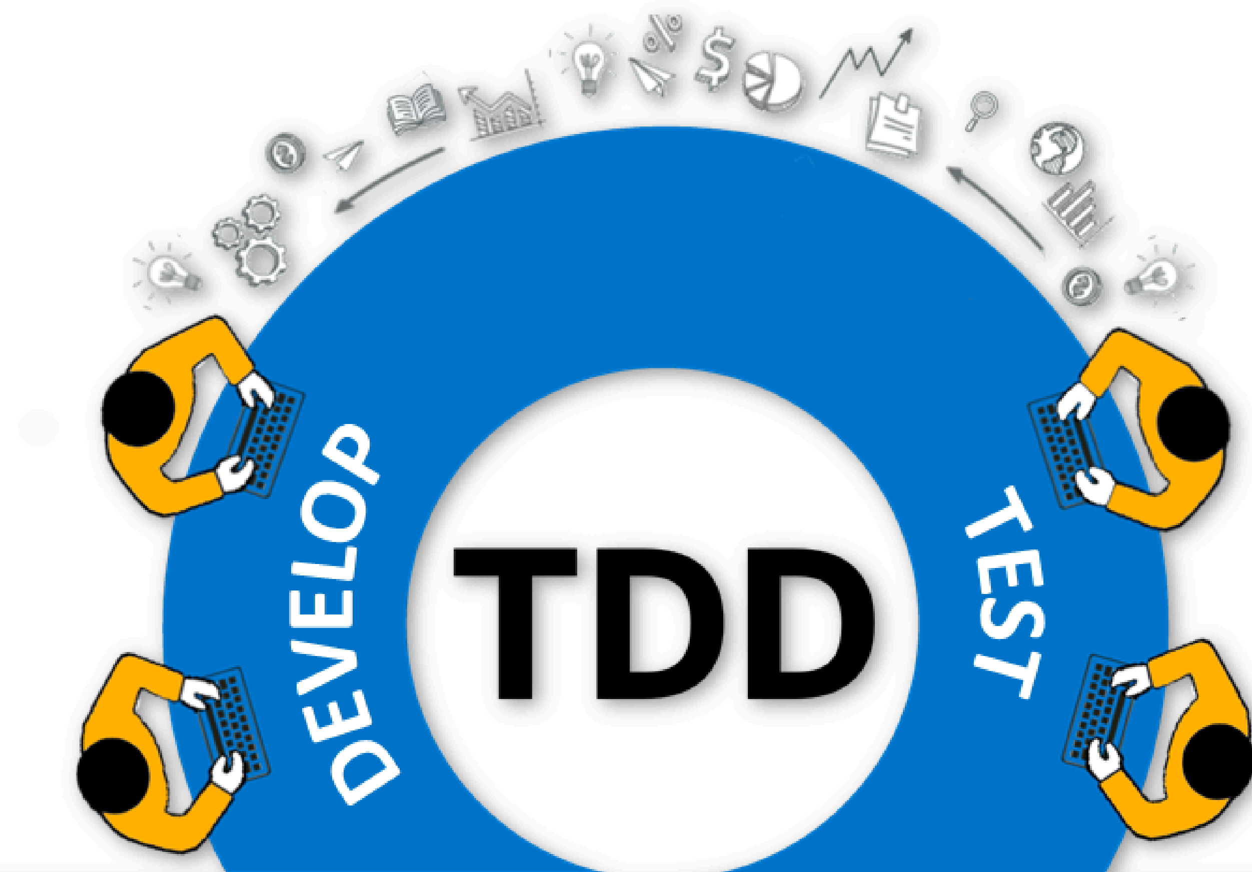
TEST-DRIVEN DEVELOPMENT

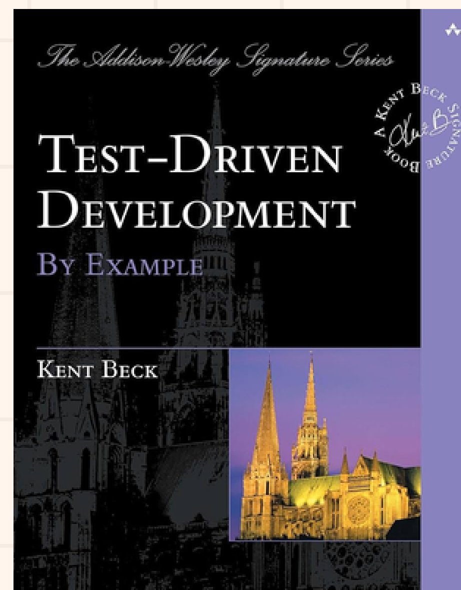


HISTORIA TEST-DRIVEN DEVELOPMENT

Technika **Test-Driven Development** (**TDD**) wywodzi się z praktyk związanych z **Extreme Programming**, które były popularyzowane na przełomie lat 90. XX wieku.

Początki **TDD** można powiązać z fundamentalnym założeniem, które jest kluczowe dla **Extreme Programming** – nacisk na krótkie cykle wytwarzania i ciągłe testowanie kodu.

