

Unit testing techniques and Role of Test doubles

Ritesh Mehrotra

What to test?

Everything!



What to test?

36 responses



Consumer perspectives

- Does it work as expected?
- Is my data safe?
- Is it easy to use?
- Is it reliable?

Provider perspectives

- Does application work as expected?
- Is customer data safe?
- Is application easy to use?
- Is application reliable?
- Is application scalable?
- Is application maintainable?
- Is application reusable?

Types of tests

TESTING

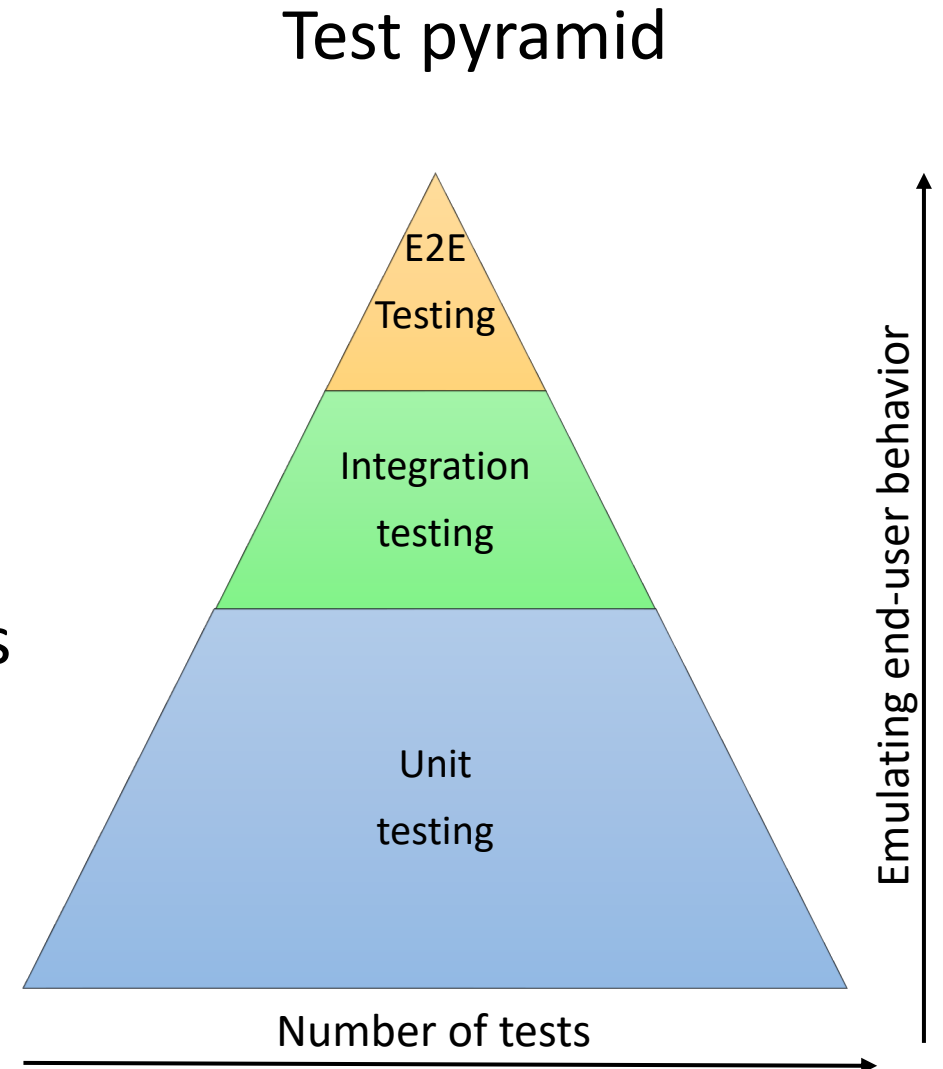
ACCEPTANCE
PERFORMANCE
CONTRACT
INTEGRATION
UNIT
PENETRATION
USABILITY

Today's focus

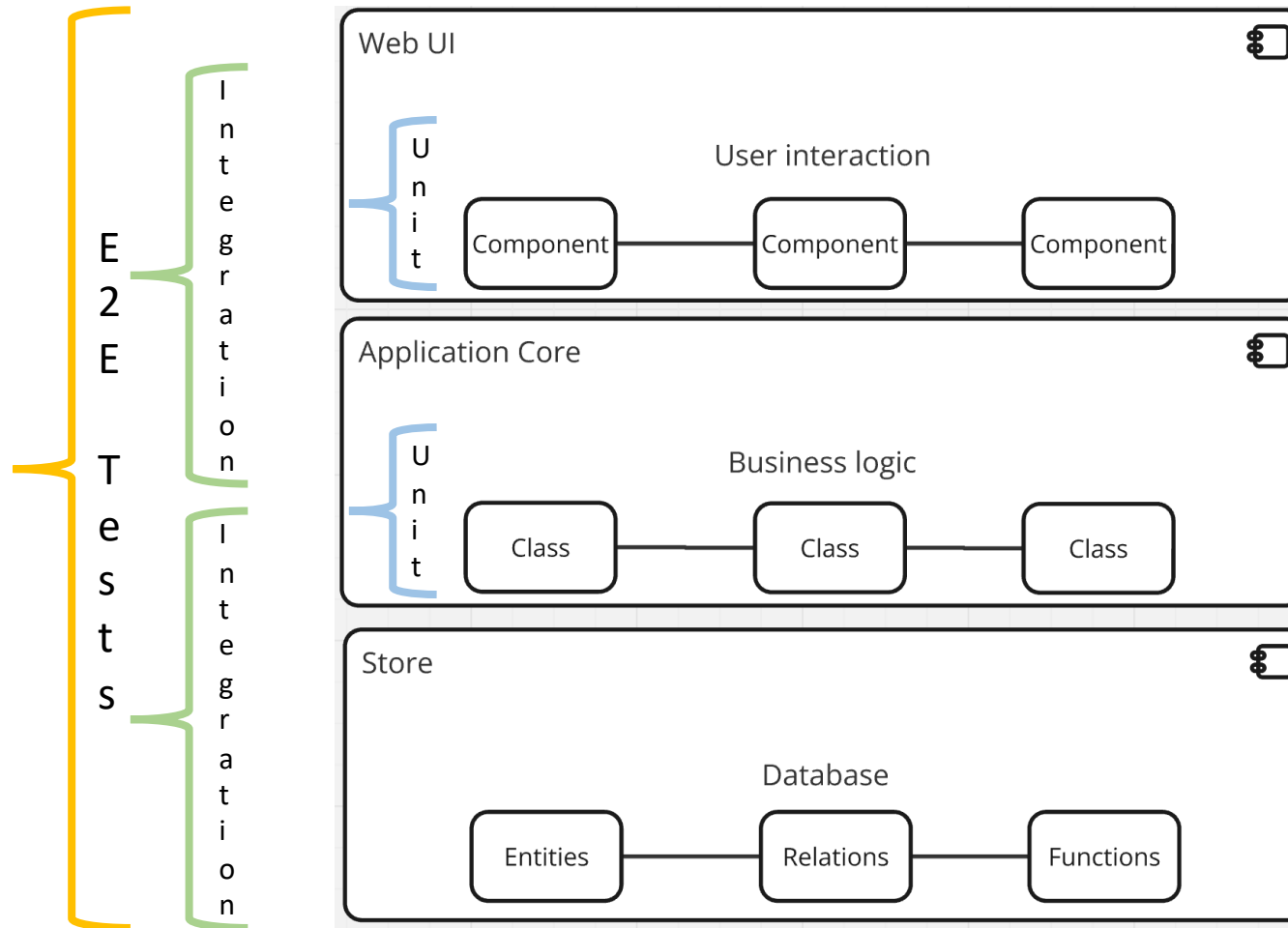
- Unit testing patterns and anti-patterns
- Test doubles - Fakes, Stubs, Mocks, Dummies and Spies

