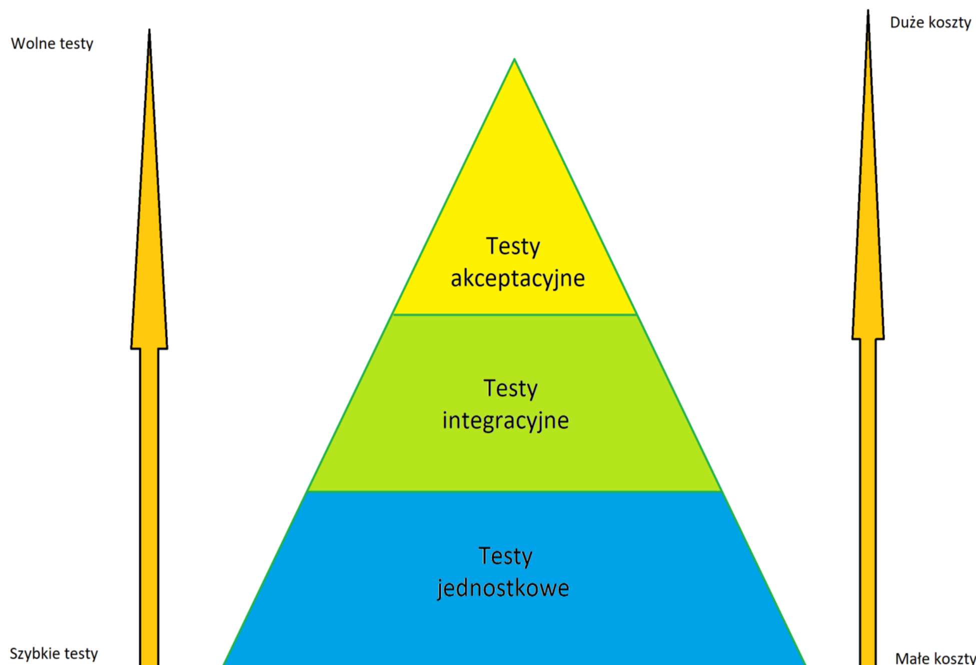
# Piramida testów i ciągła integracja

## 1. Wprowadzenie

W procesie wytwarzania oprogramowania często spotykamy się z takimi pojęciami jak: piramida testów, ciągła integracja, poziomy testów. Znane są też dobre praktyki, jak np. FIRST. Nie zawsze jednak mówimy o tym, jak połączyć dobre praktyki, piramidę testów oraz ciągłą integrację. Okazuje się jednak, że **znajomość książkowych definicji wcześniej wymienionych pojęć to za mało.** Podczas wytwarzania oprogramowania bardzo ważna jest wiedza o tym, jak je ze sobą łączyć. Właściwe połączenie wspomnianych koncepcji przysłuży się podniesieniu efektywności wytwarzania oprogramowania zwłaszcza, gdy używana jest metodyka zwinna.

## 2. Piramida testów a zasada FIRST

Piramida testów to graficzny model, który prezentuje zależność pomiędzy poziomami testów oraz ich ilością. Na tym modelu można również zaprezentować "narzut kosztowy" wytwarzania testów dla każdego poziomu:



Koncepcja piramidy testów została wymyślona po to, aby uświadomić nam, jakie powinny być relacje pomiędzy ilością testów a ich poziomem. Celem tej koncepcji jest również optymalizacja czasu wykonania się testów jako całości. W metodologii zwinnej zmiany w kodzie wprowadza się nawet kilka razy dziennie, dlatego czas testów nie powinien być zbyt długi. Idąc od dołu:

