wszystkich kroków i nie zaburzać ich kolejności** (oczywiście w ramach zdrowego rozsądku). Na początku może to być dość trudne, jednak zapewniam Cię, że nawyk ten szybko wejdzie Ci w krew.

**Cykl TDD**

Reasumując więc... Od teraz, efektywne oprogramowanie będziemy tworzyli w następujący sposób:

Wytworzenie testu jednostkowego (w tzw. fazie czerwonej) -- Implementacja w kodzie danego feature'a (faza zielona)-- Refaktor kodu (faza niebieska).



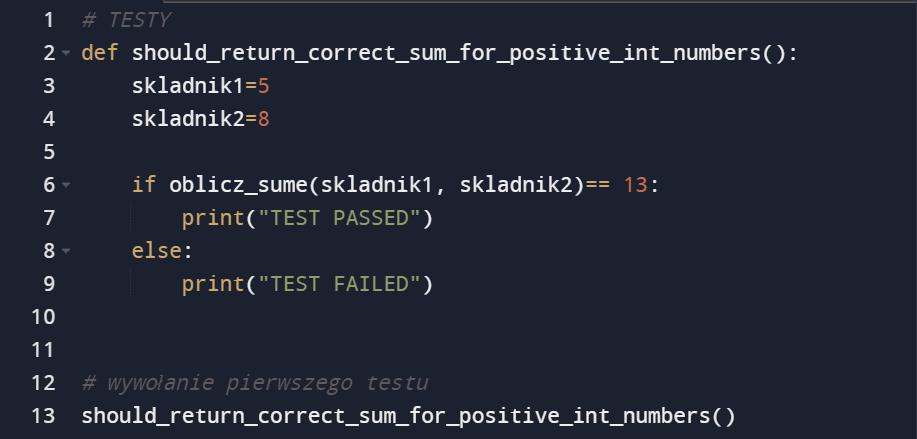
**Cykl TDD**

Jeżeli będziemy tworzyli oprogramowanie, nie łamiąc powyższej zasady, to będziemy pisali kilkanaście testów dziennie, kilkaset miesięcznie i kilkatysięcy rocznie. **Będziemy równolegle tworzyli testy i oprogramowanie!** Co więcej, z automatu testy te będą w pełni **pokrywać kod produkcyjny**! Jest to niewątpliwie zbawienie dla Naszego manager'a i zasad pisania czystego kodu (które przecież też zakładają obecność testów w projekcie).

**Przykład implementacji programu napisanego zgodnie z ideą TDD:**

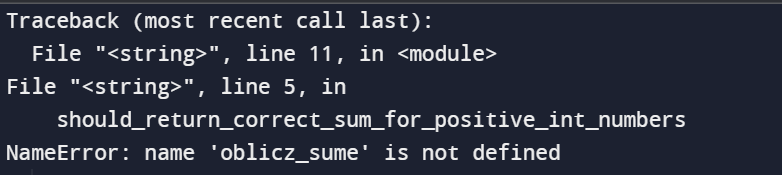
Załóżmy, że chcemy zaimplementować metodę, która będzie zwracała sumę **dwóch wartości liczbowych typu int** (trywialność przykładu na rzecz lepszego uzmysłowienia). Z racji, że nie znamy jeszcze żadnych framework'ów testujących, załóżmy, że testem będzie **prosta funkcja sprawdzająca** działanie metody sumującej.

Tak więc zacznijmy od pierwszej fazy cyklu TDD (faza czerwona):



Stworzyliśmy najprostszy test (o nazwie **should\_return\_correct\_sum\_for\_positive\_int\_numbers**), sprawdzający czy funkcja **oblicz\_sume** (którą zaimplementujemy w następnym kroku), zwraca oczekiwany wynik dla wartości dodatnich liczb.

