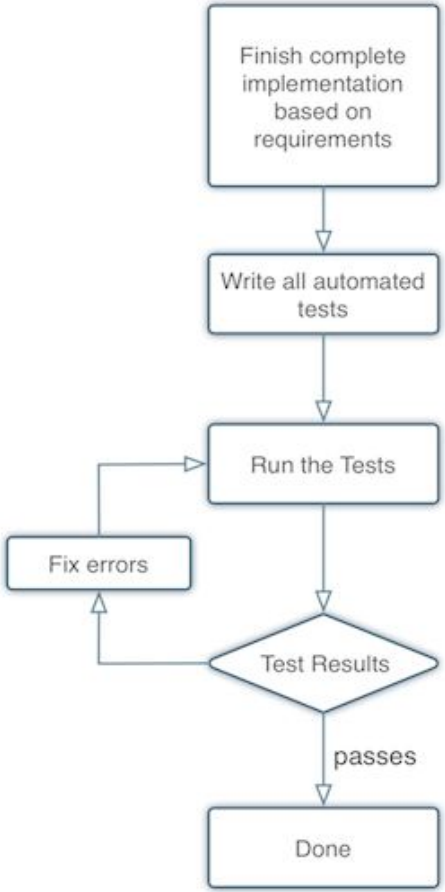
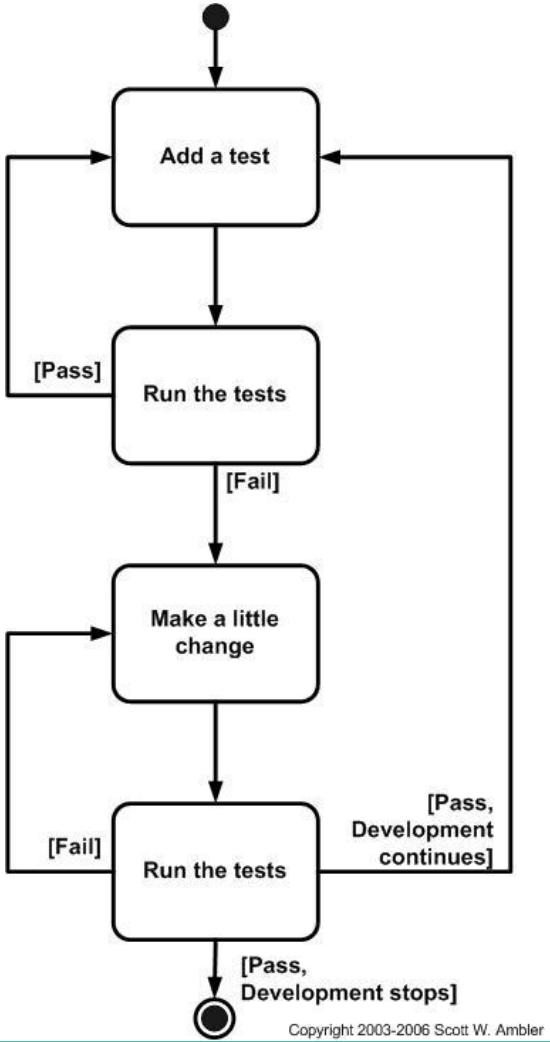
TDD