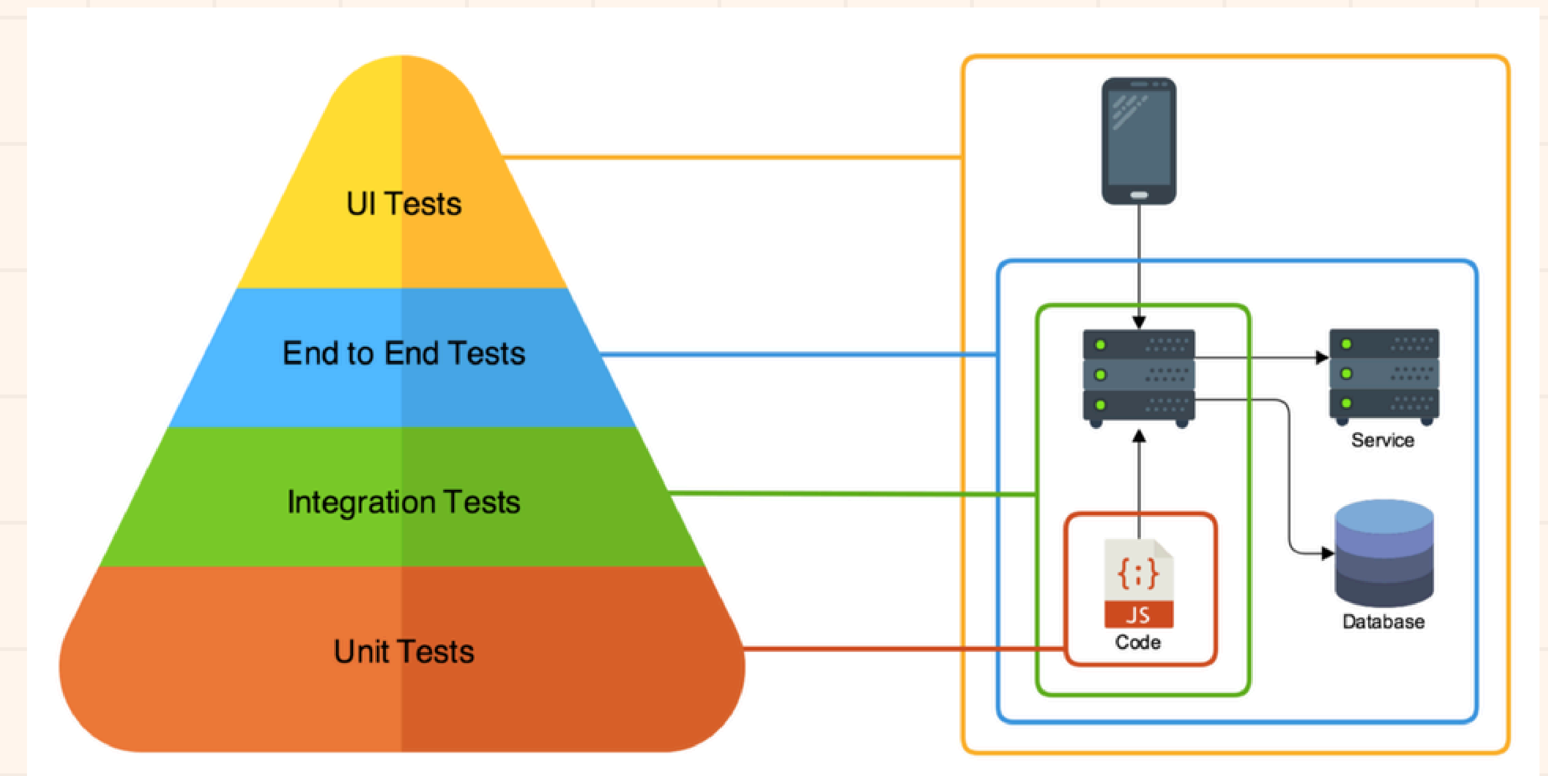
Chociaż koncepcja pisania testów przed kodem nie była zupełnie nowa, to właśnie Kent Beck sformalizował tę metodykę w ramach TDD. W 2003 roku Beck wydał książkę "Test-Driven Development: By Example", która szczegółowo opisywała proces TDD i dała programistom narzędzie do pracy w sposób bardziej zorganizowany i skuteczny.



Kent Beck
Jeden z głównych twórców techniki **TDD** oraz **Extreme Programming**

UNIT TESTY

Testy jednostkowe (ang. unit test) - metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym.



KIEDY TEST NIE JEST TESTEM JEDNOSTKOWYM?

A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

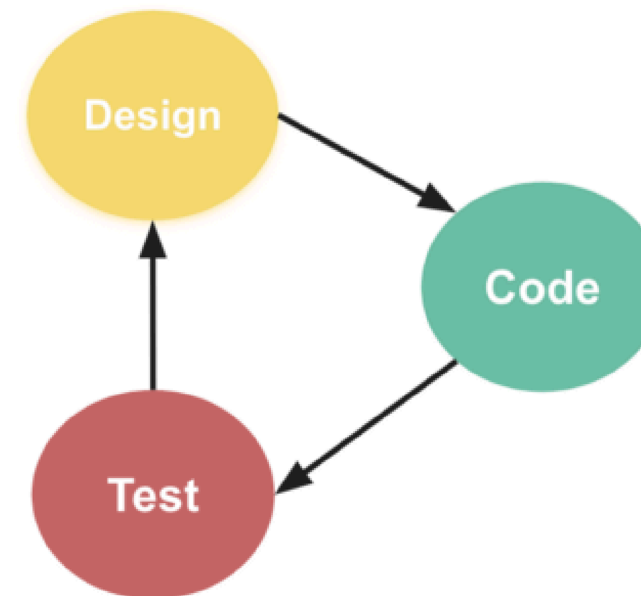
– Michael Feathers, A Set of Unit Testing Rules (2005)

CZYM JEST TDD?

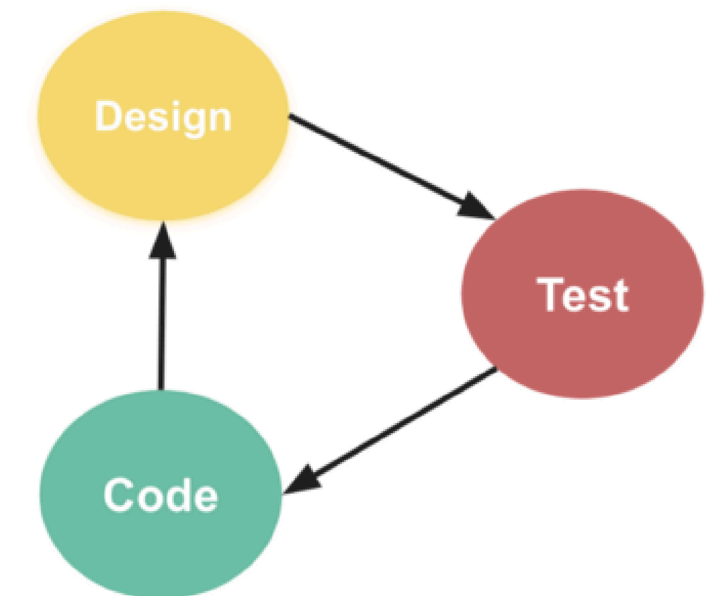
Test-Driven Development to technika tworzenia oprogramowania, w której pisanie testów poprzedza implementację właściwego kodu. Proces ten składa się z trzech głównych kroków:

1. **Napisz test**: Najpierw tworzy się test jednostkowy, który opisuje jedną funkcjonalność lub zachowanie kodu.
2. **Napisz kod**: Następnie tworzy się minimalny fragment kodu, który ma na celu sprawić, by test przeszedł pozytywnie.
3. **Refaktoryzuj**: Po uzyskaniu pozytywnego wyniku testu, można poprawić i udoskonalić kod, dbając o jego czytelność, strukturę i wydajność, przy jednoczesnym utrzymaniu pozytywnego wyniku wszystkich testów.

