



Testowanie oprogramowania, TDD

Kamil Nowak

Plan na dziś



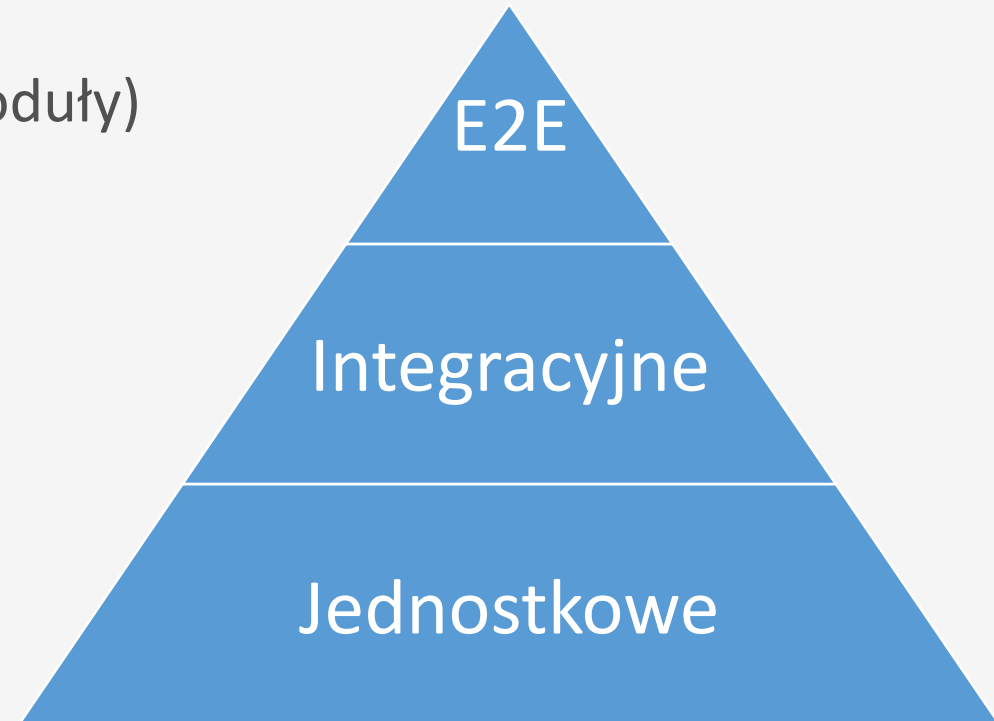
1. Testowanie oprogramowania
2. TDD.
3. Testowanie wyjątków.
4. Testowanie sparymetryzowane (DDT).
5. Mocki.

Testowanie oprogramowania.



Poziomy testów.

- Testowanie jednostkowe. (Tester testuje moduły)
- Testowanie integracyjne
- Testowanie systemowe. (E2E – End to end.)
- Testowanie akceptacyjne. (Klient)



Testowanie oprogramowania.



Testowanie jednostkowe.

Weryfikowanie poprawności działania pojedynczych elementów (jednostek) programu w izolacji od reszty systemu.

Jednostka: najmniejszy element programu, który będzie testowany w izolacji – niezależnie od innych. Jednostka to metoda, obiekt lub procedura.

Test jednostkowy to fragment kodu, który wywołuje inny fragment kodu i sprawdza poprawność pewnych założeń. Jeśli założenia okażą się błędne, test jednostkowy nie powiedzie się.



Testy jednostkowe.

- Weryfikacja funkcjonalności fragmentu kodu źródłowego – zwykle funkcji lub metody danej klasy
- Zwykle pisane przez programistów odpowiedzialnych za napisanie testowanego fragmentu kodu źródłowego – testowanie białoskrzynkowe
- Weryfikacja poprawności działania poszczególnych części kodu osobno przed ich zintegrowaniem

Testowanie oprogramowania.



Testy jednostkowe.

Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach programu, co umożliwia często wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do wprowadzenia błędnego fragmentu do programu.

Testy jednostkowe są również formą specyfikacji.



Właściwości dobrego testu jednostkowego.

- Powinien być powtarzalny i zautomatyzowany.
- Powinien być łatwy do zaimplementowania.
- Każdy powinien być w stanie go uruchomić jedną komendą, naciśnięciem jednego przycisku.
- Powinien być szybki.
- Powinien być wyizolowany.
- Powinien zwracać spójne wyniki. (Jeśli nic nie zmieniliśmy w danej jednostce).
- Gdy się nie powiedzie, powinno być łatwo wykryć przyczynę.



Zasady testowania jednostkowego.

- Celem jest weryfikacja rzeczywistego zachowania pojedynczej instancji (obiektu) danej klasy z zachowaniem oczekiwanym
- Sprawdzany jest efekt działania metody wywołanej na rzecz testowanego obiektu dla różnych argumentów (parametrów), badane są wartości zwracane
- Istnieje przynajmniej jeden przypadek testowy dla każdej metody – powinien istnieć jeden przypadek testowy dla każdego typu warunku początkowego i argumentów
- Przypadki testowe dotyczące jednego obiektu testowanego zebrane są wewnątrz jednej klasy testującej



Unittest.

unittest jest wbudowany w standardową bibliotekę pythona, zawiera zarówno framework testowy i runner testów.

Do inspiracji przy jego tworzeniu służyła biblioteka JUnit(Java) przez co jest bardzo podobny do głównych frameworków testowych w innych językach programowania.

Wspiera:

- Automatyzacje testów.
- Agregacje testów w kolekcje.
- Niezależność testów od frameworka raportującego.
- Programowanie zorientowane obiektowo.



Framework testowy unittest.

- Unittest wymaga tworzenia metod testowych które zawierają się w klasie.
- Klasa utworzona z wykorzystaniem unittest musi być subclassą ***unittest.TestCase***

```
import unittest

class FirstTest(unittest.TestCase):
```

- **Test case (Przypadek testowy)** – Indywidualna jednostka testowa. Sprawdza konkretne wyniki dla zakładanych danych wejściowych.
- **Test suite (Zestaw testów)** – Kolekcja przypadków testowych, innych zestawów testów, lub kombinacji przypadków i zestawów. Służy do agregowania testów które powinny być wykonywane razem.



Framework testowy unittest.