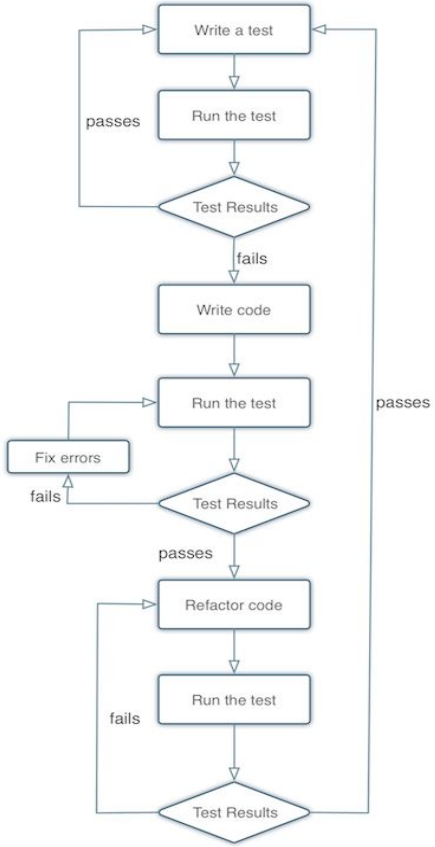
Test Driven Development



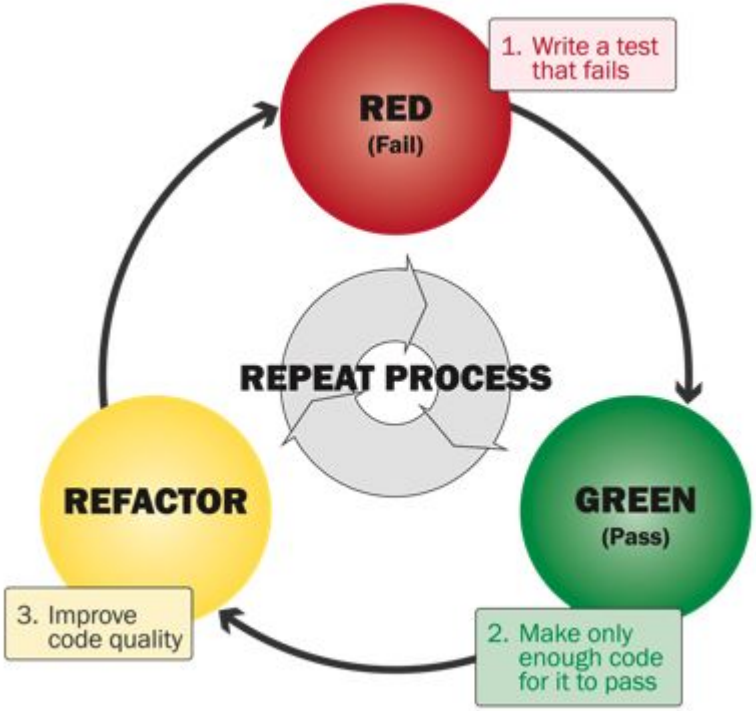
Traditional Cycle

Test First




Write a test
Run the test
passes
Test Results
fails
Write code
Run the test
passes
Fix errors
fails
Test Results
passes
Refactor code
Run the test
fails
Test Results

TDD cycle



Red -> Green -> Refactor

To jest właśnie TDD.

TDD to technika tworzenia oprogramowania sterowana przez testy, należąca do zwinnych metodyk tworzenia oprogramowania (?), polegająca na wielokrotnym stosowaniu Red - Green - Refactor.

1. Red - piszemy jeden prosty test nowej funkcjonalności
2. Green - tworzymy najprostszą implementację funkcjonalności tak, aby test przeszedł
3. Refactor - refaktoryzujemy kod, zwiększając jego jakość

Refaktoryzacja

Refaktoryzacja ma na celu zwiększenie jakości kodu poprzez nacisk na przejrzystość jak i niekiedy wydajność kodu, jednocześnie nie zmieniając samej funkcjonalności aplikacji. Jednoznaczność zachowania przed i po refaktoryzacji zapewniają nam właśnie testy, który powinny pokrywać kod naszego całego oprogramowania.

Historia

Historia prototypowej wersji TDD sięga lat 60, gdy komputery programowano jeszcze za pomocą kart dziurkowanych. Programiści mając ograniczoną ilość czasu przy maszynie, pierwsze zapisywali oczekiwany wynik, a następnie porównywali go z tym zwróconym przez komputer.

Za autora “ponownego odkrycia” TDD uważa się Kenta Becka, twórcę Extreme Programming i jednego z twórców Manifestu Agile.

Co robić, aby TDD spełniało swoje zadanie?

Uncle Bob

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Jakość testów

Jeśli testy nie weryfikują wszystkich przypadków zachowania funkcjonalności, to nie gwarantują nam one, że nasz kod ją poprawnie implementuje.

Dobre testy jednostkowe nie tylko powinny być napisane FIRST, powinne być napisane zgodnie z zasadami F.I.R.S.T.