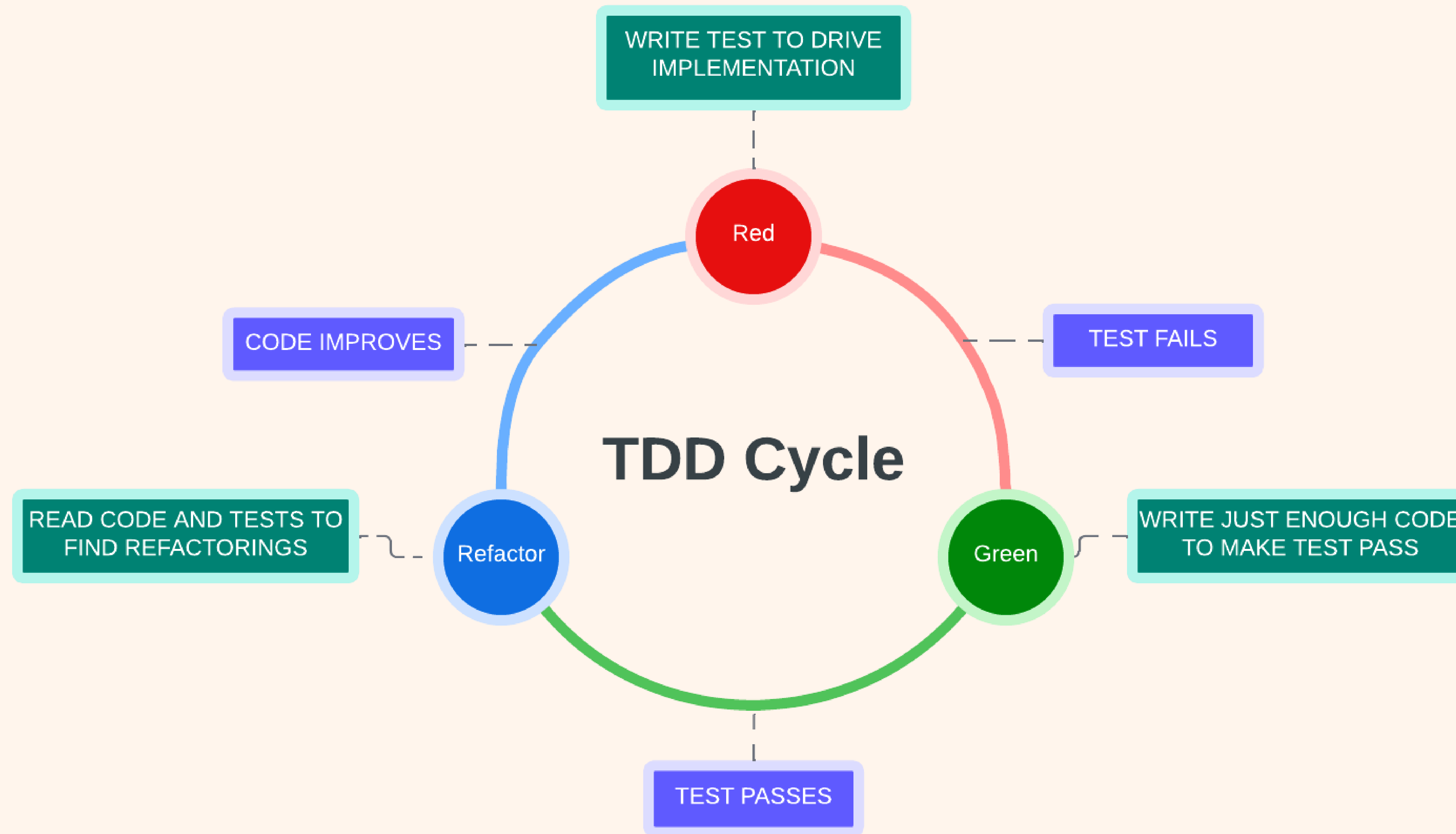
Old school approach



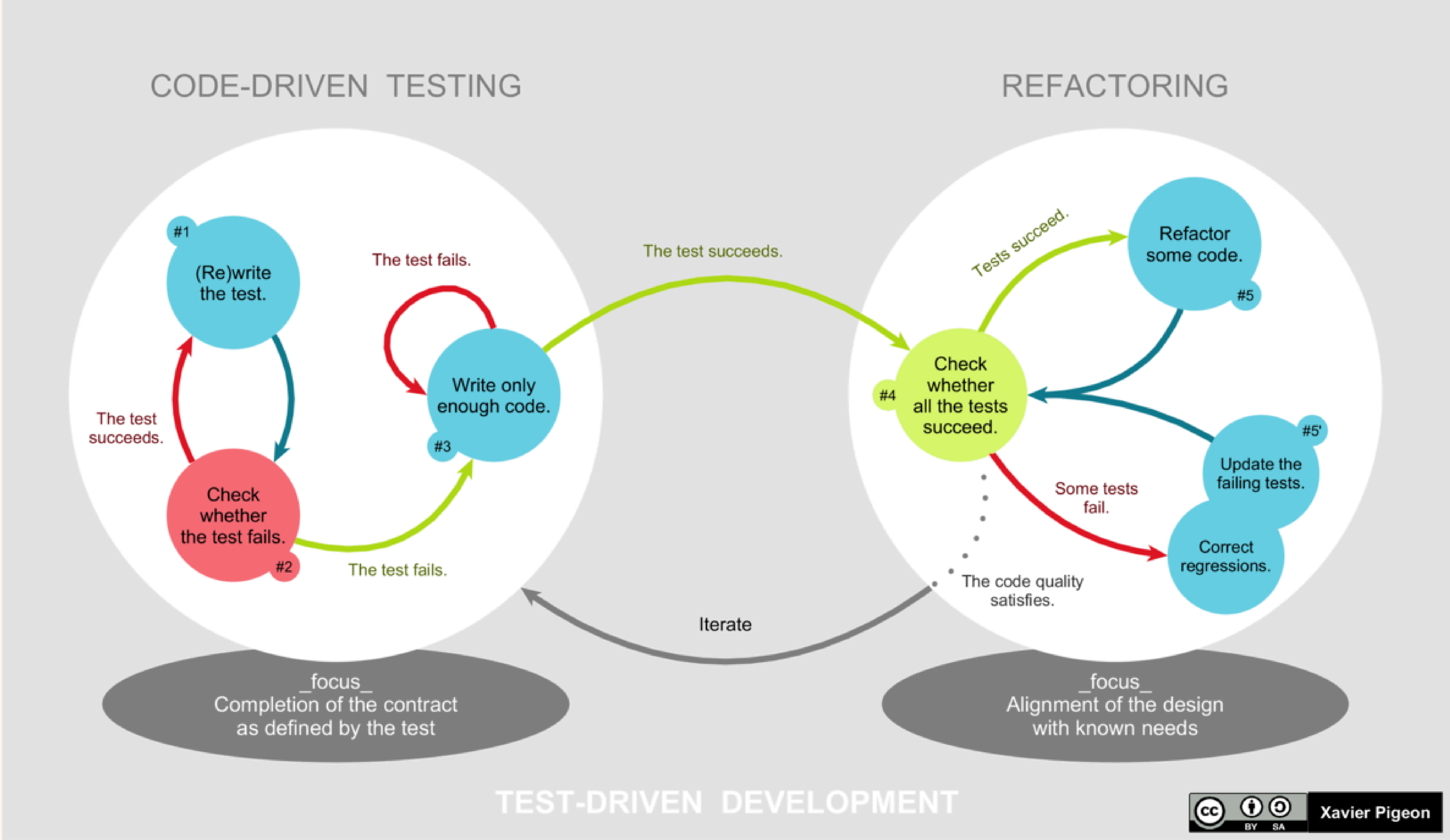
New approach



TDD TL;DR



TDD TL;DR PT.2



TRZY PRAWA TDD

1. NIE MOŻNA ZACZĄĆ PISAĆ KODU PRODUKCYJNEGO PRZED ZAKOŃCZENIEM PISANIA TESTU JEDNOSTKOWEGO, KTÓRY NIE JEST SPEŁNIONY.
2. KOD TESTU JEDNOSTKOWEGO POWINIEN BYĆ TYLKO TAK DŁUGI, ABY WYSTARCZYŁ DO NIESPEŁNIENIA TESTU, A BŁĘDNA KOMPILACJA JEST JEDNOCZEŚNIE NIEUDANYM TESTEM.
3. NIE MOŻNA PISAĆ WIĘKSZEJ ILOŚCI KODU PRODUKCYJNEGO, NIŻ JEST WYMAGANA DO PRZEJŚCIA TESTU JEDNOSTKOWEGO.

ZASADY TDD

K.I.S.S

KEEP.IT.SIMPLE, STUPID!