- **Test runner** – Komponent który zarządza wykonywaniem testów i raportowaniem wyników dla użytkownika. Test runner może posiadać interfejs graficzny bądź tekstowy, ewentualnie zwracać konkretną wartość informującą o wykonaniu testów.

```
class SecondTest(unittest.TestCase):  
  
    def test_01_success(self):  
        print("01")  
        self.assertEqual(2, 2)  
  
    def test_02_failure(self):  
        print("02")  
        self.assertEqual(2, 3)
```



Asercja. (Podstawowa definicja)

Asercja (ang. *assertion*) – predykat (forma zdaniowa w danym języku, która zwraca prawdę lub fałsz), umieszczony w pewnym miejscu w kodzie. Asercja wskazuje, że programista zakłada, że predykat ów jest w danym miejscu prawdziwy. W przypadku gdy predykat jest fałszywy (czyli niespełnione są warunki postawione przez programistę) asercja powoduje przerwanie wykonania programu. Asercja ma szczególne zastosowanie w trakcie testowania tworzonego oprogramowania, np. dla sprawdzenia luk lub jego odporności na błędy. Zaletą stosowania asercji jest możliwość sprawdzenia, w którym fragmencie kodu źródłowego programu nastąpił błąd. [Wikipedia]

Każda asercja zawiera wyrażenie boolowskie, które według założenia będzie prawdziwe, gdy asercja zostanie wykonana. Jeżeli wyrażenie nie jest prawdą, system zwróci **AssertionError**. Poprzez sprawdzenie, czy wyrażenie logiczne jest prawdziwe, asercja potwierdza zakładane założenia dotyczące zachowania oprogramowania, zwiększając zaufanie do naszego oprogramowania.



Asercja. (Podstawowa definicja)

Asercja (ang. *assertion*) – predykat (forma zdaniowa w danym języku, która zwraca prawdę lub fałsz), umieszczony w pewnym miejscu w kodzie. Asercja wskazuje, że programista zakłada, że predykat ów jest w danym miejscu prawdziwy. W przypadku gdy predykat jest fałszywy (czyli niespełnione są warunki postawione przez programistę) asercja powoduje przerwanie wykonania programu. Asercja ma szczególne zastosowanie w trakcie testowania tworzonego oprogramowania, np. dla sprawdzenia luk lub jego odporności na błędy. Zaletą stosowania asercji jest możliwość sprawdzenia, w którym fragmencie kodu źródłowego programu nastąpił błąd. [Wikipedia]

Każda asercja zawiera wyrażenie boolowskie, które według założenia będzie prawdziwe, gdy asercja zostanie wykonana. Jeżeli wyrażenie nie jest prawdą, system zwróci **AssertionError**. Poprzez sprawdzenie, czy wyrażenie logiczne jest prawdziwe, asercja potwierdza zakładane założenia dotyczące zachowania oprogramowania, zwiększając zaufanie do naszego oprogramowania.




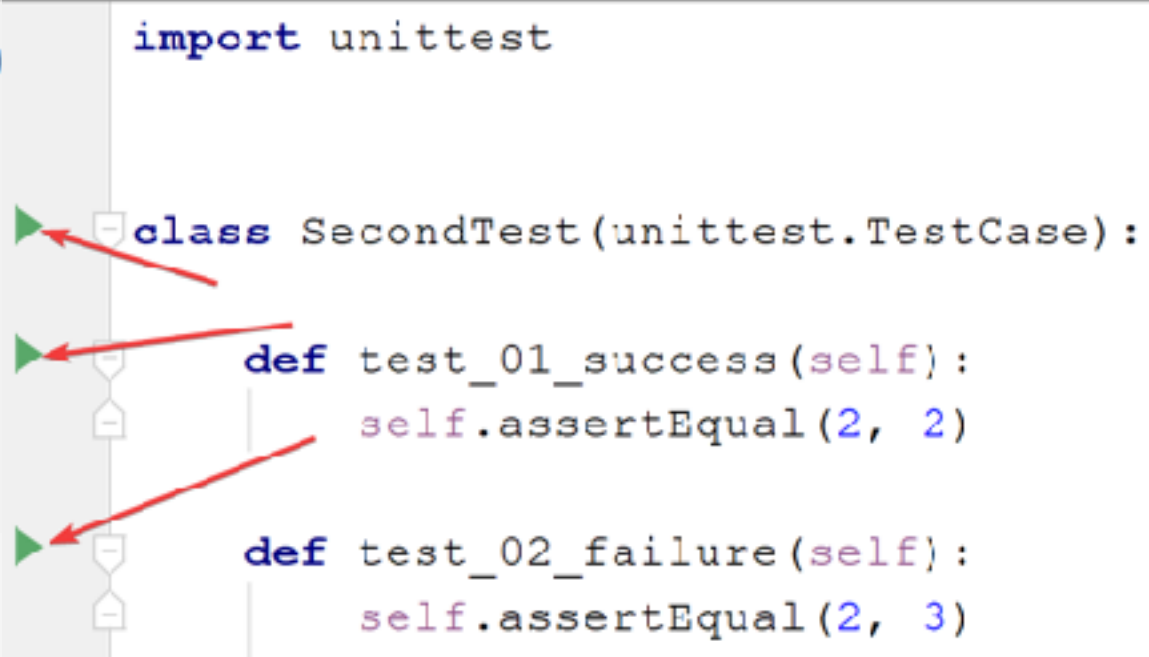




Asercje w module unittest.

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a</code> is <code>b</code>
<code>assertIsNot(a, b)</code>	<code>a</code> is not <code>b</code>
<code>assertIsNone(x)</code>	<code>x</code> is None
<code>assertIsNotNone(x)</code>	<code>x</code> is not None
<code>assertIn(a, b)</code>	<code>a</code> in <code>b</code>
<code>assertNotIn(a, b)</code>	<code>a</code> not in <code>b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

Unittest. Uruchamianie testów.



Pycharm – uruchomienie bezpośrednio z IDE.

 Run 'test_suite' Ctrl+Shift+F10	
 Debug 'test_suite'	
 Run 'test_suite' with Coverage	
 Profile 'test_suite'	
 Concurrency Diagram for 'test_suite'	



Unittest. Uruchamianie testów.

