Wstęp do TDD (ang. *Test Driven Development*)

Jak otrzymać „przejrzysty kod, który działa”? Wiele złych mocy czyha na programistów, aby zniweczyć ich wysiłki zmierzające do uzyskania przejrzystości kodu i zapewnienia poprawnego działania (czy nawet działania w ogóle). I, skoro nie mamy żadnych zdolności pozwalających złagodzić nasze obawy, pozostaje nam programowanie intensywnie wspierane przez zautomatyzowane testy, czy może inaczej, **programowanie sterowane testami** — „sterowane” w sensie dosłownym, bo w istocie oparte na dwóch zasadach:

- pisaniu nowego kodu tylko wtedy, gdy istniejący kod nie przechodzi pomyślnie zautomatyzowanych testów,
- eliminowaniu duplikacji.

Te banalne (wydawałoby się) zasady mają jednak niebanalne konsekwencje pod względem zachowań zarówno indywidualnych, jak i grupowych, a także pod względem implikacji technicznych. Otóż, kiedy chcemy pozostawać w zgodzie z nimi, zmuszeni jesteśmy do:

- projektowania w sposób organiczny, czyli poprzez uruchamianie kodu zapewniającego sprzężenie zwrotne między podejmowanymi decyzjami;
- tworzenia własnych testów — nierozsądnie byłoby oczekiwać, dwadzieścia i więcej razy dziennie, aż niezbędne testy napisze ktoś inny;
- programowania w taki sposób, by w używanym środowisku programistycznym nawet małe zmiany skutkowały wyraźnymi reakcjami;
- projektowania aplikacji jako zespołu luźno powiązanych, wysoce spójnych wewnętrznie komponentów, bo aplikacje takie testuje się wyraźnie łatwiej i efektywniej.

W praktyce przekłada się to na następujący (banalny czy nie — kwestia gustu) scenariusz postępowania projektanta-programisty.

1. Napisany właśnie test wykazuje **niepoprawność** aplikacji, co sygnalizowane jest przez znany “czerwony pasek” postępu w oknie testowania (być może sam test nie może zostać skompilowany i uruchomiony, bo zawiera formalne błędy).
2. Do kodu testowanej aplikacji wprowadzane zostają zmiany, **z jedną i tylko jedną** intencją: spowodowania, by wspomniany test został przez tę aplikację zaliczony, co objawi się w postaci “zielonego paska”. Inne aspekty wprowadzanych zmian — między innymi ich sensowność z perspektywy logiki aplikacji — mają w tym momencie znaczenie drugorzędne, nawet jeśli przez purystów uważane jest to za ciężki grzech.
3. Eliminujemy z kodu aplikacji wszelkie duplikacje w stosunku do kodu testów; tę czynność nazywa się **refaktoringiem** lub **refaktoryzacją**.

I tak oto rytm pracy projektanta programisty zaczyna przypominać trans, w którym nieustannie przewija się mantra „czerwony-zielony-refaktoring”.

Opisane postępowanie przyczynia się do zredukowania czegoś, co można by nazwać „nasyceniem defektami”, czyli do zmniejszania średniej liczby błędów objawiających się w jednostce czasu.

W konsekwencji możliwe staje się jasne sprecyzowanie celu: tworzenie **wyłącznie** takiego kodu, który konieczny jest do zaliczenia załamujących się aktualnie testów. Ten stricte techniczny imperatyw pociąga za sobą niebagatelne konsekwencje o charakterze społecznym, ponieważ:

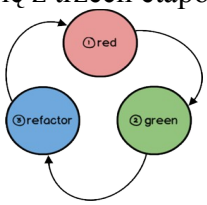
- niski wskaźnik nasycenia defektami pozwala ekipie odpowiedzialnej za zapewnienie jakości (QA) działać w sposób raczej proaktywny niż reaktywny;
- zredukowanie liczby przykrych niespodzianek pozwala menedżerom projektów na lepszą organizację dziennych harmonogramów zaangażowania programistów;
- znaczące uproszczenie języka technicznych konwersacji pozwala programistom na bardziej intensywną interakcję — minuta po minucie, zamiast dzień po dniu czy tydzień

po tygodniu.

- i ponownie: dzięki niskiemu wskaźnikowi nasycenia defektami, codziennie otrzymujemy oprogramowania wzbogacone o nowe funkcje i nadające się do zaoferowania klientom — tym obecnym i tym nowym.