FAKE IT
TILL YOU
MAKE IT

< YAGNI >

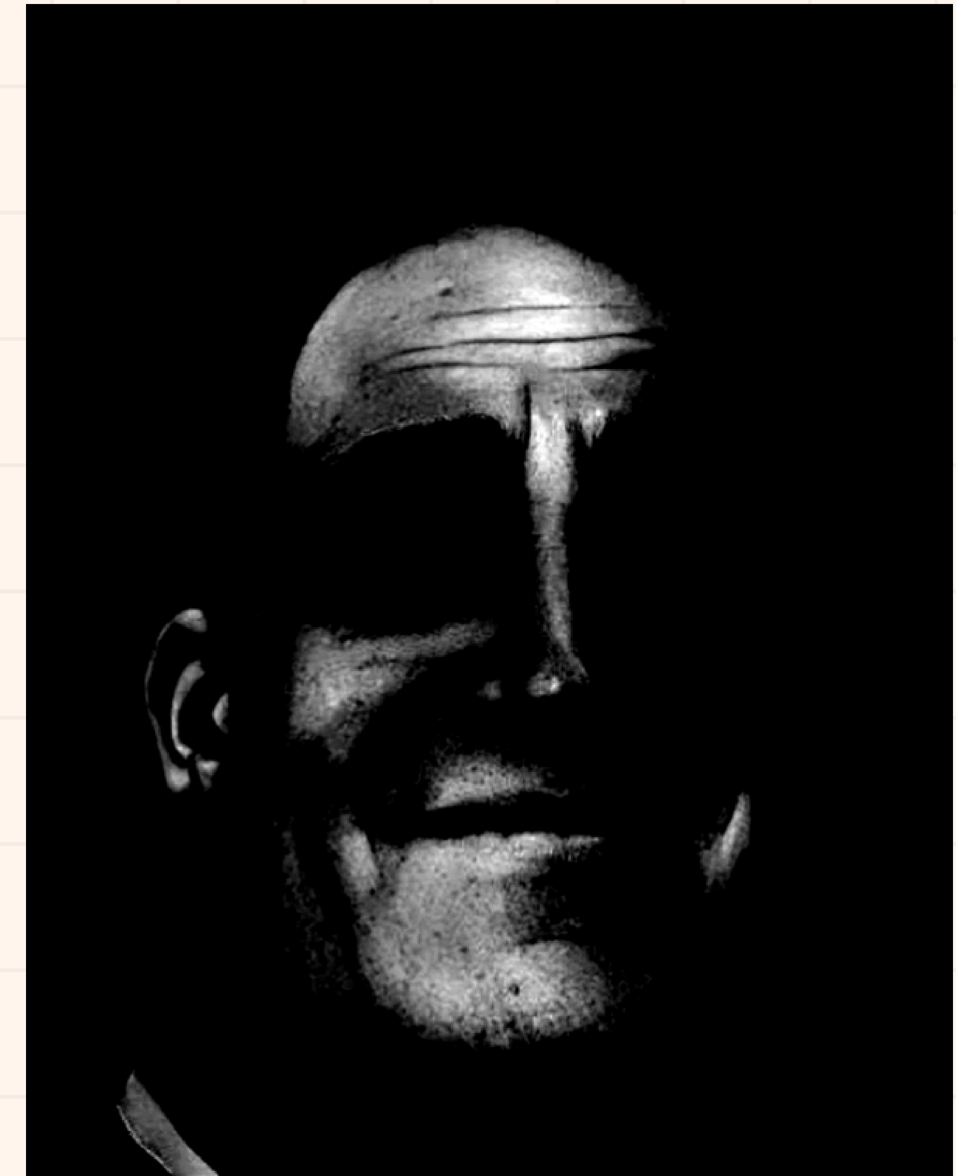
YOU AREN'T GONNA NEED IT

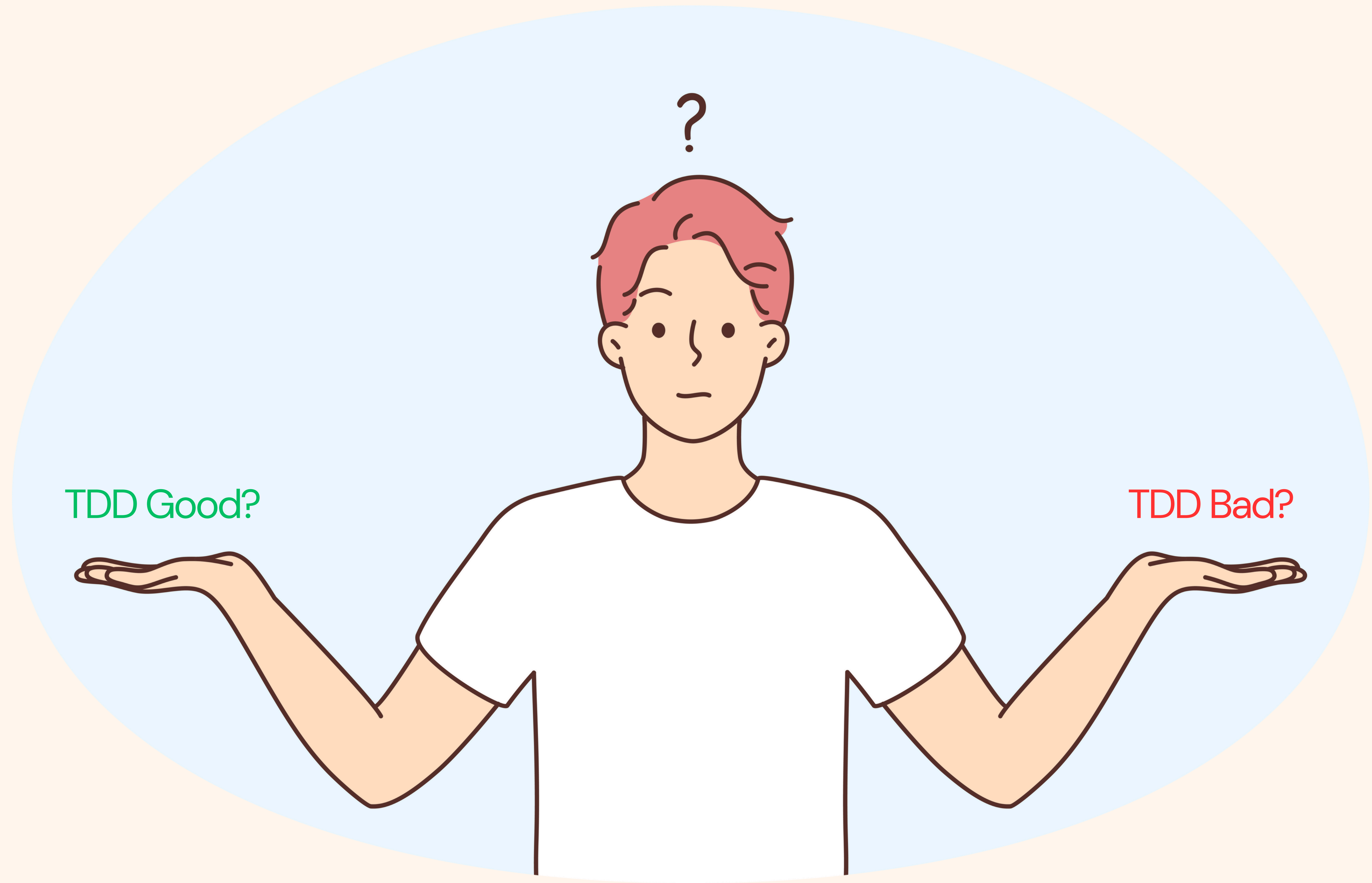
ZALETY TECHNIKI TDD

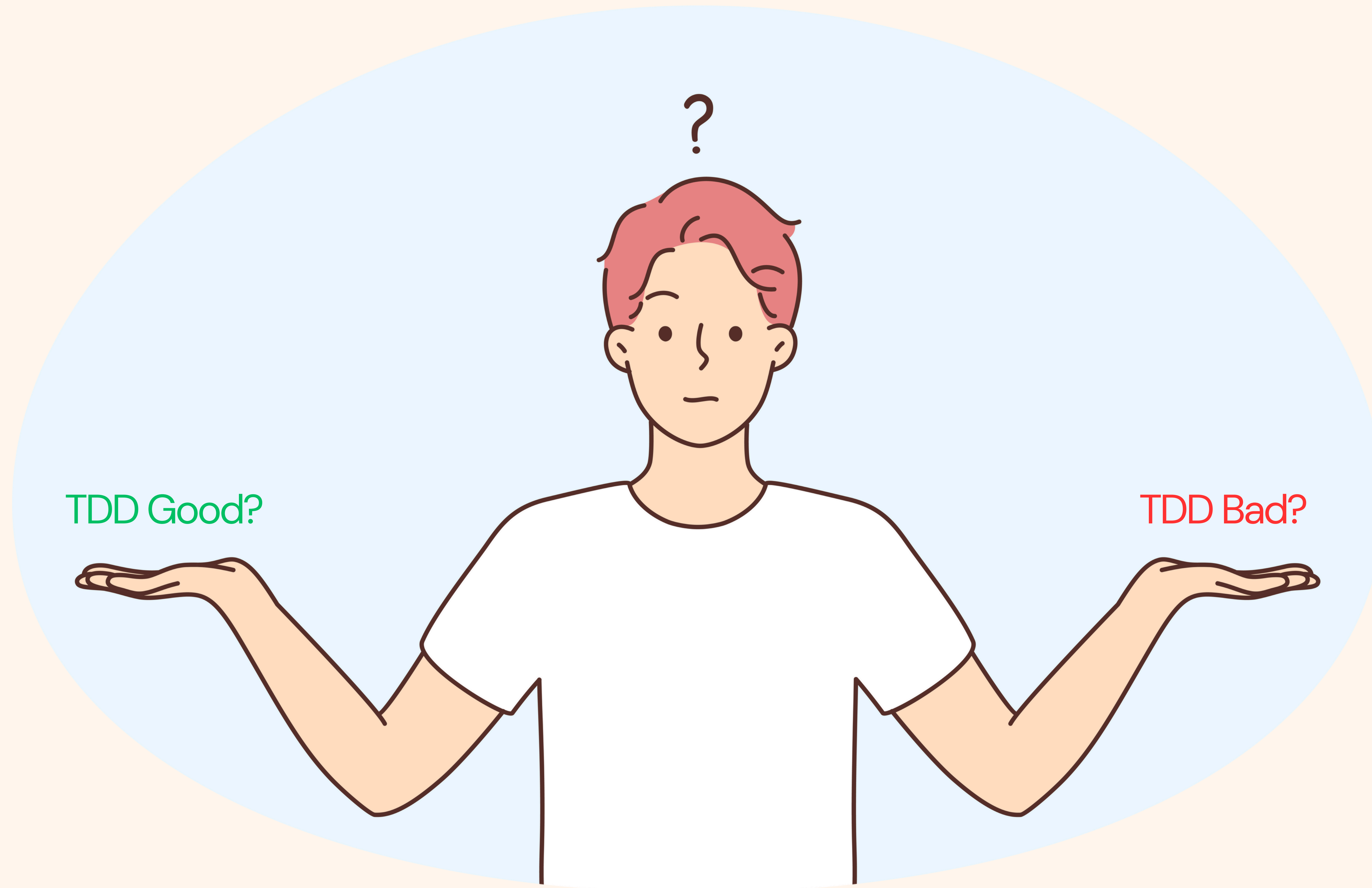
- Wczesne wykrywanie błędów
- Wyższa jakość kodu
- Ułatwiona refaktoryzacja
- Lepsza struktura i modularność kodu
- Wysokie pokrycie kodu testami
- Lepsze zrozumienie wymagań

WADY TECHNIKI TDD

- Większy nakład czasu na początku projektu
- Błędy mogą znaleźć się w testach, nie tylko w kodzie produkcyjnym
- Koszt utrzymania testów może być wysoki
- Możliwość napisania słabych testów
- Zwiększenie ilości kodu




?
TDD Good?
TDD Bad?

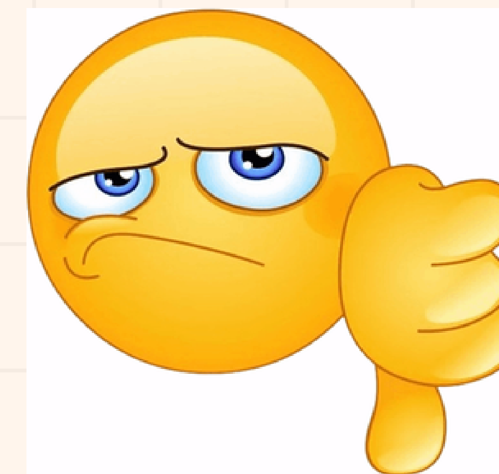


It depends...

KIEDY ZASTOSOWAĆ TDD?

- Kiedy rozpoczynasz nowy projekt, TDD może pomóc w określeniu wymagań i zapewnieniu, że kod będzie łatwy w utrzymaniu.