Pycharm – uruchomienie bezpośrednio z IDE. Wyniki.

```
Tests failed: 1, passed: 1 of 2 tests - 4 ms
Test Results
  start 1
    FirstTest
      test failure
        4ms
        4ms
        4ms
        4ms

Drugi test który nie przechodzi

Ran 2 tests in 0.005s

FAILED (failures=1)

Failure
Traceback (most recent call last):
  File "C:\Users\Lukasz\AppData\Local\Programs\Python\Python36\lib\unittest\case.py", line 59, in testPartExecutor
    yield
  File "C:\Users\Lukasz\AppData\Local\Programs\Python\Python36\lib\unittest\case.py", line 601, in run
    testMethod()
  File "C:\Repozytorium\sda\testowanie\start 1.py", line 11, in test_failure
    assert 2 == 3
AssertionError
```




Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Wszystkie testy.

- Można tak:

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest start_1.py
```

- Lub na końcu pliku z test case :

```
if __name__ == '__main__':  
    unittest.main()
```

```
(python) C:\Repozytorium\sda\testowanie>python start_1.py
```



Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Wszystkie testy. Wyniki.

```
class SecondTest(unittest.TestCase):  
  
    def test_01_success(self):  
        self.assertEqual(2, 2)  
  
    def test_02_failure(self):  
        self.assertEqual(2, 3)
```

```
C:\Repozytorium\sda\testowanie>python -m unittest test_2  
.F  
=====  
FAIL: test_02_failure (test_2.SecondTest)  
-----  
Traceback (most recent call last):  
  File "C:\Repozytorium\sda\testowanie\test_2.py", line 10, in test_02_failure  
    self.assertEqual(2, 3)  
AssertionError: 2 != 3  
  
-----  
Ran 2 tests in 0.001s  
  
FAILED (failures=1)
```



Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Wyniki interpretacja.

- Pierwsza linia wyświetla rezultat testów, jeden test nie przeszedł „F”, jeden przeszedł „.”

F.

- Blok FAIL zawiera:
 - Nazwę metody testowej
 - Nazwę modułu i test case
 - Ścieżkę do błędnej metody.
 - Szczegóły asercji.

```
FAIL: test_failure (test_1.FirstTest)
-----
Traceback (most recent call last):
  File "C:\Repozytorium\sda\testowanie\test_1.py", line 10, in test_failure
    assert 2 == 3
AssertionError
```



Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Konkretnie testy.

- Wybrane moduły testowe:

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest start_1 start_2
```

- Wybrana klasa testowa:

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest start_1.FirstTest
```

- Wybrana metoda testowa:

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest start_1.FirstTest.test_success
```



Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Opcje.

- Więcej detali w raporcie:

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest -v start_1
```

```
test_failure (start_1.FirstTest) ... Drugi test ktory nie przechodzi  
FAIL  
test_success (start_1.FirstTest) ... Pierwszy test ktory przechodzi.  
ok
```

- Dodatkowe opcje -h (help):

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest -h
```



Unittest. Uruchamianie testów.

Linia komend – uruchomienie z konsoli. Test Discovery. Nazewnictwo.

- Nazewnictwo metod i modułów ma ważne znaczenie dla automatycznego wykrywania które testy mają być wykonane przez test runner.
- Unittest wspiera proste wykrywanie testów przez użycie Test Discovery.
- Discovery przeszuka aktualny folder w którym się znajduje w poszukiwaniu modułów nazwanych zgodnie z ***test*.py***

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest
```

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest discover
```

- By określić inny wzór nazewnictwa należy za opcją **-p** podać wzór.

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest discover -p "start*.py"
```

Unittest. Uruchamianie testów.



Pycharm z unittestem automatycznie rozpoznaje metody które mają zaczynać się od test.

```
3  
4  class SecondTest(unittest.TestCase):  
5  
6      def test_01_success(self):  
7          self.assertEqual(2, 2)  
8  
9      def test_02_failure(self):  
10         self.assertEqual(2, 3)  
11  
12     def xxx_test(self):  
13         self.assertEqual(2, 2)
```

Unittest. Uruchamianie testów.



Taki test można uruchomić bezpośrednio z konsoli.

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
(python) C:\Repozytorium\sda\testowanie>python -m unittest -v test_2.SecondTest.xx_test
xx_test (test_2.SecondTest) ... XXXX
ok

-----

Ran 1 test in 0.001s

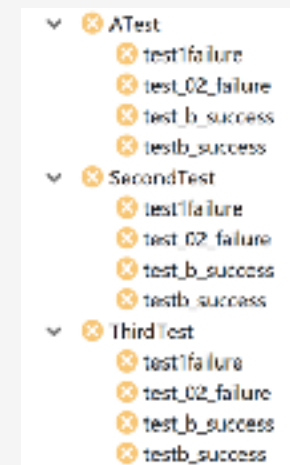
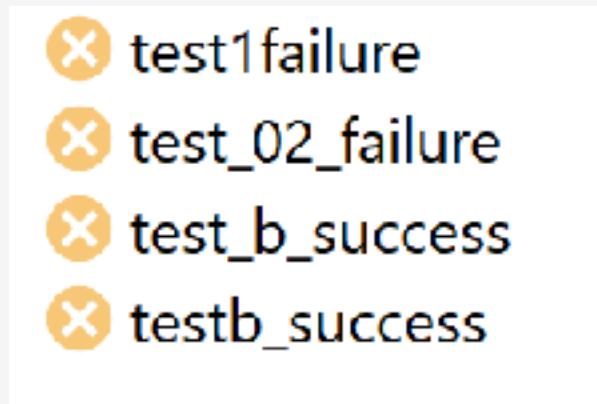
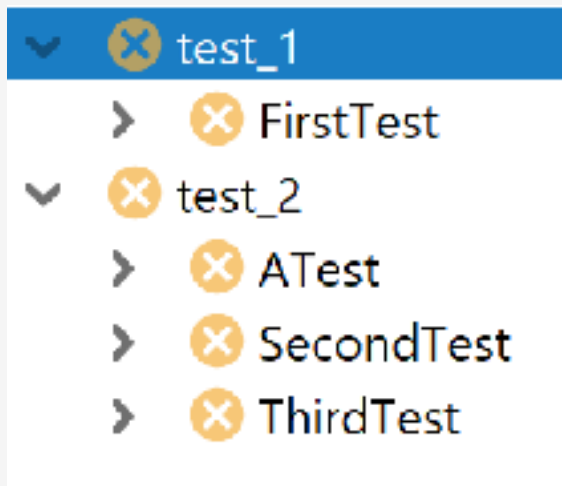
OK
```




Unittest. Kolejność wykonywania testów.

Unittest domyślnie wykonuje testy w kolejności numerycznej, później alfabetycznej.