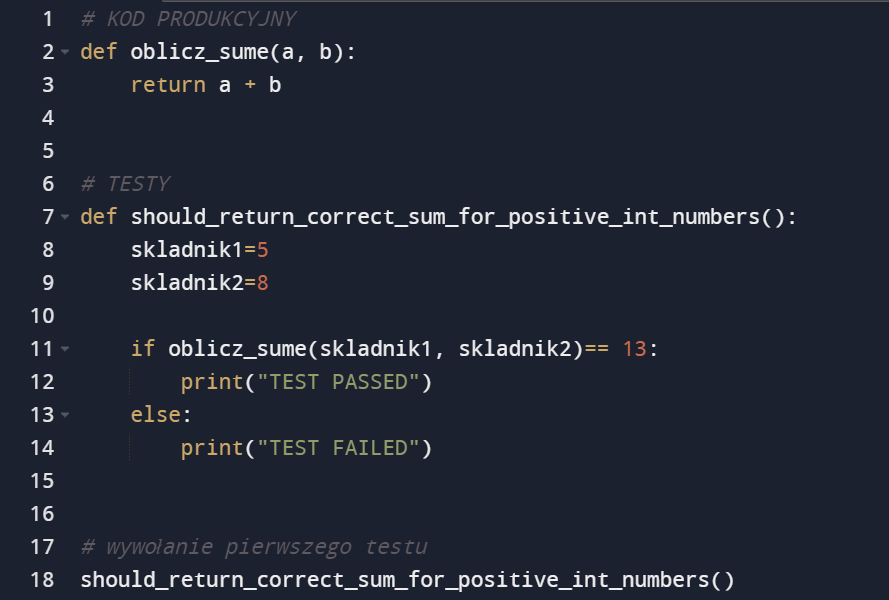
Po uruchomieniu procesu interpretacji powyższego kodu nastąpi błąd:



Mówiący o tym, iż funkcja oblicz\_sume nie została znaleziona. Czyli tak naprawdę Nasz program działa, nie działając :) Oczywiście żartuję w tym momencie, ale wszystko odbywa się zgodnie z założeniami (faza czerwona ma po prostu zwracać błąd interpretacji z oczywistych względów - implementacją metody oblicz\_sume przecież przed nami ;).

**Faza zielona:**

Faza zielona to już "naprawienie" naszego kodu i sprawienie, że przejdzie on proces interpretacji. Czyli najzwyczajniej w świecie zajmiemy się już implementacją kodu produkcyjnego!



**Faza niebieska**

No i na końcu faza niebieska, czyli refaktoring kodu. Oczywiście o czysty kod powinniśmy dbać już w momencie tworzenia funkcjonalności, ale jeżeli wkradły się błędy estetyczne, to najwyższy czas by je w tej fazie poprawić. Poprawy wymagają **linie 8, 9 oraz 11** powyższego kodu (brak spacji w odpowiednich miejscach).

**Potwarzalność cyklu**

I tak wyżej opisany cykl powtarzamy. To znaczy po ukończeniu fazy niebieskiej, z powrotem wracamy do fazy czerwonej, piszemy testy do następnego feature'a itd. W taki właśnie sposób wytwarza się oprogramowanie w cyklu TDD.

**Konwencje nazywania testów**

Z powyższego przykładu,na pewno Twoją uwagę zwróciła dość długa nazwa stworzone testu jednostkowego (should\_return\_correct\_sum\_for\_positive\_int\_numbers). Fakt, nazwa wygląda dość ekscentrycznie, ale powinieneś zacząć się przyzwyczajać do jej długości. Testy jednostkowe nazywamy bowiem zgodnie z pewnymi zasadami i de facto może to prowadzić do faktycznie tak przydługich ciągów liter. Przyznaj jednak, że lepiej poświęcić chwilę na porządne i wyczerpujące nazwanie danego testu, aniżeli później poszukiwanie miejsca i przyczyny, która spowodowała niezaliczenie testu. Widząc bowiem FAIL przy teście o nazwie should\_return\_correct\_sum\_for\_positive\_int\_numbers), od razu wiemy, że Nasz program oblicza nieprawidłową sumę dla wartości dodatnich i błyskawicznie jesteśmy w stanie to skorygować.

Najpopularniejsze sposoby nazywania testów:

1. **Should\_ExpectedBehavior\_When\_StateUnderTest**, np.should\_throwexception\_when\_ageLessThan18
2. **When\_StateUnderTest\_Expect\_ExpectedBehavior,** np. when\_ageLessThan18\_expect\_isAdultAsFalse
3. **Given\_Preconditions\_When\_StateUnderTest\_Then\_ExpectedBehaviour,** np. Given\_UserIsAuthenticated\_When\_InvalidAccountNumberIsUsedToWithdrawMoney\_Then\_TransactionsWillFail

**Zasady pisania czystych testów**

Z tej sekcji chciałbym, żebyś w zasadzie zapamiętał jedną najważniejszą kwestię: **jakość** **testów jest tak samo ważna jak kodu produkcyjnego**.