- W projektach z dużą ilością logiki biznesowej, TDD może pomóc w zrozumieniu i przetestowaniu złożonych interakcji między komponentami.
- Projekty, które są przewidywane do długotrwałego użytkowania, mogą korzystać z TDD, ponieważ dobrze napisane testy jednostkowe ułatwiają wprowadzanie zmian w przyszłości.

- W małych projektach, gdzie kod jest prosty i mało złożony, pisanie testów przed kodowaniem może być bardziej czasochłonne niż po prostu zaimplementowanie funkcjonalności.
- W projektach, które mają krótki czas życia, TDD może być zbędne. W takich sytuacjach bardziej opłacalne może być szybkie wprowadzenie kodu bez szczegółowych testów.
- W projektach, gdzie wymagania są niejasne lub mogą się zmieniać z dnia na dzień, TDD może stać się obciążeniem.



DOBRE PRAKTYKI

Pożądana struktura testu jednostkowego:

1. Wprowadź jednostkę testowaną (ang. Unit Under Test, UUT) w stan potrzebny do przeprowadzenia testu.
2. Wymuś na UUT wykonanie pożądanej czynności i przechwyć wynik/output.
3. Upewnij się, że rezultat testu jest taki, jaki oczekiwano.
4. Przywróć UUT do stanu sprzed testu, aby kolejne testy mogły wykonać się z tym samym stanem początkowym.

Nie testuj tylko pozytywnych scenariuszy.

Pisz małe testy, skoncentrowane na jednej “jednostce”.

Zapewnij, aby testy dawały powtarzalne rezultaty oraz żeby były niezależne (użyj “test doubles”)

ZŁE PRAKTYKI

Pisanie testów, które zależą na zmianach w poprzednich testach lub ich wynikach (kolejność wykonywania testów nie powinna mieć wpływu na ich rezultaty!).

Pisanie testów, które działają bardzo wolno.

Tworzenie testów, które sprawdzają dużo więcej niż jest to potrzebne.