- Jeśli istnieje potrzeba wykonywania testów w określonej kolejności (Pamiętajmy by były niezależne!) To warto wiedzieć w jakiej kolejności testy wykonuje unittest.
- Jeśli nie ma konkretnej numeracji po słowie kluczowym test, unittest patrzy na kolejność alfabetyczną. Ponadto znaki specjalne mają pierwszeństwo przed kolejnością alfabetyczną.
- Powyższe zasady dotyczą wykonywania modułów, przypadków(klas), metod w klasach.





Test Fixture.

- Test fixture (Środowisko testowe, element testowy). Reprezentuje potrzebne przygotowania do wykonania jednego bądź większej ilości testów i połączonym z tym sprzątaniem po testach.
- Unittest posiada specjalne metody które domyślnie są puste ale są wykonywane w odpowiednich momentach w danym Test Case.
- Metoda **setUp()** – unittest wywołuje ją bezpośrednio przed wykonaniem metody testowej (Każdej metody).
- Metoda **tearDown()** – unittest wywołuje ją bezpośrednio po wykonaniu metody testowej, nawet wtedy gdy ta metoda zgłosiła błąd. Jeśli metoda podczas wykonywania metody **setUp()** zostanie wywołany błąd wtedy metoda **tearDown()** nie zostanie wykonana.

Unittest. Organizacja testu.



setUp()

```
class FirstTest(unittest.TestCase):  
  
    def setUp(self) -> None:  
        print("setUp")  
  
    def test1(self):  
        print("test1")  
  
    def test2(self):  
        print("test2")
```

```
setUp  
test1  
setUp  
test2
```

Unittest. Organizacja testu.



```
tearDown()  
class FirstTest(unittest.TestCase):  
  
    def tearDown(self) -> None:  
        print("tearDown")  
  
    def test1(self):  
        print("test1")  
  
    def test2(self):  
        print("test2")
```

```
test1  
tearDown  
test2  
tearDown
```

Unittest. Organizacja testu.



setUp() + tearDown()

```
class FirstTest(unittest.TestCase):  
  
    def setUp(self) -> None:  
        print("setUp")  
  
    def tearDown(self) -> None:  
        print("tearDown")  
  
    def test1(self):  
        print("test1")  
  
    def test2(self):  
        print("test2")
```

```
setUp  
test1  
tearDown  
setUp  
test2  
tearDown
```

Unittest. Organizacja testu.



Test Fixture.

- **setUpClass()** – Metoda klasy (classmethod) wywoływana przed wszystkimi testami w danej klasie dziedziczącej po unittest.TestCase.
- **tearDownClass()** – Metoda klasy wywoływana po wszystkich testach w danej klasie.
- **setUpModule()** – Metoda która jest wykonywana przed danym modulem w którym znajdują się testy.
- **tearDownModule()** – Metoda która jest wykonywana po danym module testowym.

- **Classmethod** – Metoda która otrzymuje jako argument klasę(cls), podobnie do metody instancji które otrzymują instancje (self)



Unittest. Organizacja testu.

Test Fixture.

```
def setUpModule():
    print("setUpModule")

def tearDownModule():
    print("tearDownModule")

class FirstTest(unittest.TestCase):

    @classmethod
    def setUpClass(cls) -> None:
        print("First_setUpClass")

    @classmethod
    def tearDownClass(cls) -> None:
        print("First_tearDownClass")

    def setUp(self) -> None:
        print("First_setUp")

    def tearDown(self) -> None:
        print("First_tearDown")

    def test1(self):
        print("First_test1")
```

```
setUpModule
First_setUpClassFirst_setUp
First_test1
First_tearDown
First_setUp
First_test2
First_tearDown
First_tearDownClass
Second_setUpClassSecond_setUp
Second_test1
Second_tearDown
Second_setUp
Second_test2
Second_tearDown
Second_tearDownClass
tearDownModule
```

Autor: Łukasz Łopusiński

Prawa do korzystania z materiałów posiada Software
Development Academy



Test Suite.

- Unittest w większości przypadków sam zbierze Test Case w Zestaw testów odwołując się do `unittest.main()`.
- Gdy chcemy być pewni, że testy są odpowiednio pogrupowane możemy do tego użyć `unittest.TestSuite`.
- `TestSuite` może być budowany na kilka sposobów.

Unittest. Organizacja testu.



Test Suite.

```
class SecondTest(unittest.TestCase):  
    def runTest(self):  
        self.assertEqual("a", "a")
```

```
import unittest  
  
from testowanie.test_2 import SecondTest  
  
def suite():  
    suite = unittest.TestSuite()  
    suite.addTest(SecondTest())  
    return suite  
  
if __name__ == '__main__':  
    runner = unittest.TextTestRunner()  
    runner.run(suite())
```

Unittest. Organizacja testu.



Test Suite.

```
class SecondTest(unittest.TestCase):  
  
    def test_01_success(self):  
        print("01")
```

```
import unittest  
  
from testowanie.test_2 import SecondTest  
  
def suite():  
    suite = unittest.TestSuite()  
    suite.addTest(SecondTest('test_01_success'))  
    return suite  
  
if __name__ == '__main__':  
    runner = unittest.TextTestRunner()  
    runner.run(suite())
```

Unittest. Organizacja testu.



Test Suite.

```
import unittest

def setUpModule():
    print("setUpModule")

def tearDownModule():
    print("tearDownModule")

class FirstTest(unittest.TestCase):

    @classmethod
    def setUpClass(cls) -> None:
        print("First_setUpClass")

    @classmethod
    def tearDownClass(cls) -> None:
        print("First_tearDownClass")

    def setUp(self) -> None:
```

```
import unittest

import testowanie.test_1 as test_1

def suite():
    loader = unittest.TestLoader()
    suite = unittest.TestSuite()
    suite.addTest(loader.loadTestsFromTestCase(test_1.FirstTest))
    suite.addTests([loader.loadTestsFromModule(test_1), loader.loadTestsFromTestCase(test_1.FirstTest)])
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner(verbosity=3)
    runner.run(suite())
```



Asercje w module unittest.