Approach to testing

- Validate the application use cases
 - Positive scenarios
 - Negative scenarios
 - Side effects
 - Exceptional flow
- Interaction between different components
- End to end flow of application



Layered N-tier architecture



Unit Testing vs Integration Testing

Characteristics	Unit Test	Integration Test
Interface	Works independently of real interface (File system, Database, API)	Depends on interface
Time	Quick to run	Time consuming operation
Reliability	Very reliable since meant to be isolated	Flaky at time depending on environment stability
Target	Tests behavior of the code	Tests behavior as well as interactions between objects
Environment	Environment independent	Environment dependent

Struggles with Unit testing

- Not valuable enough?
- Time consuming exercise
- Contributing factors
 - Code coupling makes maintainability and testability harder
 - Complex code leads to complex and flaky tests
 - Tests need to be updated every time code structure changes
- Unit testing for sake of code coverage?

Structure of a test

- Arrange – Bring SUT and its dependencies to a desired state
- Act – Call methods on SUT and capture the return value (if any)
- Assert – Verify the outcome

```
20      @Test
21      public void test_move_forward() {
22          //Arrange
23          Rover rover = new Rover( x: 1, y: 1, Direction.NORTH);
24          //Act
25          rover.moveForward();
26          //Assert
27          assertEquals( expected: "1 2 N", rover.locate());
28      }
```

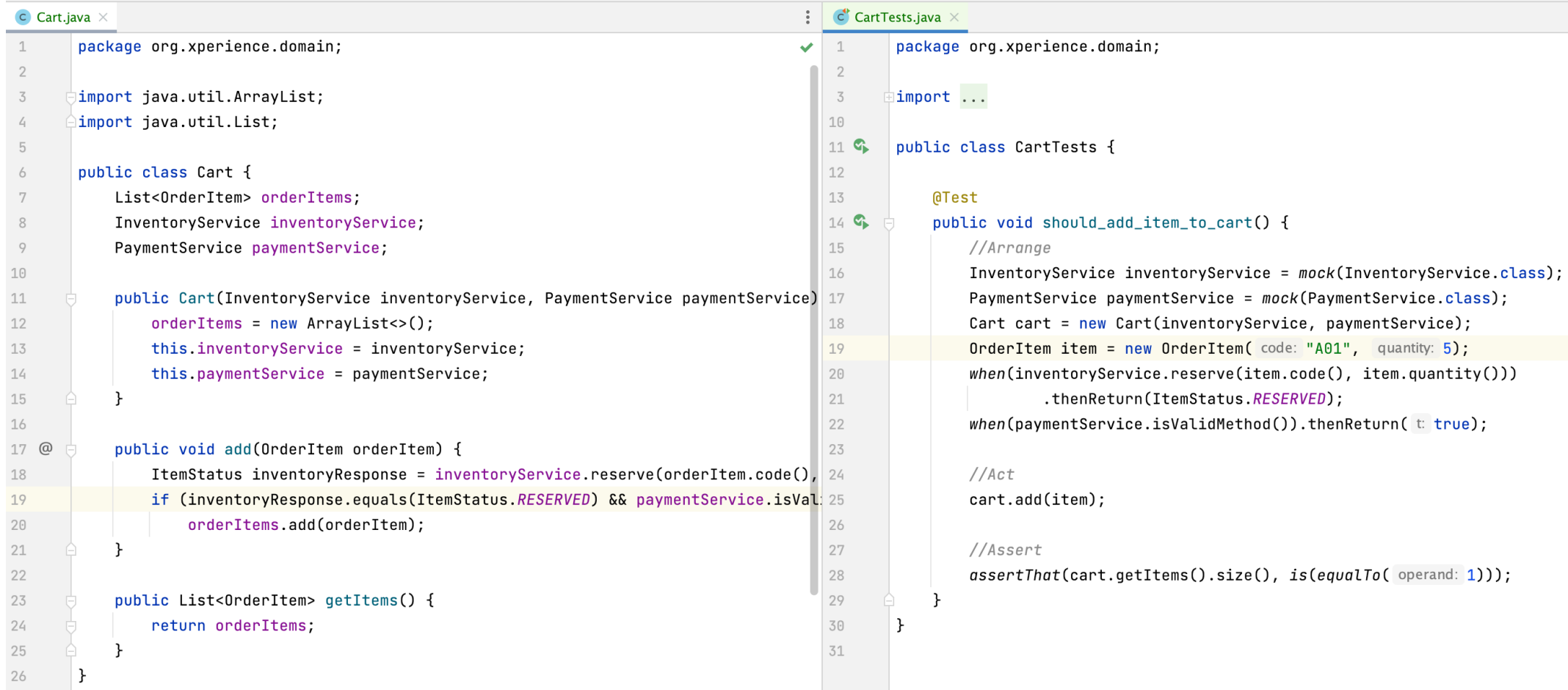
Trivial tests

```
User.java x
1 package org.xperience.domain;
2
3 public class User {
4     private String username;
5
6     public String getUsername() { return username; }
7
8
9     public void setUsername(String username) { this.username = username; }
10
11 }
12
13
14
UserTests.java x
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import static org.junit.jupiter.api.Assertions.assertNotNull;
3
4 import org.junit.jupiter.api.Test;
5 import org.xperience.domain.User;
6
7 public class UserTests {
8
9     @Test
10     public void should_instantiate_User(){
11         User user = new User();
12         assertNotNull(user);
13     }
14
15     @Test
16     public void should_set_username() {
17         User user = new User();
18         user.setUsername("new_username");
19         assertEquals( expected: "new_username", user.getUsername());
20     }
21 }
22
```