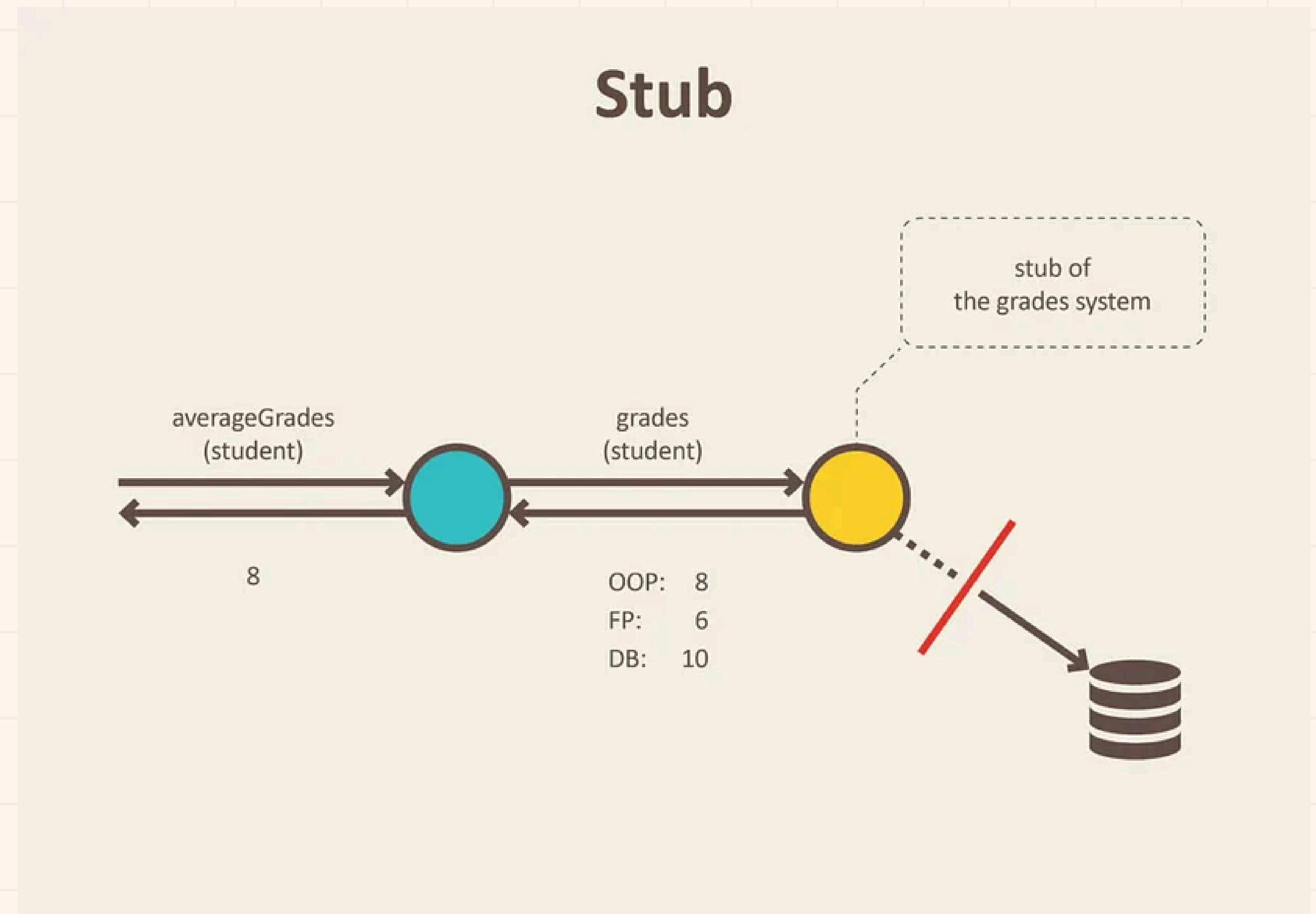
Testowanie bardzo precyzyjnych pomiarów czy zachowań.

TESTOWI “DUBLERZY”