Metoda	Sprawdza
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a</code> is <code>b</code>
<code>assertIsNot(a, b)</code>	<code>a</code> is not <code>b</code>
<code>assertIsNone(x)</code>	<code>x</code> is None
<code>assertIsNotNone(x)</code>	<code>x</code> is not None
<code>assertIn(a, b)</code>	<code>a</code> in <code>b</code>
<code>assertNotIn(a, b)</code>	<code>a</code> not in <code>b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>



Given When Then

Test powinien być małą historyjką która na podstawie określonych warunków, po wykonaniu odpowiedniej akcji (Testu), doprowadzi do odpowiednich wyników(Asercje).

Given tworzy sekcję w której tworzymy założenia początkowe. Ustawiamy stan systemu (zwany również stanem świata) na potrzebny do testów.

When w tej sekcji wykonujemy akcję którą chcemy testować.

Then wykonujemy sprawdzenia czy aplikacja zachowała się zgodnie z oczekiwaniami. Najczęściej poprzez wykorzystanie asercji lub interakcji z mockami

Nie zawsze musimy używać wszystkich części, na przykład gdy używamy **setUp()** .

Scenariusz testu.



Given When Then

```
def test_01_given_when_then(self):  
    # Given  
    account = Account(100, "owner")  
  
    # When  
    account.transfer(-50)  
  
    # Then  
    self.assertEqual(account.balance(), 50)
```



Napisz testy do podanego przykładu.

- Są dwie klasy: Account i Card.
- Odwołanie się do obiektu Account powinno zwracać numer konta.
- Metoda Account.owner powinna zwracać Imię i nazwisko właściciela.
- Metoda Account.balance powinna zwracać aktualny stan konta.
- Metoda Account.number powinna zwracać numer konta.
- Metoda Account.transfer powinna zmieniać stan konta o podaną kwotę
- Odwołanie się do obiektu Card powinno zwracać imię i nazwisko właściciela konta.
- Metoda Card.check_pin powinna sprawdzić czy pin jest poprawny.
- Metoda Card.get_account powinna zwrócić konto do którego karta jest „podpięta”



Dekoratory testów.

- `@unittest.skip(reason)`
Pomija wybrany test bezwarunkowo. Reason -> Powód dla którego ma być pominięty jako wiadomość.
- `@unittest.skipIf(condition, reason)`
Pomija test jeśli warunek jest spełniony
- `@unittest.skipUnless(condition, reason)`
Pomija test, jeśli warunek jest spełniony wtedy go wykonuje.
- `@unittest.expectedFailure`
Jeśli test będzie nieudany, wtedy zostanie to odnotowane jako sukces, jeśli przejdzie to wtedy jako nieudany



Pytest <https://docs.pytest.org/en/latest/>

Pytest – zewnętrzny moduł do wykonywania testów w języku python

Cechy:

- Szczegółowy opis na temat niepowodzenia testu w asercji.
- Możliwość filtrowania testów.
- Możliwość uruchomienia testów od ostatniego który się nie powiódł
- Mnóstwo wtyczek i fixtures.
- Uruchamia test suites i test cases z unittest i z nose.
- Uruchamianie testów bez potrzeby klas.

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F [100%]

===== FAILURES =====
_____ test_answer _____

    def test_answer():
>         assert inc(3) == 5
E         assert 4 == 5
E         + where 4 = inc(3)