Its not valuable if we our tests assert on

- Getters and Setters
- Object instantiation
- Composition of a class with its coordinating classes
- A mathematical formula like multiplication or square root

Complex test



```
1 package org.xperience.domain;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Cart {
7     List<OrderItem> orderItems;
8     InventoryService inventoryService;
9     PaymentService paymentService;
10
11     public Cart(InventoryService inventoryService, PaymentService paymentService) {
12         orderItems = new ArrayList<>();
13         this.inventoryService = inventoryService;
14         this.paymentService = paymentService;
15     }
16
17     @Test
18     public void add(OrderItem orderItem) {
19         ItemStatus inventoryResponse = inventoryService.reserve(orderItem.code(),
20             if (inventoryResponse.equals(ItemStatus.RESERVED) && paymentService.isValidMethod()) {
21                 orderItems.add(orderItem);
22             }
23     }
24
25     public List<OrderItem> getItems() {
26         return orderItems;
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
```

- Too large arrangement
- Indicative of an abstraction issue

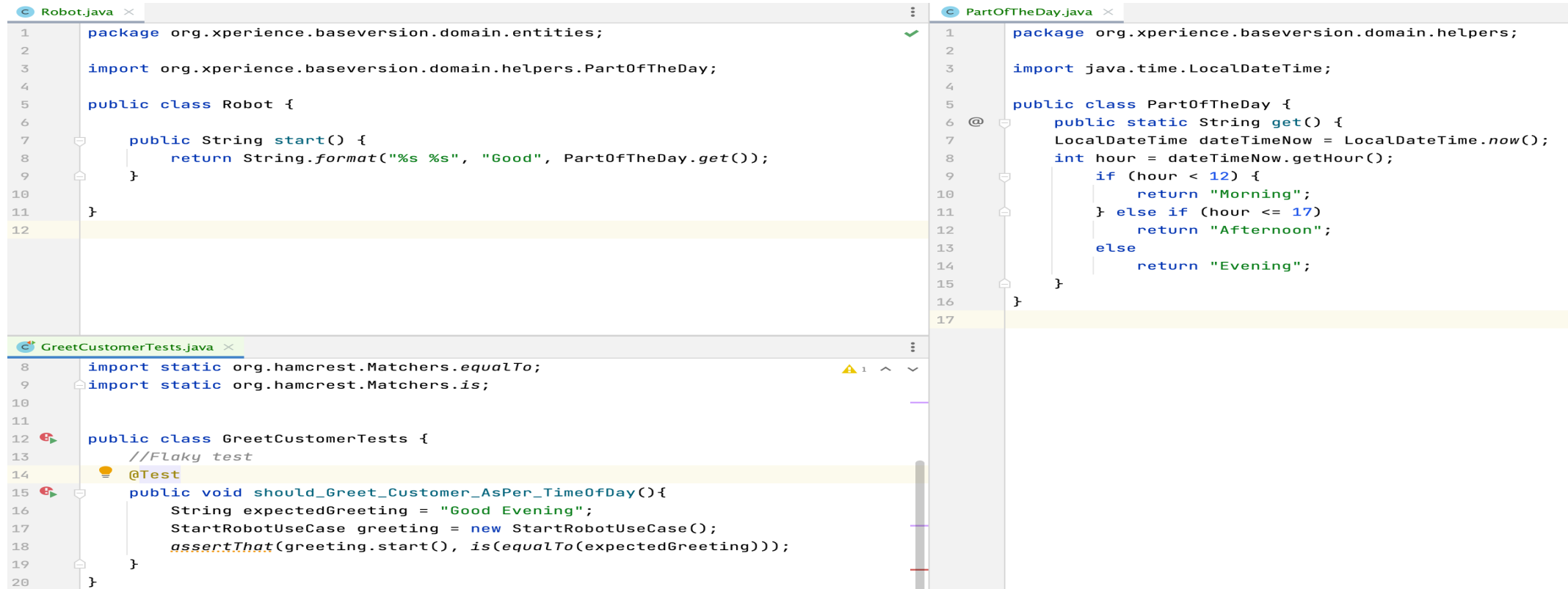
Partial test

```
Cart.java x
1 package org.xperience.domain;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Cart {
7     List<OrderItem> orderItems;
8     InventoryService inventoryService;
9     PaymentService paymentService;
10
11     public Cart(InventoryService inventoryService, PaymentService paymentService) {
12         orderItems = new ArrayList<>();
13         this.inventoryService = inventoryService;
14         this.paymentService = paymentService;
15     }
16
17     @ public void add(OrderItem orderItem) {
18         ItemStatus inventoryResponse = inventoryService.reserve(orderItem.code(),
19         if (inventoryResponse.equals(ItemStatus.RESERVED) && paymentService.isValid
20             orderItems.add(orderItem);
21     }
22
23     public List<OrderItem> getItems() {
24         return orderItems;
25     }
26 }

CartTests.java x
1 package org.xperience.domain;
2
3 import ...
4
10
11 public class CartTests {
12
13     @Test
14     public void should_add_item_to_cart() {
15         //Arrange
16         InventoryService inventoryService = mock(InventoryService.class);
17         PaymentService paymentService = mock(PaymentService.class);
18         Cart cart = new Cart(inventoryService, paymentService);
19         OrderItem item = new OrderItem( code: "A01", quantity: 5);
20         when(inventoryService.reserve(item.code(), item.quantity()))
21             .thenReturn(ItemStatus.RESERVED);
22         when(paymentService.isValidMethod()).thenReturn( t: true);
23
24         //Act
25         cart.add(item);
26
27         //Assert
28         assertThat(cart.getItems().size(), is(equalTo( operand: 1)));
29     }
30 }
31
```


Flaky test

- Tests that pass or fail inconsistently, even when the code hasn't changed. These tests undermine confidence in the test suite.