a) Definicja testów jednostkowych zmieniała się z czasem. Początkowo mówiło się, że służą one do tego, aby sprawdzić poprawność działania klas, metod, obiektów w programowaniu obiektowym albo procedur w programowaniu proceduralnym. Ten rodzaj testów jest “najszerszym” fundamentem piramidy testów. Taki sposób prezentacji nie oznacza jednak, że ilość testów jednostkowych musi być rzędu 10000 czy 15000. Ilość testów powinna być podyktowana poprzez identyfikację warunków testowych za pomocą odpowiednich technik. Testy jednostkowe są bardzo szybkie, więc dodanie kilku nowych nie wpłynie negatywnie na czas ich wykonania. Zakres tych testów powinien być jednak przemyślany, ponieważ wykonywanie wielu takich testów w izolacji od innych klas nie daje dużej wartości biznesowej. Trywialne testy powinno się minimalizować, a nie zwiększać ich ilości. Koncepcja piramidy testów będzie miała największy sens, jeśli założymy, że to testy funkcjonalne sprawdzają logikę biznesową w izolacji od zewnętrznych komponentów, np. baza danych, zewnętrzne api itd. Jeśli dana funkcjonalność jest zaimplementowana tak, że korzysta z wielu klas, to również można ją przetestować jednostkowo. Takie podejście wniesie w ten rodzaj testów o wiele więcej wartości, niż testowanie trywialnych metod.

b) Testy integracyjne służą do tego, aby sprawdzić integrację serwisów lub integrację systemów (duże testy integracyjne). Ten rodzaj testów ma na celu sprawdzenie, czy **serwisy albo systemy potrafią się ze sobą komunikować**. Poprawność przetwarzania danych, czy wynik funkcji możemy sprawdzić jednostkowo. Ten rodzaj testów powinien być skierowany głównie na współdziałania serwisów. Analogicznie do testów jednostkowych, mogą być one wykonywane w izolacji od komponentów zewnętrznych oraz są szybkie. Ilość tych testów powinna być również podyktowana identyfikacją warunków testowych. Można również wziąć pod uwagę testy negatywne (sprawdzające odporność na błędy lub niepoprawne zachowanie użytkownika), o ile ma to sens. Koszt wytwarzania utrzymania tych testów jest na ogół nieco większy niż testów jednostkowych, ale nie musi to być regułą w każdym projekcie.

c) Testy akceptacyjne (E2E) znajdują się na najwyższym poziomie w piramidzie. Taka ilustracja sugeruje nam, że ilość tych testów powinna być ograniczona i bardzo dobrze przemyślana (głównie są to testy regresji). Koszt wytworzenia oraz utrzymania takich testów jest na pewno większy, niż w przypadku testów z poprzednich poziomów. Wykonują się też one o wiele dłużej, niż testy na niższych poziomach. Cel tych testów jest skierowany na naśladowanie zachowania użytkownika końcowego. Dzieje się tak zarówno w przypadku testów manualnych jak i automatycznych. Przewagą tego rodzaju testów and niższymi poziomami jest **duża wartość biznesowa**. Dlatego muszą one istnieć w procesie wytwarzania oprogramowania, ponieważ dają informację o tym, że użytkownik końcowy będzie mógł korzystać z aplikacji zgodnie z jej przeznaczeniem.

Istnieje wiele dobrych praktyk dotyczących pisania kodu aplikacji, jak i testów automatycznych. Istnieje również koncepcja, która oznaczana jest jako **FIRST.** Każda litera tej koncepcji to nazwa pewnej dobrej praktyki:

- ✓ **F**ast
- ✓ **I**solated (Independent)
- ✓ **R**epeatable
- ✓ **S**elf-validating

- ✓ **T**imely

*Fast* oznacza, że uruchomienie testów powinno być szybkie, a dzięki temu informacja zwrotna o wyniku testów jest również przekazana w szybkim czasie. Szybkie testy zachęcają twórców oprogramowania do tworzenia oraz utrzymywania testów. Przyczyną długo wykonujących się testów mogą być np.:

a) Interakcje wejście/wyjście,
b) Opóźnienia w aplikacji,
c) Słaba wydajność aplikacji,
d) Brak stosowania dobrych praktyk.