Chcę zwrócić na to szczególnie dużą uwagę, ponieważ wsród początkujących programistów zauważyłem niezbyt dobry nawyk nieprzywiązywania dużej uwagi to czystości pisania testów. Łamią oni wszelkie zasady Clean Code, który dotyczy choćby poprawności nazywania zmiennych, nietworzenia tzw. God Testów, które sprawdzają wiele zależności, rodzielania testów na jak najmniejsze części składowe (jednostki) itd.

Otóż pisanie "brudnych" testów prędzej czy później prowadzi do trudności w utrzymaniu kodu. Rozwijanie takich testów staje się na tyle uciążliwe, że programiści w końcu decydują usunąć je z kodu. A to pociąga kolejny defekt... Zespół tak postępujący nie jest w stanie tworzyć w pełni efektywnego oprogramowania (bez testów nie ma bowiem pewności, że tworzymy w działający kod produkcyjny).

Liczba błędów będzie wzrastała, a programiści będą obawiali się wprowadzać nowe funkcjonalności przez co rozwój produktu stanie w miejscu. Klaps totalny! Dlatego też, odnosząc się jeszcze raz do zasady z pierwszego akapitu - **dbaj o jakość kodu testującego w tak samo dużym stopniu, jak dbasz o czystość kodu produkcyjnego** :)

Jeżeli natomiast zastanawiasz się, jakie są najważniejsze zasady pisania kodu zgodnego z zasadami Clean Code, to odsyłam CIę do modułu z szkolenia poświęconego dobrym wzorcom projektowania apikacji!

**F.I.R.S.T**

Zanim przejdziemy do konkretnego już framework'a umożliwiającego tworzenie testów, chciałbym zatrzymać się na cechach czystych testów, które wykorzystują powyższy akronim. Uznałem, że warto zwrócić na nie szczególną uwagę, bo nie pokrywają się w pełni z standardowymi zasadami tworzenia czystego oprogramowania.

**Fast (Szybkie)** - logiczne jest, że gdy testy są wolne i zajmuje dużo czasu choćby ich uruchomienie, to podświadomie ograniczamy częstotliwość ich wykrozystania. A więc nie uruchamiamy ich zbyt często, nasz kod nie jest testowany i rezultatem może być wprowadzenie błędów. Dlatego twórz szybkie i wydajne testy! Zwracaj uwagę na ich złożoność, ilość dependencji, do których się odnoszą itd.

**Niezależne (Independent)** - testy powinny być niezależne od siebie. Czyli nie powinniśmy tworzyć żadnych kaskadowych powiązań. Testy powinny być uruchamiane w dowolnej kolejności. Jeśli testy zależą od siebie, to w momencie gdy nie uda się pierwszy test, powstaje kaskada awarii, co utrudnia diagnozę i ukrywa defekty na niższym poziomie.

**Powtarzalne (Readable)** - testy powinny być cross-platformowe i przenoszalne. To znaczy tyle, że powinno możliwe być ich uruchomienie w dowolnym środowisku - czy to w środowisku produkcyjnym, czy środowisku QA (Quality Assurance) albo nawet na osobistym laptopie.

**Samokontrolujące się (Self-Validating)** - czyli testy powinny dawać jednoznaczny wynik - powiodły się lub nie (PASSED or FAILED). Nie powinniśmy czytać jakichś dzienników czy jeszcze innych zależności, z których moglibyśmy dopiero wywynioskować, czy dany test się powiódł. Jeżeli testy nie są samokontrolujące się, to wówczas analiza błędów staje się utrudniona i czasochłonna.

**O czasie (Timely)** - zasada ta bezpośrednio nawiązuje do cyklu TDD - testy jednostkowe powinny być pisane zawsze przed tworzeniem kodu produkcyjnego. Jeżeli pisalibyśmy testy po stworzeniu kodu produkcyjnego, mogłoby się okazać, że kod ten byłby trudny, a wręcz niemożliwy do przetestowania.

**Zakończenie**

Po dogłębnej analizie powyższych sekcji, jesteśmy już gotowi przejść do praktyki i zacząć pisać pierwsze testy jednostkowe dla Naszego kodu produkcyjnego. Słowem wstępu - testować będziemy w Pythonie - tak jak cały kurs zakłada - a do przeprowadzania testów wykorzystamy obecnie najpopularniejszy moduł **pytest**. Do zobaczenia w następnych tematach!