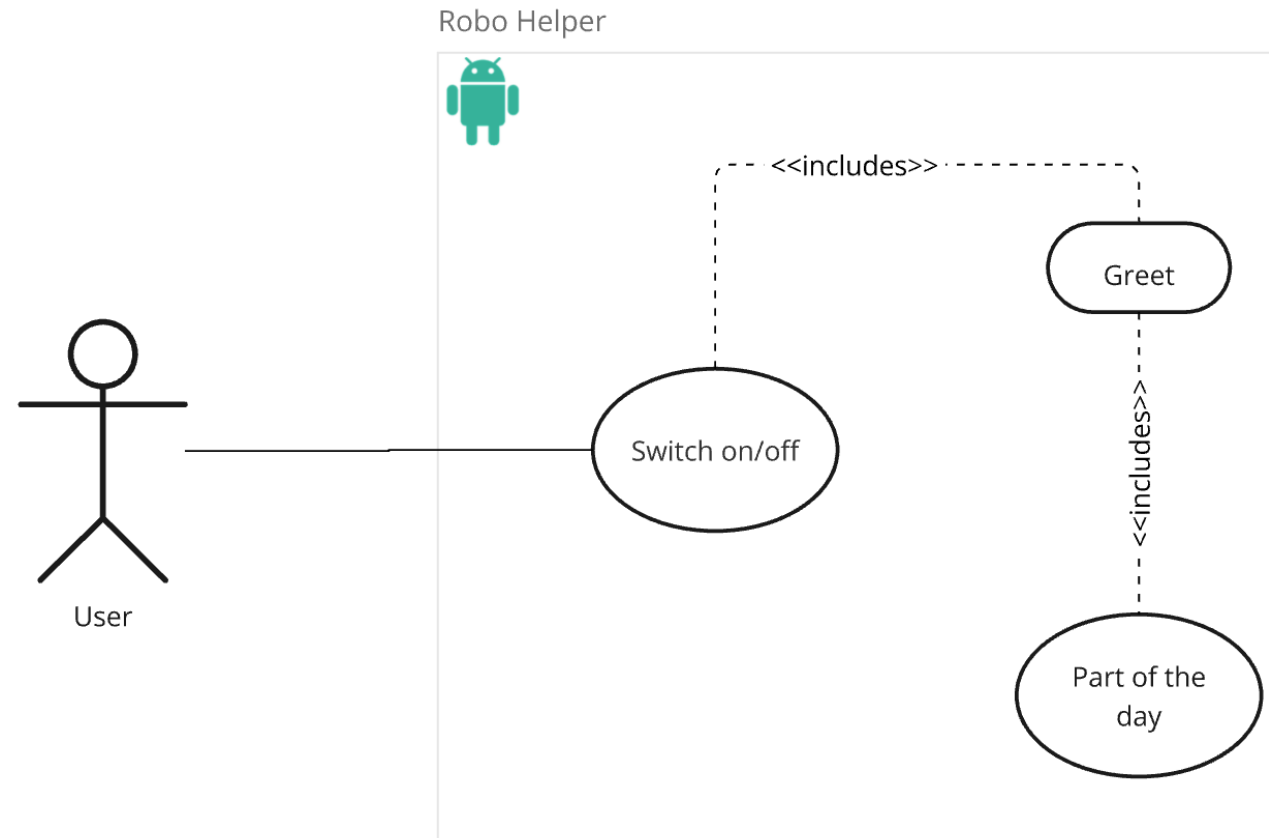


```
Robot.java
1 package org.xperience.baseversion.domain.entities;
2
3 import org.xperience.baseversion.domain.helpers.PartOfDay;
4
5 public class Robot {
6
7     public String start() {
8         return String.format("%s %s", "Good", PartOfDay.get());
9     }
10
11 }
12
```

```
PartOfDay.java
1 package org.xperience.baseversion.domain.helpers;
2
3 import java.time.LocalDateTime;
4
5 public class PartOfDay {
6     @
7     public static String get() {
8         LocalDateTime dateTimeNow = LocalDateTime.now();
9         int hour = dateTimeNow.getHour();
10         if (hour < 12) {
11             return "Morning";
12         } else if (hour <= 17) {
13             return "Afternoon";
14         } else {
15             return "Evening";
16         }
17     }
18 }
```