Rozwiązaniem wyżej wymienionych problemów może być np. stosowanie zaślepek (mocków). Można też stosować koncepcję **in-memory.** Koncepcja ta zakłada, że część danych oraz część obliczeń przechowuje lub przetwarza się w pamięci komputera. Są jednak tacy, którzy twierdzą, że od tej koncepcji powinno się odchodzić (co wcale nie musi mieć miejsca w każdej sytuacji), a zamiast niej stosować konteneryzację środowisk czy baz danych, co obecnie jest bardziej popularne.

*Isolated* oznacza, że testy nie powinny zależeć od siebie. Dzięki temu testy można uruchamiać osobno w całym zestawie i zawsze dają te same wyniki. Jeżeli istnieje test, który oczekuje jakiegoś stanu przed jego uruchomieniem, to ten stan powinien być utworzony w izolacji. W podejściu BDD taki stan powinien być utworzony w fazie *given,* o ile to możliwe.

*Repeatable* oznacza, że testy powinny być powtarzalne w każdych warunkach. Wobec tego powinny one zależeć tyko od kodu, który testują. Jeśli zależą od czegokolwiek, to ich wynik nie jest jednoznaczny. W takim przypadku raz test zwróci pozytywny wynik, a raz negatywny.

*Self-validating* oznacza, że każdy test powinien dać klarowną informację o tym, czy przeszedł pozytywnie czy nie. Jeśli nie przeszedł pozytywnie, to powinien dać informację co się stało oraz dlaczego nie przeszedł. Bardzo ważne jest w tym przypadku rozsądne nazywanie testów oraz umiejętne konstruowanie asercji.

*Timely* oznacza, żeby testy były pisane razem z nowym kodem, czyli aby były dostarczone na czas. Niedopuszczalna jest sytuacja, że testy powstają „po jakimś czasie". Mogą powstać po implementacji kodu, natomiast powinny trafić do wspólnego repozytorium w ramach tego samego zestawu zmian. Istnieją koncepcje, które wspomagają zasadę *timely* np. TDD (Test-Driven Development), gdzie najpierw powstają testy, a następnie właściwa implementacja kodu tworzonej aplikacji.

Zasada FIRST oraz piramida testów to niezależne koncepcje. Obie te praktyki mogą się łączyć w procesie testowania. Piramidę testów można potraktować jako jeden zbiór testów w sensie kolektywnym. Bez zastosowania zasady FIRST nie będzie możliwe utworzenie takiej piramidy testów, jaka opisana jest wcześniej. Problem może się również pojawić, jeżeli testy akceptacyjne **będą dodawane bez stosowania dobrych praktyk**. Problem, jaki może pojawić się podczas implementacji zasady FIRST w poszczególnych poziomach piramidy, przedstawia niżej umieszczona tabelka:

[tabela]

| | Fast | Isolated | Repeatable | Self-validating | Timely |
|---|---|---|---|---|---|
| Testy jednostkowe | OK | OK | OK | OK | OK |
| Testy integracyjne | OK | OK | OK | OK | OK |
| Testy akceptacyjne | ??? | OK | OK | OK | ??? |

Istnieje jednak kilka sposobów, aby testy akceptacyjne były wydajne oraz spełniały potrzeby testowe twórców oprogramowania, m.in. zasadę FIRST:

a) Wybrane wartości brzegowe oraz nietypowe przypadki (ale nie wszystkie) można testować na niższych poziomach.
b) Testy mogą być uruchamiane równolegle, ale muszą spełniać zasadę FIRST.
c) Mając już wstępne makiety GUI, można projektować testy E2E.
d) W projektach, które są nietypowe z dowolnego powodu, można zidentyfikować wszelkie osobliwe ich zachowania oraz opracować strategie testowe.
e) Backend, który zawiera dane niezbędne do przetestowania konkretnej funkcjonalności, może zostać zamockowany poprzez wstrzyknięcie danych za pomocą odpowiednich komend javascriptowych. Dzięki temu można skupić się na testowaniu konkretnej zmienionej lub nowej funkcjonalności bez potrzeby przechodzenia poprzez formularze, których tak naprawdę nie trzeba sprawdzać w danym momencie.

