➢ TDD - Wzorzec Red-Green-Refactor:

Główną osią wokół której obraca się TDD, jest postępowanie mogącę skrótowo być opisane jako Red-Green-Refactor.
Pracując nad kodem postępujemy według instrukcji:

1. Piszemy test dla nieistniejącej jeszcze funkcjonalności – test się załamie (**Red**).
2. Za wszelką cenę (nie bacząc na praktyki jeżeli rozwiązanie nie jest oczywiste), sprawiamy by test przeszedł (**Green**).
3. Poprawiamy kod, staramy się by nasz kod był piękny i czytelny – cały czas dbając o zielony kolor wyniku testu (**Refactor**).

Testy i kod produkcyjny są pisane razem, a testy są wykonywane kilka sekund wcześniej niż kod produkcyjny.

➢ F.I.R.S.T.

F. Fast szybkie

Gdy testy działają powoli, nie chcemy ich uruchamiać zbyt często.
Jeżeli nie uruchamiamy ich zbyt często, nie znajdziemy problemów wystarczająco szybko, aby można było łatwo je poprawić.

I. Independent niezależne

Jeden test nie powinien konfigurować warunków do następnego testu.
Powinniśmy być w stanie uruchamiać każdy test niezależnie i uruchamiać testy w dowolnie wybranym porządku.

Testy nie mogą mieć stanów początkowych, ani zasobów do wyczyszczenia.
Oznacza to także brak zależności w stosunku do zasobów zewnętrznych (baza danych, system plików, itd.)

Testy powinny być niezależne również od środowiska, na którym są uruchamiane.

R. Repeatable powtarzalne

Testy powinny być powtarzalne, czyli z każdym uruchomieniem zwracać ten sam rezultat. w każdym środowisku.

S. Self-Validating samokontrolujące się

Testy powinny mieć jeden parametr wyjściowy typu logicznego. Mogą one się powieść lub nie.
Nie powinniśmy musieć czytać plików dzienników w celu sprawdzenia, czy testy się powiodły.

T. Timely o czasie

Testy powinny być pisane bezpośrednio przed tworzeniem testowanego kodu produkcyjnego.
Jeżeli piszemy testy po kodzie produkcyjnym, może się okazać, że jest on trudny do przetestowania.

➢ Czystość testów

Jeżeli nie będziemy zachowywali czystości testów, utracimy je.
To właśnie testy jednostkowe zapewniają elastyczność, łatwość utrzymania i ponownego wykorzystania kodu.
Bez testów każda zmiana jest potencjalnym błędem.

Czyste testy to czytelne testy.

Wzorzec **Build-Operate-Check** lub **Arrange–Act–Assert** zakłada, że każdy test jest podzielony na trzy części. :

1. buduje dane testów,
2. operuje na danych testowych,
3. kontroluje, aby operacja dała oczekiwane wyniki.

Towarzyszy mu też konwencja ***given-when-then*** zakładająca, że każda z części jest umieszczano w osobnych metodach, których nazwy rozpoczynają się odpowiednio od tych słów. Pozwala to na odwoływanie się do nich z wielu testów.

➢ Dodatkowe zasady

● Jedna koncepcja na test

jedna koncepcja na test - niekoniecznie

Zasada jednej asercji jest dobrą wskazówką, lecz nie dogmatem.
Można jedynie stwierdzić, że liczba asercji powinna być zminimalizowana.

jedna koncepcja na test

Lepszą zasadą dotyczącą testów jest obejmowanie jednej koncepcji w każdej funkcji testowej.
Testowanie kilku koncepcji może powodować problem.

● **ZŁE** praktyki:

- **Testowanie prywatnych składowych**
- **Testy zawierające konfigurację**
- **Testy korzystające z konsoli systemowej**
- **Brak asercji**
- **Łapanie wszystkich wyjątków**
- **Oczekiwanie typu Exception**
- **Testowanie tylko ścieżki optymistycznej**
- **Stosowanie pętli w teście**
- **Instrukcje warunkowe w teście**

● Dobre praktyki:

- **Testuj warunki brzegowe i sytuacje wyjątkowe.**
  Załóżmy, że masz metodę, która przyjmuje tablicę, która musi mieć maksymalnie trzy elementy. Napisz kilka testów:
  - przekazując *null* zamiast tablicy,
  - przekazując pustą tablicę,
  - przekazując tablicę z trzema elementami,
  - przekazując tablicę z czterema elementami.

★JUnit

- **@Test**

Oznacza metodą testową. Metoda ta musi być **public void**.

- **Asercje**

Asercje w bibliotece JUnit to nic innego jak metody statyczne w klasie **Assert**.

Najczęściej stosowane asercje:

- **assertTrue**
- **assertFalse**

- **assertNull**
- **assertNotNull**

- **assertEquals**
- **assertNotEquals**

- **assertArrayEquals**

- **assertThrows**
- **assertDoesNotThrow**

- **assertSame**
- **assertNotSame**

- **fail**

- **Testowanie metod rzucających wyjątki**

```
@Test
public void shouldThrownIllegalArgumentExceptionOnWrongParameters() {
    assertThrows(IllegalArgumentException.class, new Range(20, 10));
}
```

- **Cykl życia klasy z testami jednostkowymi**

Adnotacje, które pozwalają na wykonanie fragmentów kodu przed/po testach:

@BeforeEach – metoda jest uruchamiana przed każdym testem jednostkowym, pozwala na przygotowanie instancji do testu.
@AfterEach – metoda jest uruchamiana po każdym teście jednostkowym, pozwala na „posprzątanie” po teście,
@BeforeClass – metoda statyczna jest uruchamiana jest raz przed uruchomieniem pierwszego testu z danej klasy.
@AfterClass – metoda statyczna jest uruchamiana jest raz po uruchomieniu wszystkich testów z danej klasy,