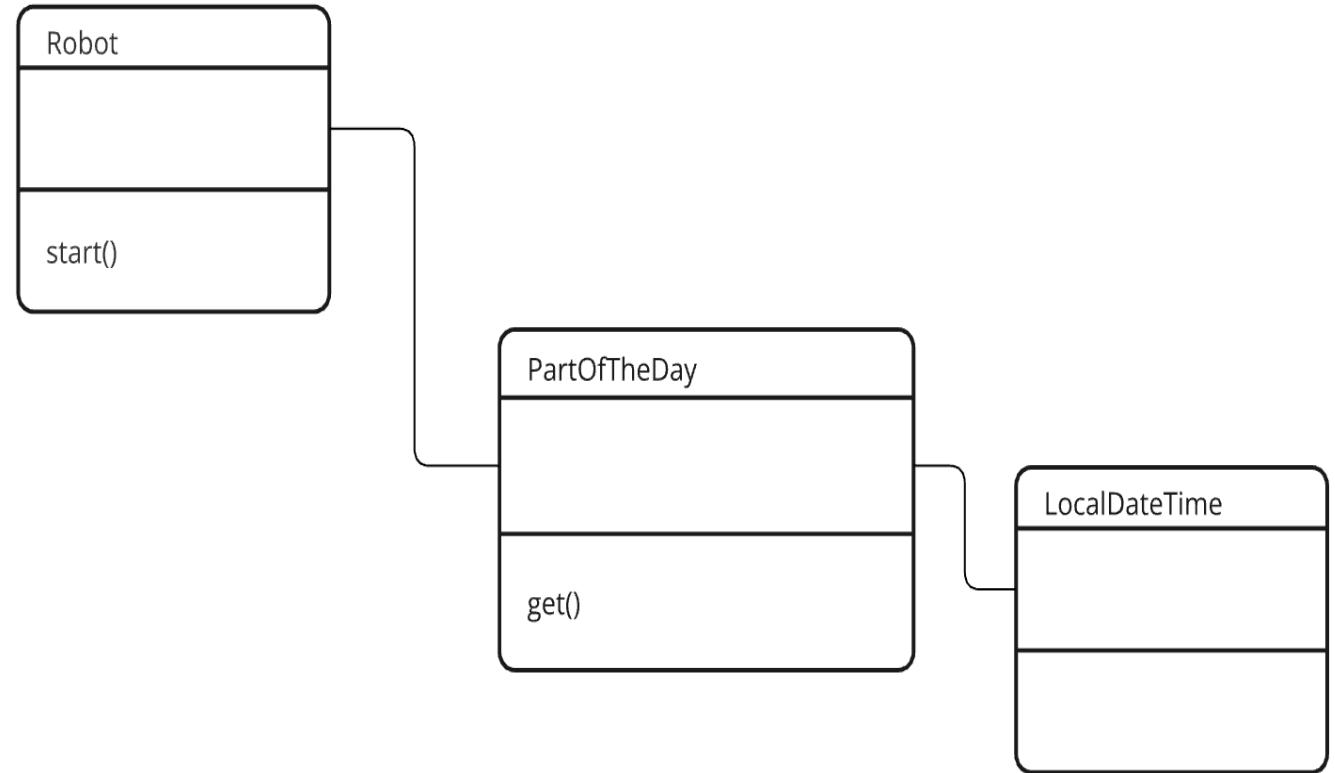
```
GreetCustomerTests.java
8 import static org.hamcrest.Matchers.equalTo;
9 import static org.hamcrest.Matchers.is;
10
11 public class GreetCustomerTests {
12     //Flaky test
13     @Test
14     public void should_Greet_Customer_AsPer_TimeOfDay(){
15         String expectedGreeting = "Good Evening";
16         StartRobotUseCase greeting = new StartRobotUseCase();
17         assertThat(greeting.start(), is(equalTo(expectedGreeting)));
18     }
19 }
20
```

Use Case – Robo Helper Start



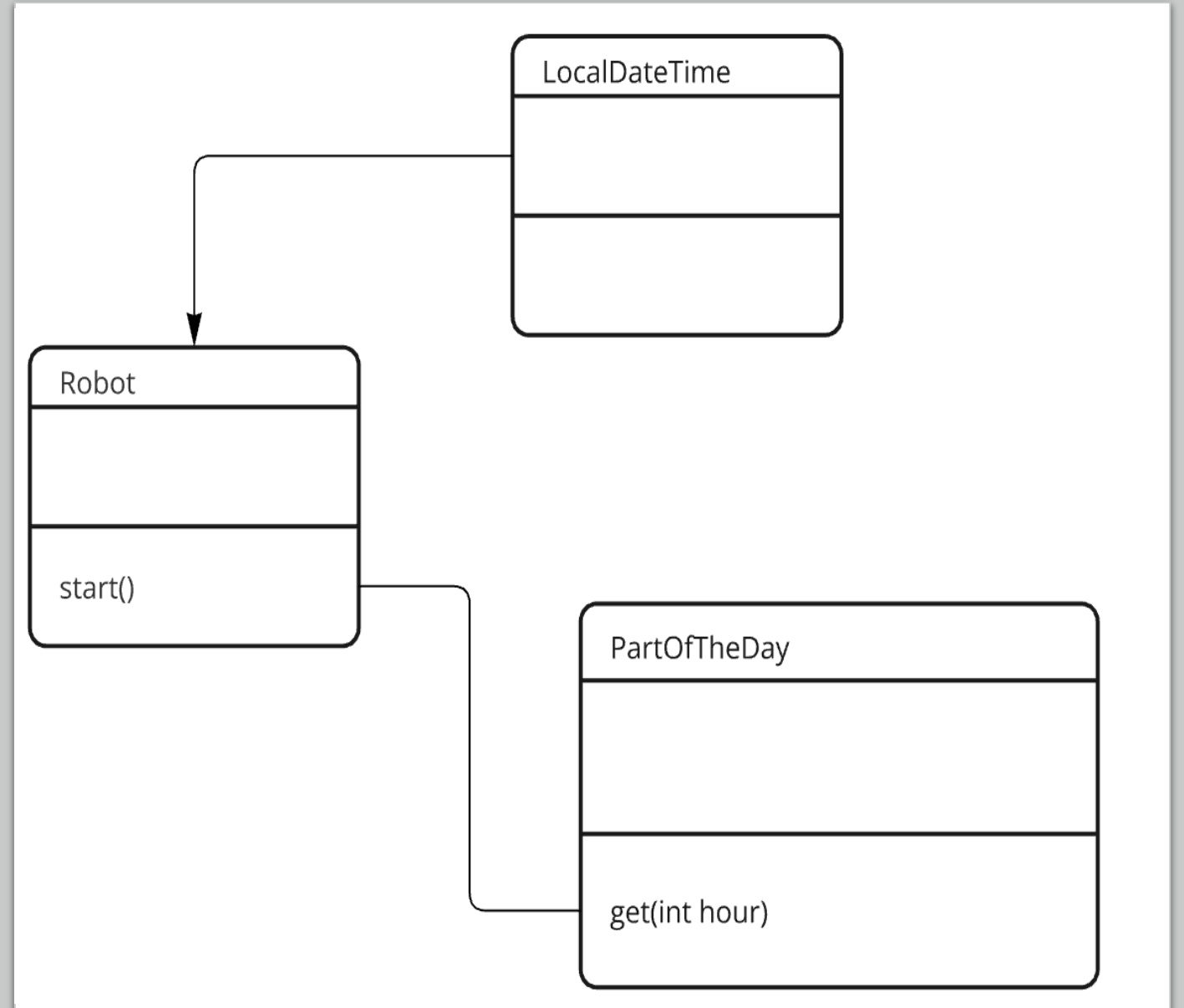
Challenges

- System time (Data source) tightly coupled within PartOfTheDay class
 - Can't reuse this class with any other source of date/time
- It violates SRP – Fetches time and processes it
- Unit test cannot be deterministic