F.I.R.S.T

F - Fast

I - Isolated/Independent

R - Repeatable

S - Self-Validating

T - Thorough/Timely

Struktura pojedynczego testu

A - Arrange (Given)

A - Act (When)

A - Assert (Then)

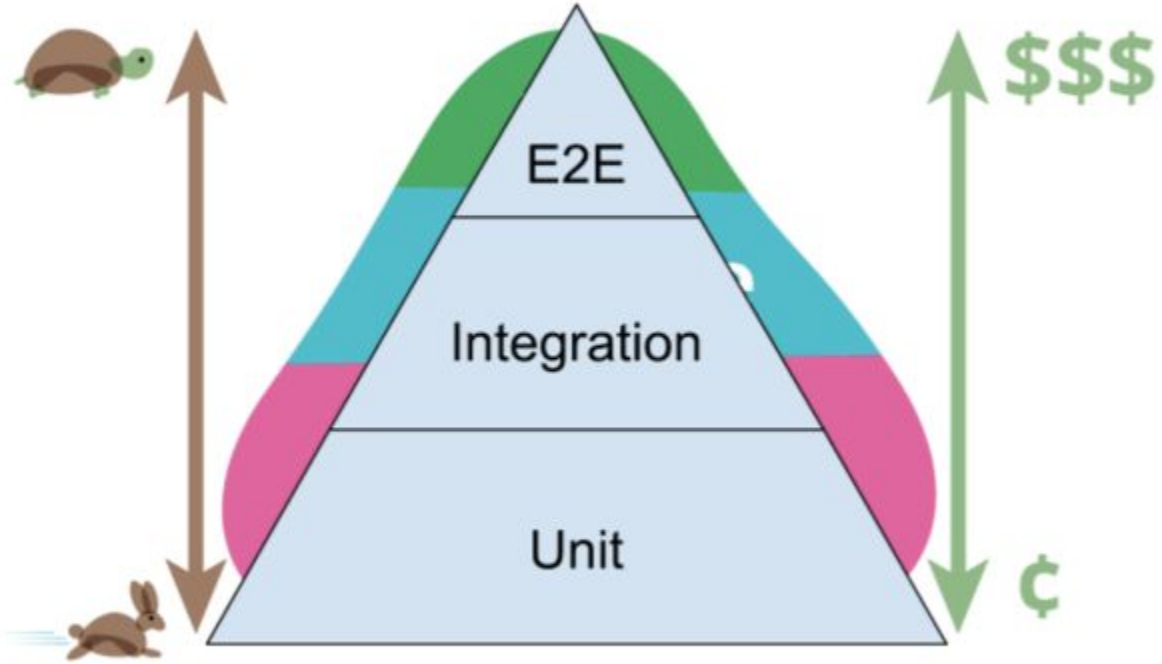
```php
public function testAddElement()
{
    // Arrange
    $element = 'some element';
    $collection = new Collection();

    // Act
    $collection->add($element);

    // Assert
    $this->assertEquals(1, count($collection));
}
```

Pokrycie testami

Do bezpiecznego pisania nowych funkcjonalności i refaktoryzacji kodu potrzebujemy odpowiedniego pokrycia kodu testami. Jedna niewinna zmiana w kodzie może uszkodzić istotną część aplikacji.


E2E
Integration
Unit
$$$
¢

Ustalenie spójnej konwencji

Zespół powinien ustalić jasne i spójne reguły organizacji testów, które będą implementowane przez narzędzia do testów automatycznych jak i przestrzegane przez programistów.

Wspólna konwencja zwiększa czytelność testów i ułatwia wprowadzanie nowych pracowników do projektu.

(wybieramy wzorce organizowania testów np.
Testcase Class per Class, Testcase Class per Feature ...)