Podsumowując, możliwe jest łączenie piramidy testów oraz zasady FIRST, ale wymaga to znajomości pewnych procesów oraz zaangażowania.

## 3. Piramida testów, FIRST a ciągła integracja.

Ciągła integracja (ang. continuous integration), ciągłe dostarczanie to bardzo popularny termin. Zrozumienie oraz prawidłowe wdrożenie ciągłej integracji w całej organizacji dostarcza wymiernych korzyści wszystkim interesariuszom. Ciągła integracja (CI) to koncepcja, która rozwiązuje problem budowania, testowania oraz integracji kodu. Pozwala ona upewnić się, czy zmiany w kodzie nie wprowadziły nowych defektów albo regresu, ponieważ CI uruchamia wszystkie testy w tym akceptacyjne. Zasadniczym elementem tej koncepcji jest współdzielone repozytorium kodu źródłowego, do którego dostęp powinna mieć każda osoba pisząca kod. Istotnym elementem ciągłej integracji jest serwer ciągłej integracji. Dzięki niemu można bez udziału człowieka sprawdzić, czy każda kolejna zmiana dokonana przez osobę piszącą kod nie wprowadziła defektów. W momencie oddawania zmian do repozytorium kodu serwer ciągłej integracji samoczynnie wykrywa i pobiera tę zmianę, następnie buduje aplikację oraz uruchamia testy. Sytuacja, w której co najmniej jeden test się nie powiedzie, wymusza na zespole wprowadzenie poprawki, zanim kod zostanie ponownie oddany do repozytorium.

Koncepcja ciągłej integracji wnosi wiele pozytywnych aspektów w procesie produkcji oprogramowania. Jednym z założeń CI jest to, żeby w jak najkrótszym czasie otrzymać informację zwrotną o wynikach testów. Pojęcie *"w jak najkrótszym czasie"* to wielkość względna. W miarę rozwoju oprogramowania oraz automatyzacji testowania czas wykonania testów wzrasta. Nie można dopuścić do sytuacji, że ten czas jest zbyt długi, ponieważ niesie to potencjalną nieefektywność. Im szybciej zespół otrzyma informacje o wprowadzonym defekcie, tym szybciej defekt ten zostanie wzięty pod uwagę, zanim kod zostanie oddany do repozytorium. W tym momencie pojawia się przestrzeń na to, aby wspólnie z CI zastosować zasadę FIRST oraz piramidę testów. **Piramida testów oraz CI wspierają szybkie wykonanie testów** w procesie ciągłej integracji. Dzięki temu zespół może od razu zareagować w przypadku pojawienia się defektu w kodzie oprogramowania, nawet kilka razy dziennie.

## 4. Podsumowanie

Celem napisania artykułu było przedstawienie czytelnikowi korzyści, jakie mogą płynąć z łączenia dobrych praktyk wytwarzania testów oraz tworzenia samego oprogramowania. W wyżej przedstawionych rozdziałach strategia ta dotyczy łączenia zasady FIRST, piramidy testów oraz ciągłej integracji.

Istnieje również wiele innych dobrych oraz sprawdzonych praktyk programistycznych, które można łączyć w jednym repozytorium. Warto korzystać z takiej możliwości, ponieważ zawsze daje to wymierne korzyści oraz zapewnia jakość dostarczanego oprogramowania.

***Autor:***

Marek Żukowicz jest absolwentem matematyki na Uniwersytecie Rzeszowskim. Obecnie pracuje jako Test Engineer w Aptitude Software. Jego zainteresowania skupiają się wokół testowania oraz jakości oprogramowania, procesów oraz koncepcji związanych z wytwarzaniem oprogramowania, zastosowania algorytmów ewolucyjnych oraz zastosowania matematyki w procesie testowania. Interesuje się również muzyką.