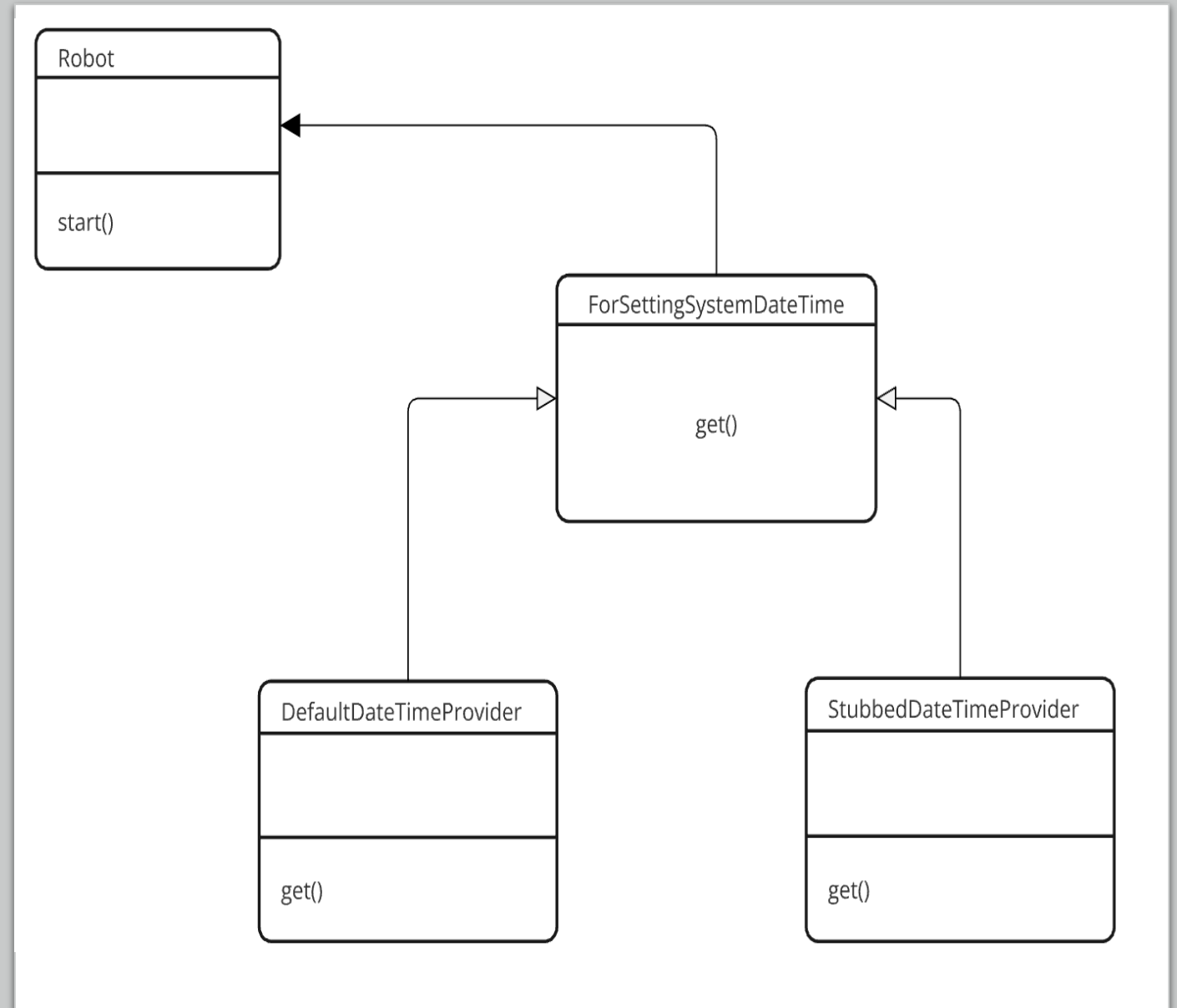
Consideration

Pass LocalDateTime as
parameter to TimePeriod



Inversion of control

- Abstract concrete implementation to an interface
- Refactor to inject a test double implementation to Robot