Red-Green-Refactor

Kluczowym aspektem TDD jest cykl pisania testów. Najpierw piszemy testy, następnie implementujemy funkcjonalność, a na końcu refaktoruzyjemy. Cykl nazywany jest najczęściej Red-Green-Refactor lub TDD Mantra, składa się z trzech etapów, które jako całość są powtarzane:



1.**Red**: Piszemy test, który się **nie** powodzi.

a.Testy piszemy do pustych, ale istniejących już klas i metod.

b.Uruchamiamy test i oczekujemy, że się nie powiedzie.

2.**Green**: Piszemy kod, aby testy się powiodły.

a.Implementujemy kod.

b.Uruchamiamy testy. Wszystkie testy muszą się powieść.

3.**Refactor**: Refaktoryzacja kodu — wprowadzenie zmian, które poprawiają jakość kodu (np. usunięcie duplikacji), ale nie zmieniają jego funkcjonalności.

a.Po refaktoryzacji, uruchamiamy wszystkie testy by sprawdzić czy czegoś nie zepsuliśmy.

b.Ten punkt jest często lekceważony lub pomijany w procesie. Nie zapominajmy o tym, równie ważnym co dwa poprzednie, elemencie.

Cztery główne rodzaje testów to:

•testy jednostkowe (*unit tests*) — testujemy pojedynczą, jednostkową część kodu: zazwyczaj klasę lub metodę;

•testy integracyjne (*integration tests*) — testujemy kilka komponentów systemu jednocześnie;

•testy regresyjne (*regression tests*) — po wprowadzeniu naszej zmiany uruchamiane są wszystkie testy w danej domenie biznesowej celem sprawdzenia czy zmiana nie spowodowała błędu w innej części systemu;

•testy akceptacyjne (*acceptance tests*) — testy mające na celu odpowiedzieć na pytanie czy aplikacja spełnia wymagania biznesowe.

TDD oparte jest tylko i wyłącznie o testy jednostkowe, tj. piszemy jednostkowe testy przed implementacją kodu (Red-Green-Refactor), badamy pokrycie testów jednostkowych, uruchamiamy

przy domyślnym buildzie testy jednostkowe. Testy integracyjne mogą być pisane opcjonalnie. Podstawową różnicą między obydwoma rodzajami testów jest to, że **testy jednostkowe testują pojedyncza część kodu**, natomiast **testy integracyjne mają na celu sprawdzenie kilka komponentów działających razem**.

Przykładowy test jednostkowy z fazami Red-Green-Refactor opisany jest pod adresem: http://dariuszwozniak.net/2013/06/30/kurs-tdd-czesc-4-nasz-pierwszy-test-jednostkowy/

Wady i zalety TDD

Zalety TDD:

•Dokładne zrozumienie wymagań dokumentacji. Testy piszemy zawsze względem dokumentacji.

•Testy jako dokumentacja jest zawsze aktualna w czasie.

•Testy nie wprowadzają niejednoznaczności, cechy którą może posiadać dokumentacja papierowa.

•Wymuszanie dobrego designu kodu i szybka identyfikacja potencjalnych błędów w designie, np. problem z zależnościami.

•Lepsza zarządzalność kodu w czasie.

•Łatwiejsze i bezpieczniejsze łatanie kodu.

•Natychmiastowy i automatyczny *feedback* na temat błędu w kodzie.

•Testy regresyjne pozwalają stwierdzić czy po naszych zmianach nie zepsuliśmy przy okazji czegoś w innej części systemu.

•Krótszy, całkowity, czas procesu developmentu.

•Dużo mniej ręcznego debugowania.

Wady TDD:

•Czas i wysiłek na trening i przygotowanie developerów.

•Potrzeba dyscypliny osobistej i zespołowej. Testy muszą być zarządzane i poprawiane w czasie w taki sam sposób jak cała reszta kodu.

•Początkowa percepcja dłuższego czasu developmentu.

•Nie wszyscy menadżerowie dają się przekonać. Biją argumentem dwukrotnie dłuższego developmentu, choć całkowity czas trwania developmentu (wliczając szukanie i naprawę błędów, nie tylko pisanie kodu) w TDD jest krótszy niż w nie-TDD.