test_sample.py:6: AssertionError
===== 1 failed in 0.12 seconds =====
```



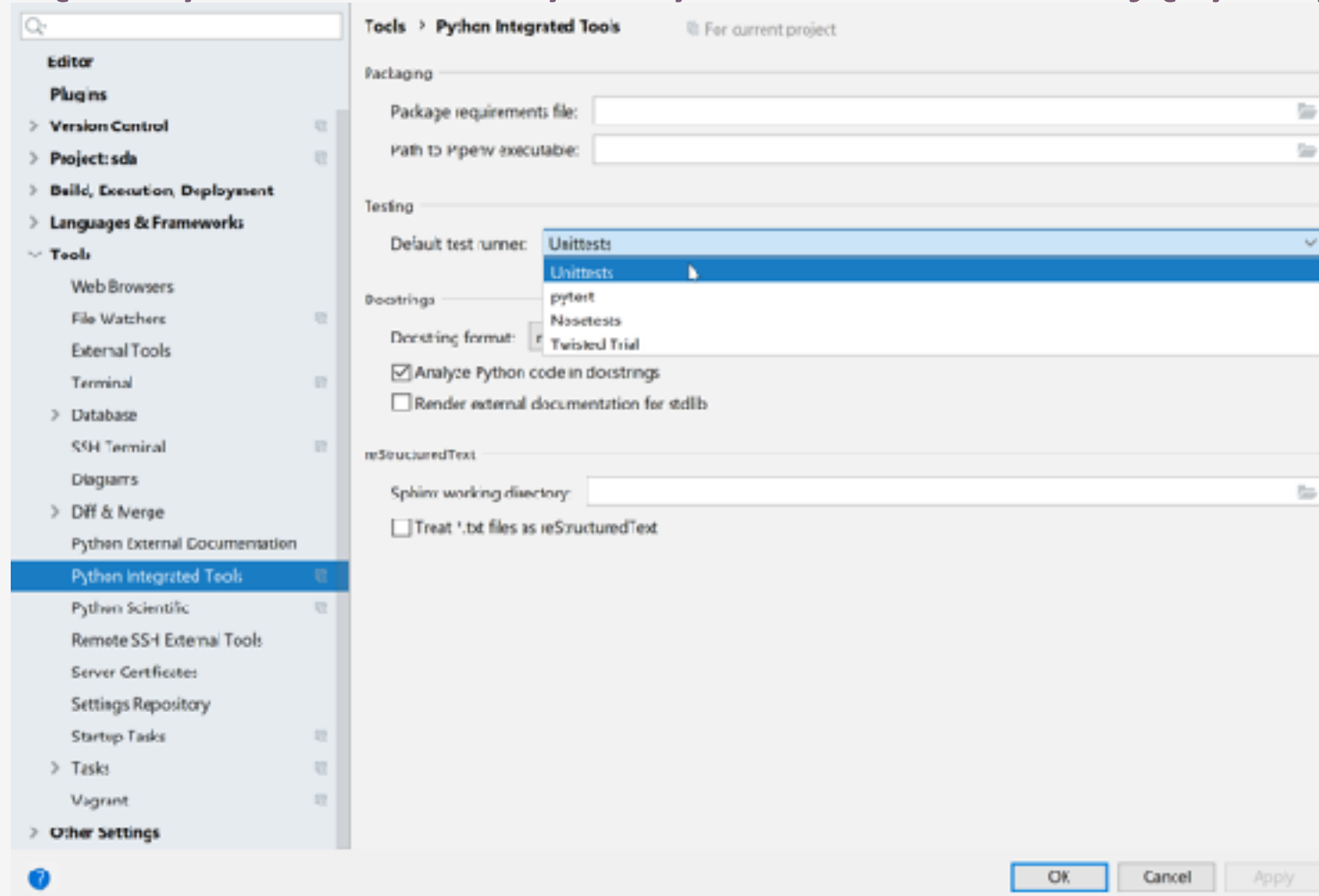
Pytest – zewnętrzny moduł do wykonywania testów w języku python

- ***pip install pytest***
- By uruchomić testy napisane z wykorzystaniem unittest, wystarczy wpisać w linii komand:
pytest <nazwa_modułu>.py

Pytest <https://docs.pytest.org/en/latest/>



Pytest – zewnętrzny moduł do wykonywania testów w języku python





Test-Driven Development. [*„Programowanie sterowane testami.”*]

- Proces tworzenia (nie testowania) oprogramowania
- Polega na pracy w krótkich cyklach.
- W trakcie jednego cyklu jest kilka faz a pierwszą z nich jest pisanie testów.
- Zalicza się do zwinnych (Agile) technik programowania.
- Wywodzi się z ruchu XP (Extreme Programming), który wprowadził wiele innych nowatorskich pojęć np. Pair Programming, programowanie w parach.
- W podejściu TDD testy są przede wszystkim specyfikacją wymagań a dopiero w dalszej kolejności jednostkowymi testami.



Test-Driven Development. [*„Programowanie sterowane testami.”*]

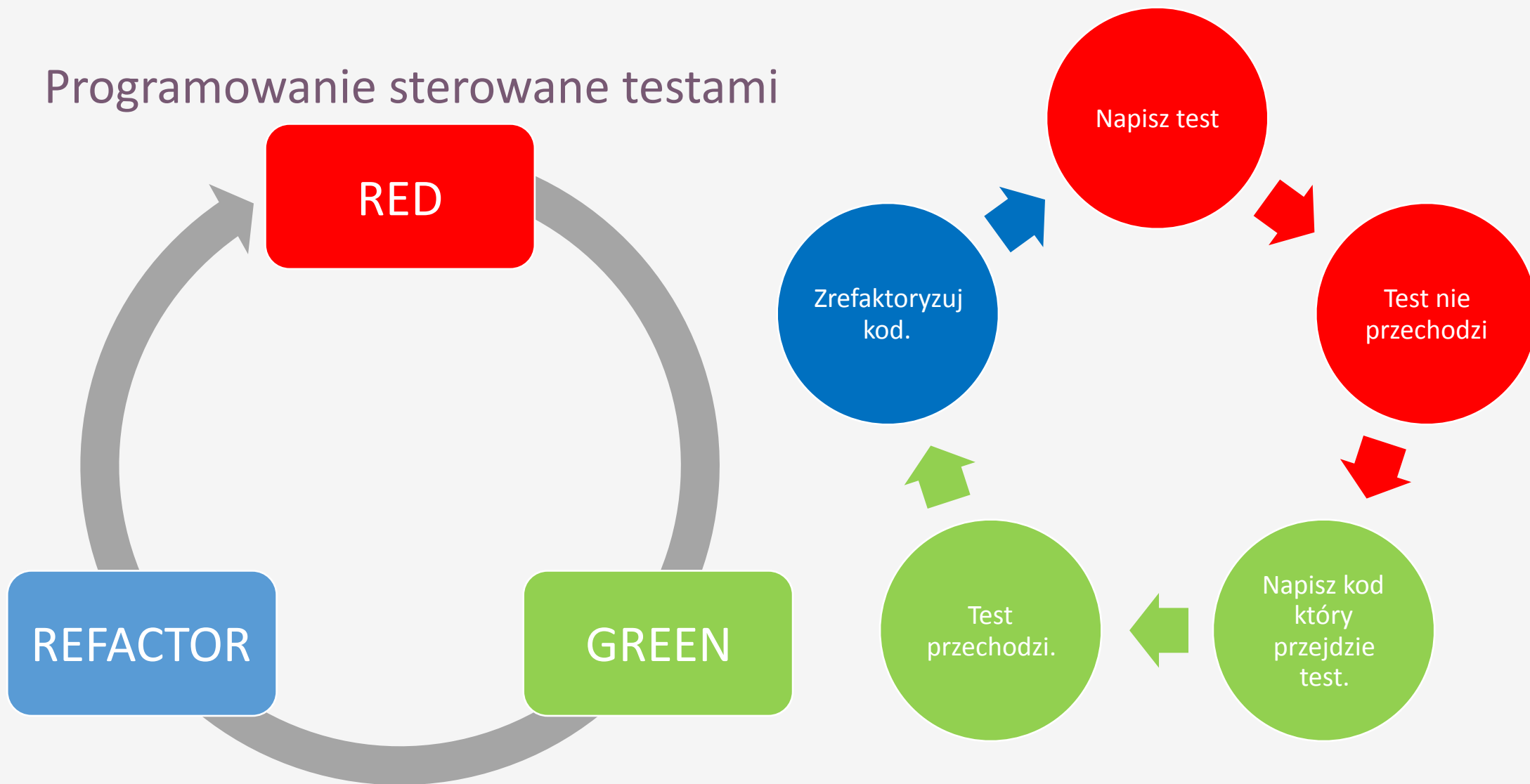
TDD – inaczej też nazywane test-first (najpierw testuj).

Cykl życia TDD:

- Piszemy test który nie przechodzi.
- Tworzymy minimalną ilość kodu by test przeszedł.
- Refaktoryzujemy kod.
- Powtarzamy cykl.



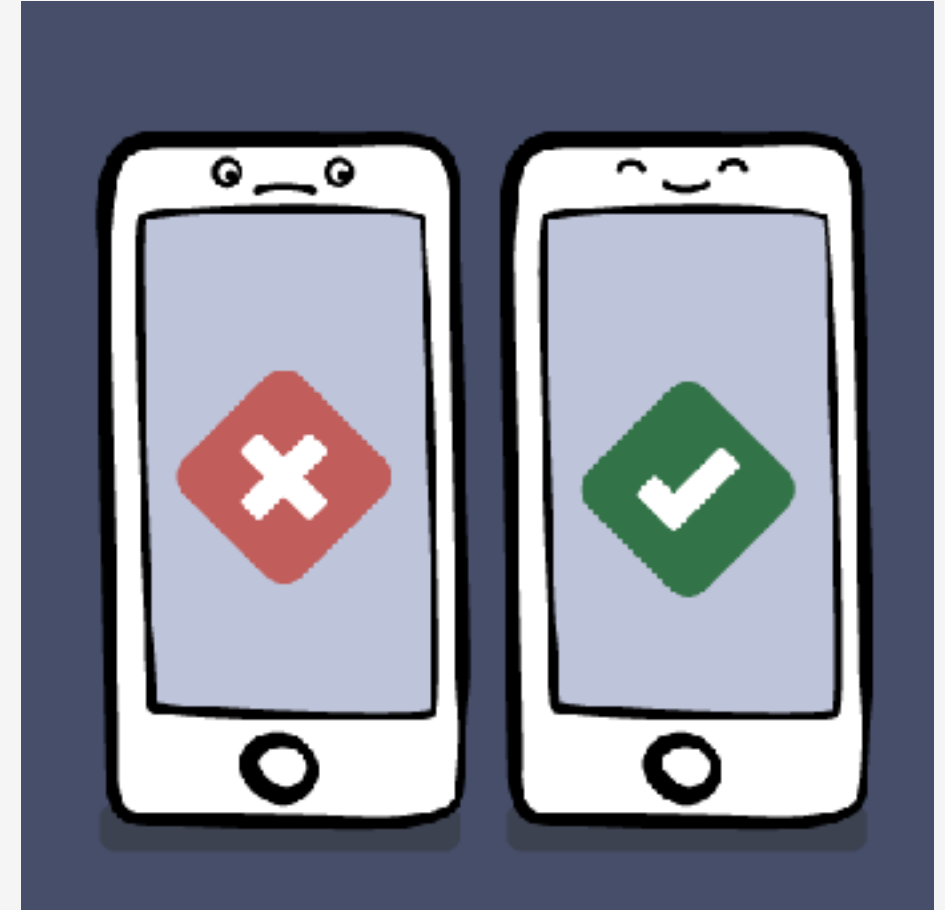
Programowanie sterowane testami





Najlepsze praktyki testowania - FIRST

- Fast - szybkie
- Isolated / Independent - izolowane / niezależne
- Repeatable - powtarzalne
- Self-validating - samosprawdzające
- Thorough - gruntowne





TDD jako wzorzec.

- **Izolowanie testów** -> Testy nie powinny być od siebie zależne w żadnym stopniu.
- **Lista testów** -> Zanim zaczniesz sporządzić listę wszystkich testów o których wiesz, że powinienes je napisać.
- **Najpierw testy** -> Zanim zaczniemy pisać kod, powinniśmy napisać testy które go zweryfikują.
- **Najpierw asercje** -> Pisanie testu zaczynamy od napisania asercji.
- **Dane testowe** -> Powinniśmy używać danych powodujących, że testy będą łatwe do analizy i wnioskowania. (Rzeczywiste dane).
- **Przejrzyste dane** -> Powinniśmy umieścić w teście wartość spodziewaną i faktycznie otrzymaną, przy czym należy uwidocznic między nimi relacje.

Zadanie TDD.



- Jako posiadacz konta chcę wypłacić pieniądze z bankomatu.
- Jako posiadacz konta chcę przelać pieniądze na swoje konto z innego konta.
- Jako bankomat chcę zablokować kartę w przypadku 3 prób niepoprawnie wpisanego pinu.

Testowanie wyjątków.



Asercje związane z testowaniem wyjątków.

- W Pythonie wyjątki są obiektami dziedziczącymi po klasie ***BasicException***.
- Wyjątek to sygnał, że wystąpił błąd lub inny niepożądany stan.
- Zawierają informacje o typie błędu, miejscu w kodzie, w którym wystąpił wyjątek oraz (opcjonalne) dodatkowej wiadomości.
- Wystąpienie wyjątku, który nie jest obsługowany spowoduje zakończenie działania aplikacji.
- Wyjątki rzuca się za pomocą słowa kluczowego *raise* a przechwytuje za pomocą słowa kluczowego *except*.