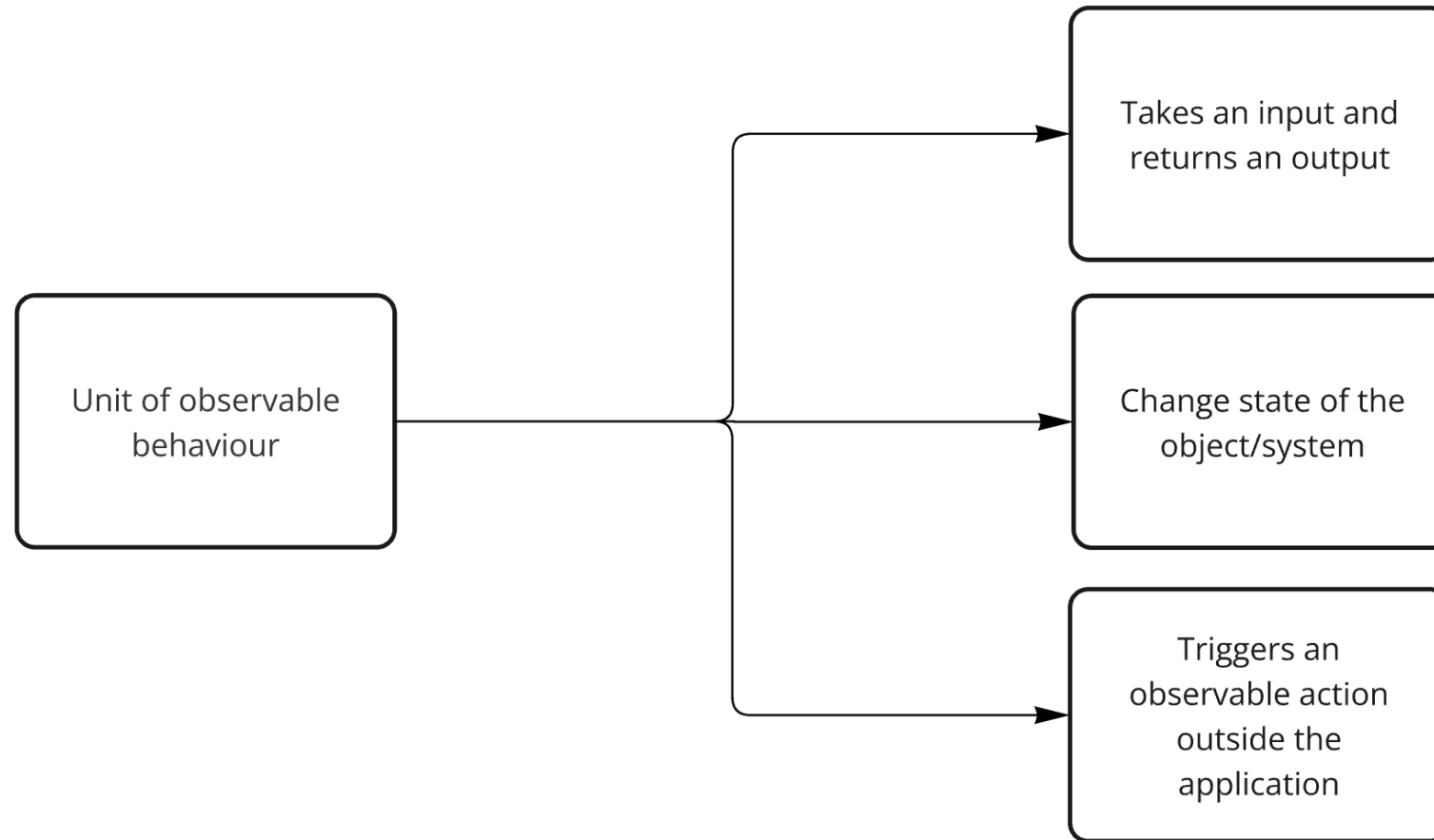
Good: Mocking tools lets you live with complexity

Bad: Mocking tools lets you live
with complexity

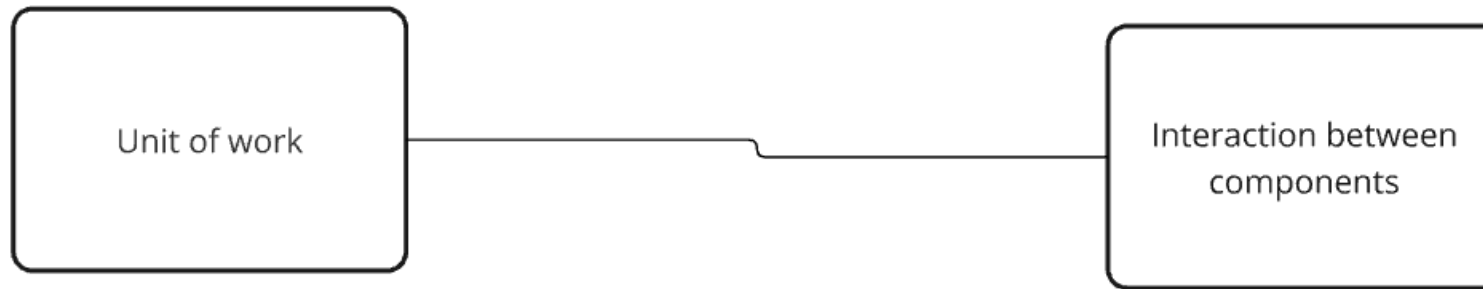
Good practices for unit testing

- Deterministic
- Automated
- Quick to run
- Should not fail when code's internal structure changes
- Should fail when the behavior of code changes
- Cheap to read, write and change
- Tests should reduce (and not introduce) risk. Example: Private method made public for sake of testing
- Tests should be isolated and not dependent on each other

Unit of work(Behavior)



Unit of work (interaction)



Changing state or forwarding actions requires internal handling

- Application depends on another component for inputs (indirect input)
- Application produces certain outputs that cannot be tested (indirect outputs)