CZYLI STUB’Y, FAKE’I I MOCKI

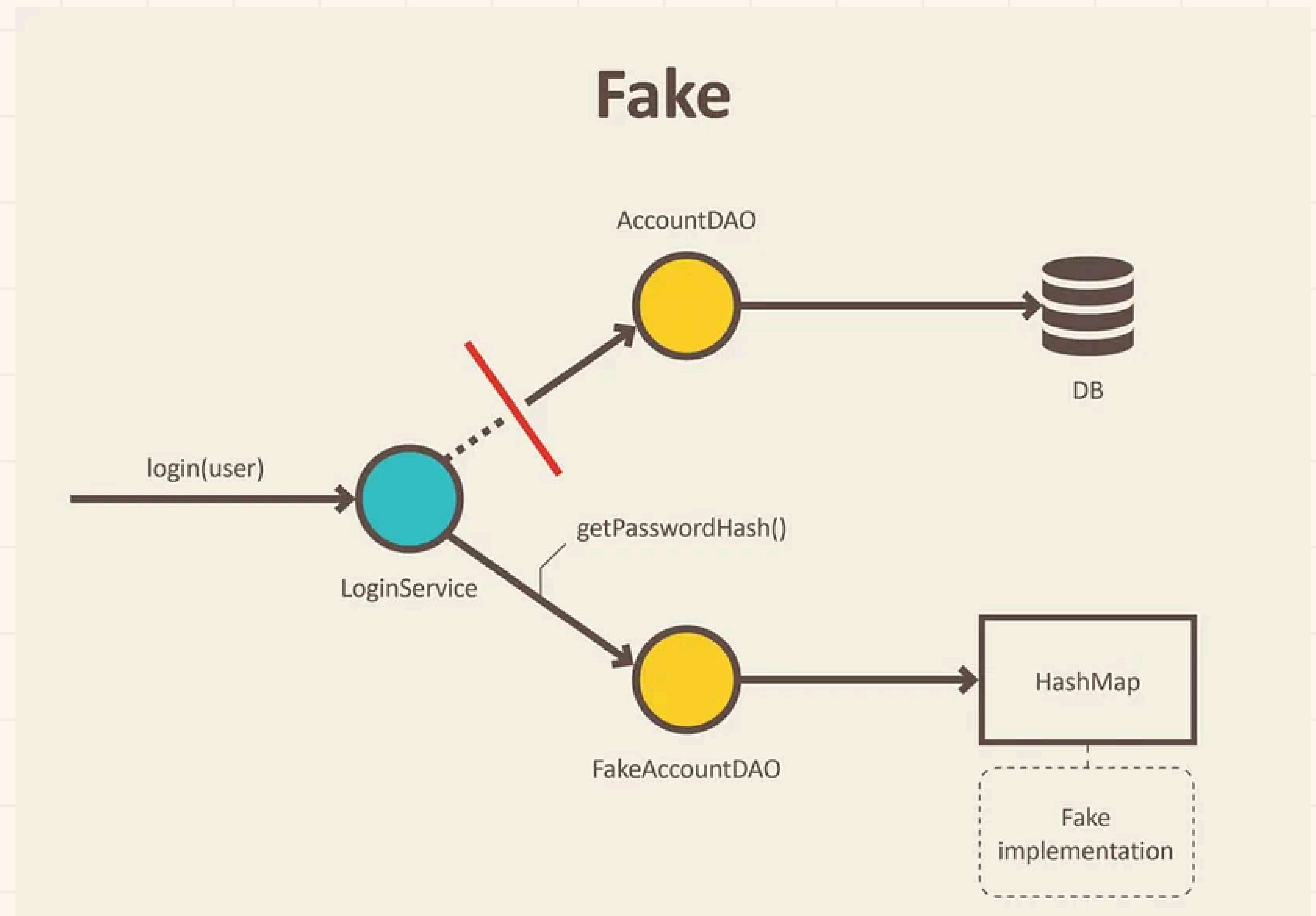
STUB

Stub to obiekt, który w testach służy do imitowania właściwej implementacji. Jego zadaniem jest wyłącznie zwrócenie zadanej wartości.



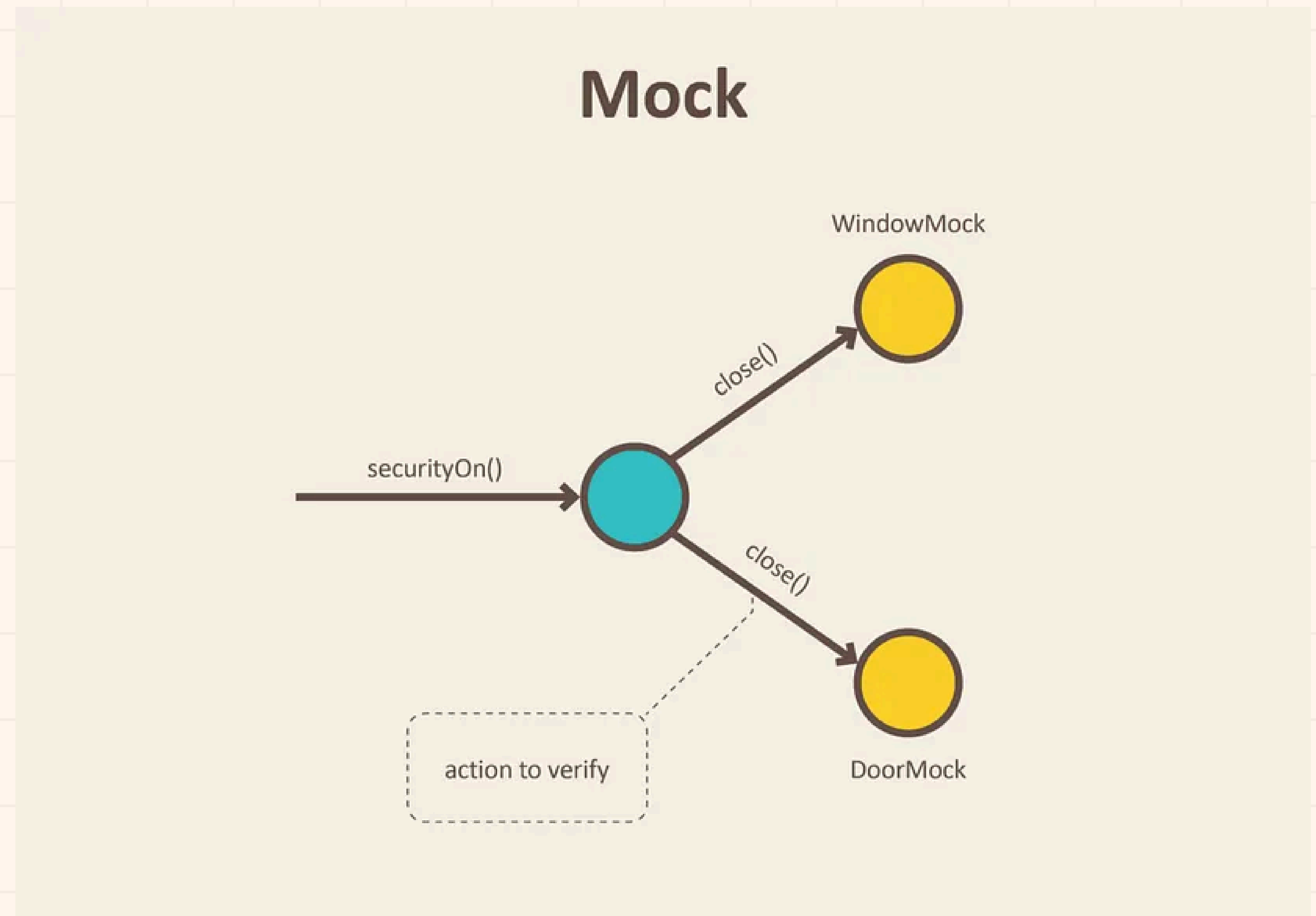
FAKE

Fake to obiekt posiadający działającą implementację, odbiegającą od kodu produkcyjnego. Zazwyczaj dostarcza uproszczoną wersję oryginalnych obiektów.



MOCK

Mock to obiekt, którego używa się zamiast rzeczywistej implementacji w trakcie testów jednostkowych. Pozwala on na określenie jakich interakcji spodziewamy się w trakcie testów.



POPULARNE NARZĘDZIA UŻYWANE W TDD

JUnit

EASYMOCK