```
def value_error():  
    raise ValueError
```

```
def try_except_error():  
    try:  
        value_error()  
    except ValueError:  
        print("Found error")
```



Dlaczego testujemy wyjątki?

Ponieważ wyjątki wyrzucane są w sytuacjach błędów testy powinny sprawdzać co dzieje się w takich szczególnych wypadkach.

Sprawdzać należy nie tylko sam fakt wyrzucenia wyjątku ale również:

- Czy typ wyrzuconego wyjątku jest adekwatny do zaistniałej sytuacji?
- Czy wiadomość przekazana w wyjątku jest zrozumiała?

Testowanie wyjątków.



Asercje związane z testowaniem wyjątków.

Metoda	Sprawdza
<code>assertRaises(exc, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>exc</i>
<code>assertRaisesRegex(exc, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>exc</i> and the message matches regex <i>r</i>
<code>assertWarns(warn, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>warn</i>
<code>assertWarnsRegex(warn, r, fun, *args, **kwargs)</code>	<code>fun(*args, **kwargs)</code> raises <i>warn</i> and the message matches regex <i>r</i>
<code>assertLogs(logger, level)</code>	The with block logs on <i>logger</i> with minimum <i>level</i>

Testowanie wyjątków.



Asercje związane z testowaniem wyjątków.

- Wyjątki w języku Python testuje się z użyciem menadżera kontekstu with.