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID’ny kod

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID’ny kod
- wolniejsze narastanie długu technicznego

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID’ny kod
- wolniejsze narastanie długu technicznego
- zawsze aktualna “dokumentacja”

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID’ny kod
- wolniejsze narastanie długu technicznego
- zawsze aktualna “dokumentacja”
- testowanie funkcjonalności bez uruchamiania całego oprogramowania

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID’ny kod
- wolniejsze narastanie długu technicznego
- zawsze aktualna “dokumentacja”
- testowanie funkcjonalności bez uruchamiania całego oprogramowania
- szybki feedback

Zalety TDD

- zwiększona szansa na wykrycie błędu w fazie developmentu
- testowalny, SOLID'ny kod
- wolniejsze narastanie długu technicznego
- zawsze aktualna "dokumentacja"
- testowanie funkcjonalności bez uruchamiania całego oprogramowania
- szybki feedback
- wydajniejsze debugowanie

Wady TDD

Z założeń TDD

Najpierw programista pisze automatyczny test sprawdzający dodawaną funkcjonalność.

Na początku zaczyna się od przypadku, który nie przechodzi testu – zapewnia to, że test na pewno działa i może wyłapać błędy.

Na tym etapie jedynym celem kodu jest przejście testu. Potencjalne niebezpieczeństwa:

- Programista nigdy nie wróci do kodu
- Programista przyzwyczai się do złych praktyk

Dłuższy czas wytwarzania aplikacji

Metodologia TDD wymaga dodatkowego czasu na stworzenie testów jednostkowych oraz wymaga czasu w na utrzymanie testów.

Pierwsza wada może być bardzo zniechęcająca dla kierownictwa, szczególnie w początkowym okresie korzystania z TDD, kiedy deweloper potrzebuje czasem **nawet do 30% więcej czasu na wykonanie tych samych zadań**. Z czasem jednak, po nabraniu płynności w pisaniu testów, ten dodatkowy czas zmniejsza się, dzięki czemu koszt stosowania tej metodologii maleje.

Czas poświęcany przez programistę

Druga wada wynika z faktu, że aby testy były użyteczne, muszą być **odpowiednio zarządzane i uaktualniane wraz ze zmianami logiki aplikacji**. Dopóki dodajemy nową funkcjonalność, problem ten praktycznie nie istnieje, jednak przy wprowadzaniu zmian w istniejącej funkcjonalności należy pamiętać, że potrzebny jest czas na odpowiednie zmodyfikowanie istniejących testów jednostkowych.

Kiedy (nie) stosować TDD?

Chciałoby się powiedzieć – zawsze. Praktyka jednak pokazuje, że nie ma sensu korzystać z TDD w przypadku **krótkich, kilkulinijkowych projektów**. W takiej sytuacji nadmiarowość czasu jest po prostu zbyt duża. Nie ma również większego sensu korzystanie z tej metodologii w przypadku aplikacji (dalej, kilkulinijkowych), które nigdy nie będą rozwijane.

Kiedy (nie) stosować TDD?

Ponieważ generalną zasadą TDD jest pisanie testów jednostkowych przed tworzeniem właściwego kodu. Jak pokazuje praktyka, **nie ma większego sensu tworzenie testów jednostkowych do istniejącego kodu** (jakość takich testów zazwyczaj jest bardzo niska), zatem pozostaje w istniejących projektach tworzenie testów jednostkowych do nowej funkcjonalności oraz korzystanie z pełni dobrodziejstw Test Driven Development w nowych projektach.

Jak nie robić TDD

Źle napisany test, prowadzi do źle napisanej funkcji

Zbyt skomplikowane oprogramowanie, aby móc je przetestować

Testowanie szczegółów implementacji.

Powolne testy, sprawdzające wiele funkcjonalności

Zależności między przypadkami testowymi lub samymi testami

Testowanie dokładnego czasu wykonania lub wydajności wykonania.

Dalsze uwagi

Pamiętać należy, że to nie jest sztuka dla sztuki, TDD to tylko i wyłącznie narzędzie, które ma nam służyć. Piszemy testy do jasno sprecyzowanej funkcjonalności a nie same dla siebie – to nie jest wprawka w kodowaniu. To takie samo narzędzie jak debugowanie czy profilowanie.

Damir Tomicic

Apr 13, 2016 at 12:03 PM •

Talking about Agile .. here's the way to make your daily standup meeting much shorter. 😀