```
def test_success(self):  
    with self.assertRaises(ValueError):  
        int('XXXX')  
  
def test_fail(self):  
    with self.assertRaisesRegex(ValueError, 'literal'):  
        int('1')
```

Testy sparametryzowane (DDT)



- Bardzo często piszemy testy dla wielu różnych parametrów wejściowych.
- Tworzenie oddzielnych przypadków testowych dla każdego parametru oddzielnie jest czasochłonne i może przesłaniać obraz testów.
- Jeśli mamy dużo danych które powinny przejść konkretne testy, łatwiej byłoby podać listę wszystkich parametrów.
- Testy sparametryzowane można też nazwać DDT -> Data Driven Testing, testowanie sterowane danymi.
- Uniwersalnym modułem który pozwala tworzyć testy sparametryzowane jest *parameterized*.
- Oprócz niego można używać wbudowanych dekoratorów w pytest, bądź biblioteki ddt.



Testy sparametryzowane (DDT)

- *pip install parameterized*

```
import unittest

from parameterized import parameterized

class TestSequence(unittest.TestCase):
    @parameterized.expand([
        ["foo", "a", "a", ],
        ["bar", "a", "b"],
        ["lee", "b", "b"],
    ])
    def test_sequence(self, name, a, b):
        self.assertEqual(a, b)
```



Mock.

Mock object – Zamiennik, wydmuszka, zaślepka, **Atrapa**. Imituje prawdziwy obiekt wewnątrz środowiska testowego. Jest to wszechstronne i przydatne narzędzie które zwiększa wydajność i jakość testów.

Mocków używa się by kontrolować zachowanie programu podczas testów.

Na przykład:

- Jeśli program wykonuje zapytania do zewnętrznego serwisu, wtedy test sprawdza jedynie zachowanie serwisu na tyle na ile tego oczekujemy. Niekiedy zmiany w takich serwisach mogą spowodować poważne awarie i nieudane wykonanie testów.
- Ze względu na to warto kontrolować środowisko i zastępować zewnętrzne serwisy wydmuszkami które będą robić jedynie to czego od nich oczekujemy.

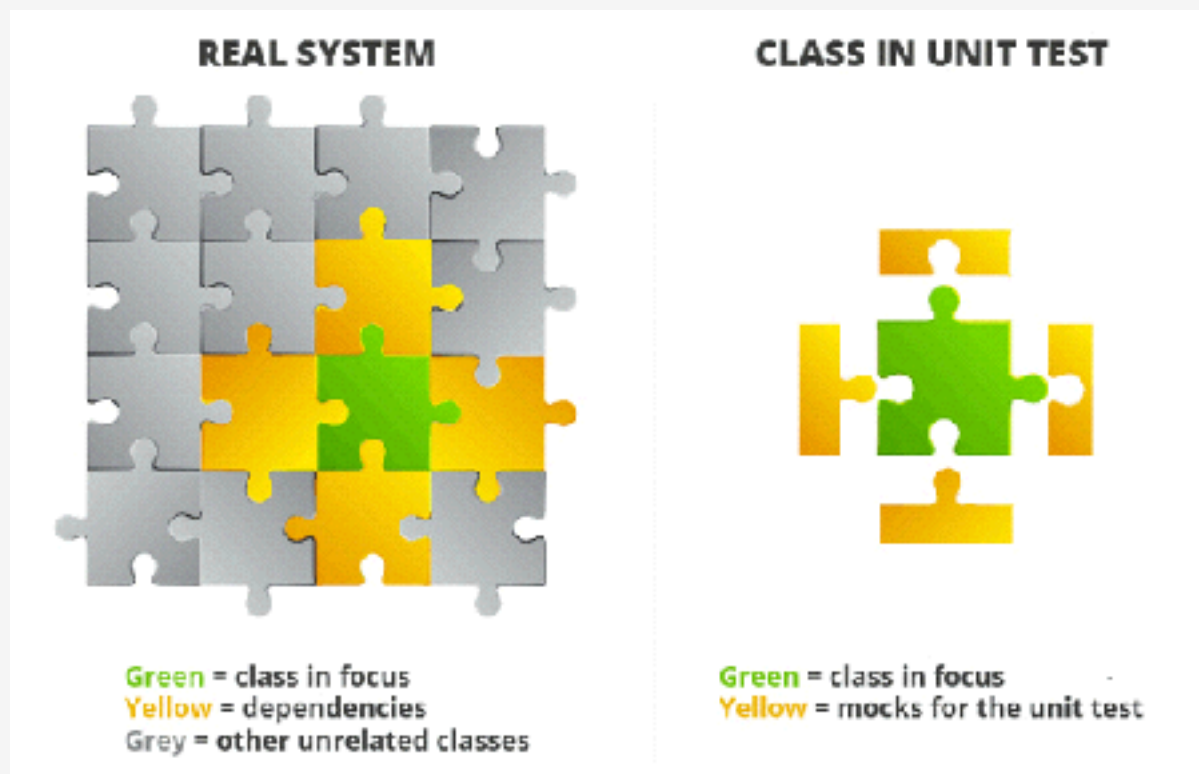


Mock - Atrapa

- Atrapa jest symulowanym prostym obiektem, który naśladuje zachowanie rzeczywistego obiektu.
- Atrapy mają taki sam interfejs jak naśladowane obiekty.
- Można ich używać wielokrotnie i w różnych konfiguracjach.
- Ograniczają liczbę niezbędnych zależności.
- Wynik testów z użyciem atrapy może być deterministyczny.



Mock - Atrapa





Unittest.Mock w Python

Python posiada wbudowane mocki razem z pakietem unittest.

Używanie mocków w pythonie pomoże kontrolować wykonywanie programu i pozwoli zwiększyć pokrycie kodu testami.

Obiekt Mock w pythonie zawiera dane na temat jego użycia, które można sprawdzić:

- Czy dana metoda została wywołana.
- Jak dana metoda została wywołana.
- Jak często dana metoda została wywoływana.



Unittest.Mock w Python

- Mock – uniwersalny obiekt Mock. Można się do niego odwoływać, tworzy atrybuty w momencie gdy się do niego dostajemy. Odwoływanie się do tego samego atrybutu zwraca zawsze ten sam wynik.

```
mock = Mock()
mock.something = "something"
mock.x = 2 * 3
```

```
<Mock id='16576336'>
something
6
```



Unittest.Mock w Python

- MagiMock – dziedziczy po klasie Mock. Wszystkie „Magiczne” metody są już przygotowane i gotowe do użycia. Ma też warianty NonCallable -> Gdy chcemy zasymulować obiekty, które nie powinny być wywoływane

```
account = Account(1, "Jan")
account.owner = MagicMock(return_value="Kazimierz")

print(account.owner())
```



Unittest.Mock w Python

- patch – używany jako dekorator, bądź z menadżerem kontekstu „with”, pozwala na łatwe, tymczasowe podmienienie klasy w wybranym Mock module. Defaultowo patch tworzy obiekt MagicMock

```
class SomeClass():  
  
    def something(self):  
        return "Something"  
  
with patch.object(SomeClass, 'something', return_value="Other") as new_mock:  
    some = SomeClass()  
    print(some.something())
```