Test doubles

Types of test doubles

Stubs

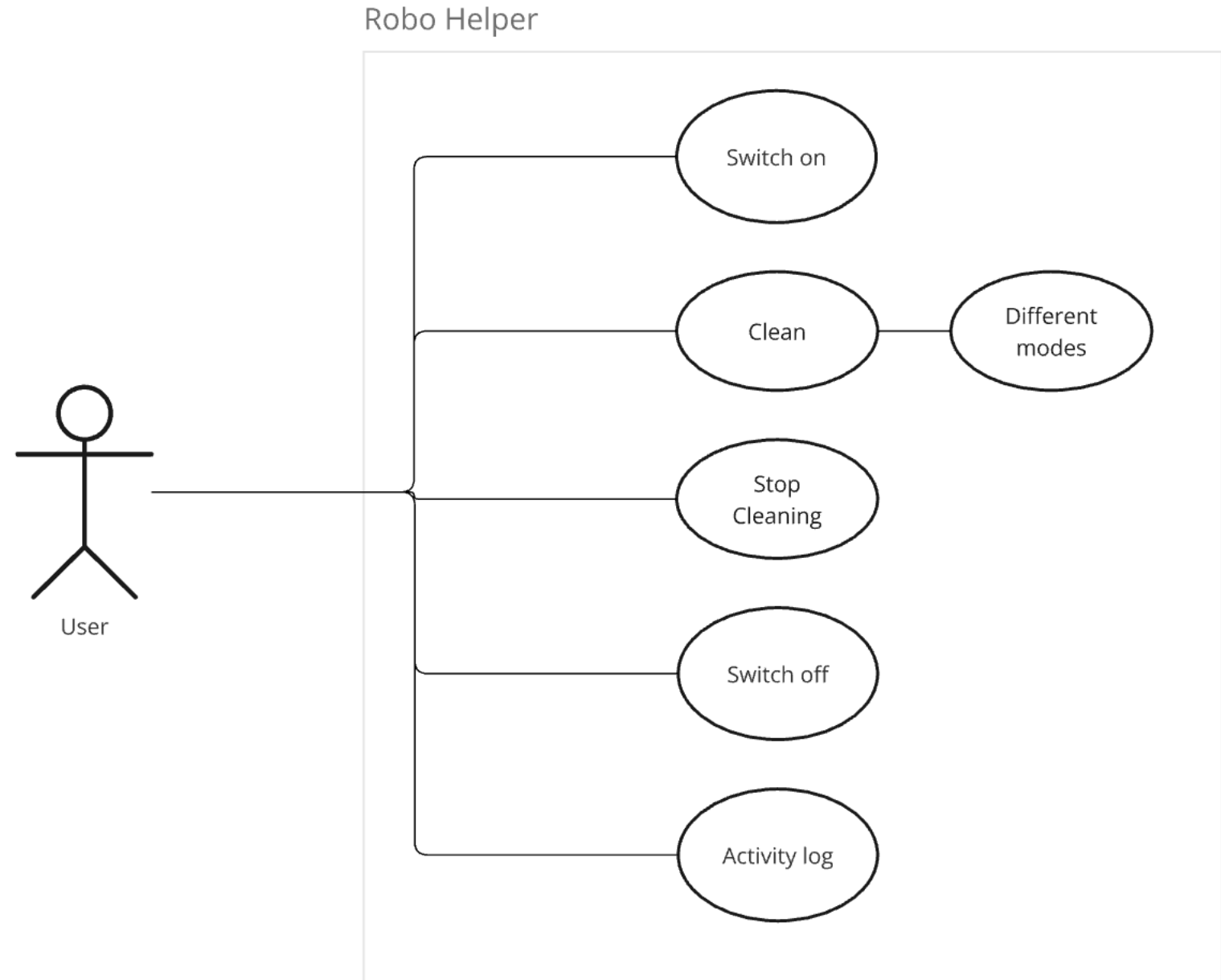
Dummy

Fakes

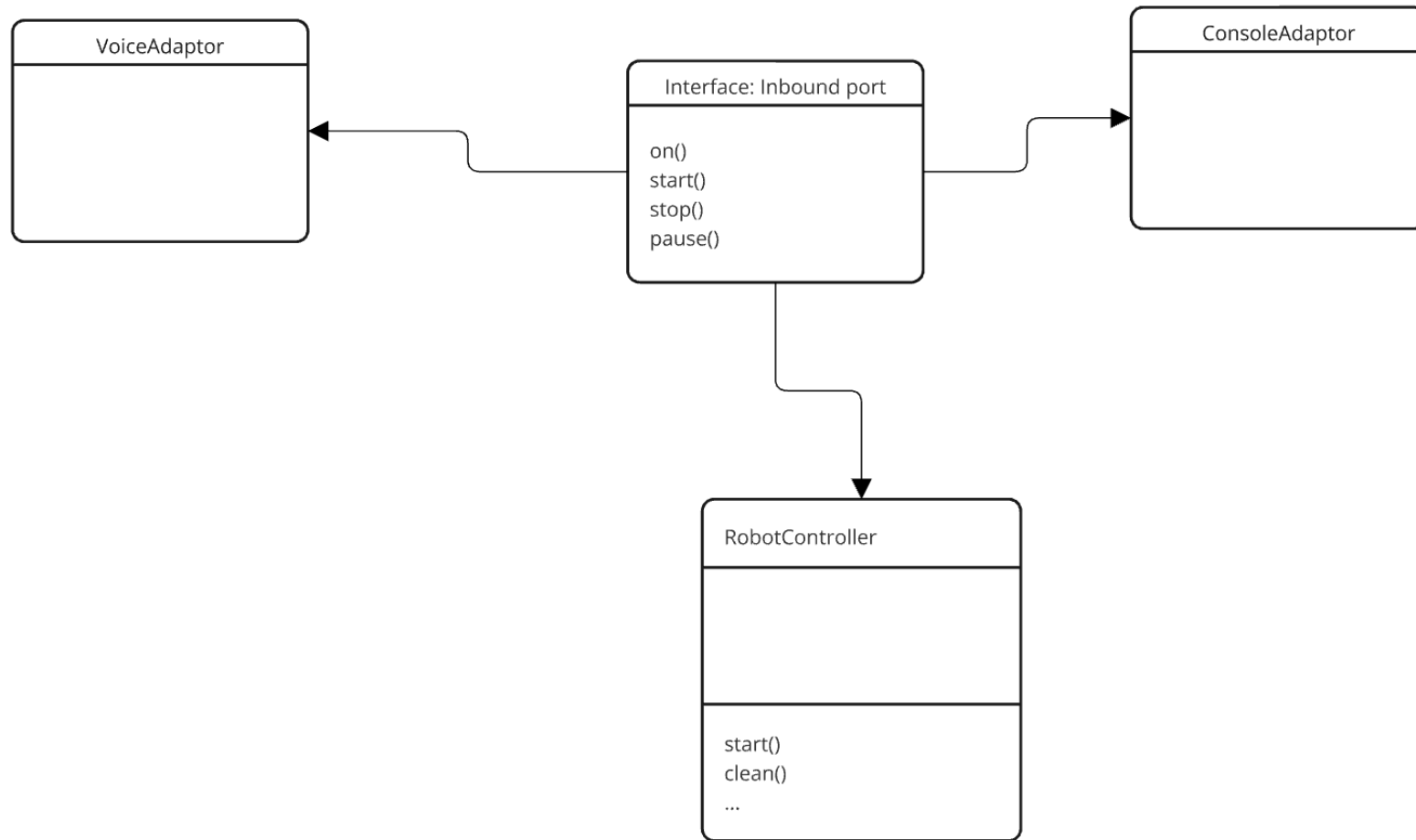
Mock

Spies

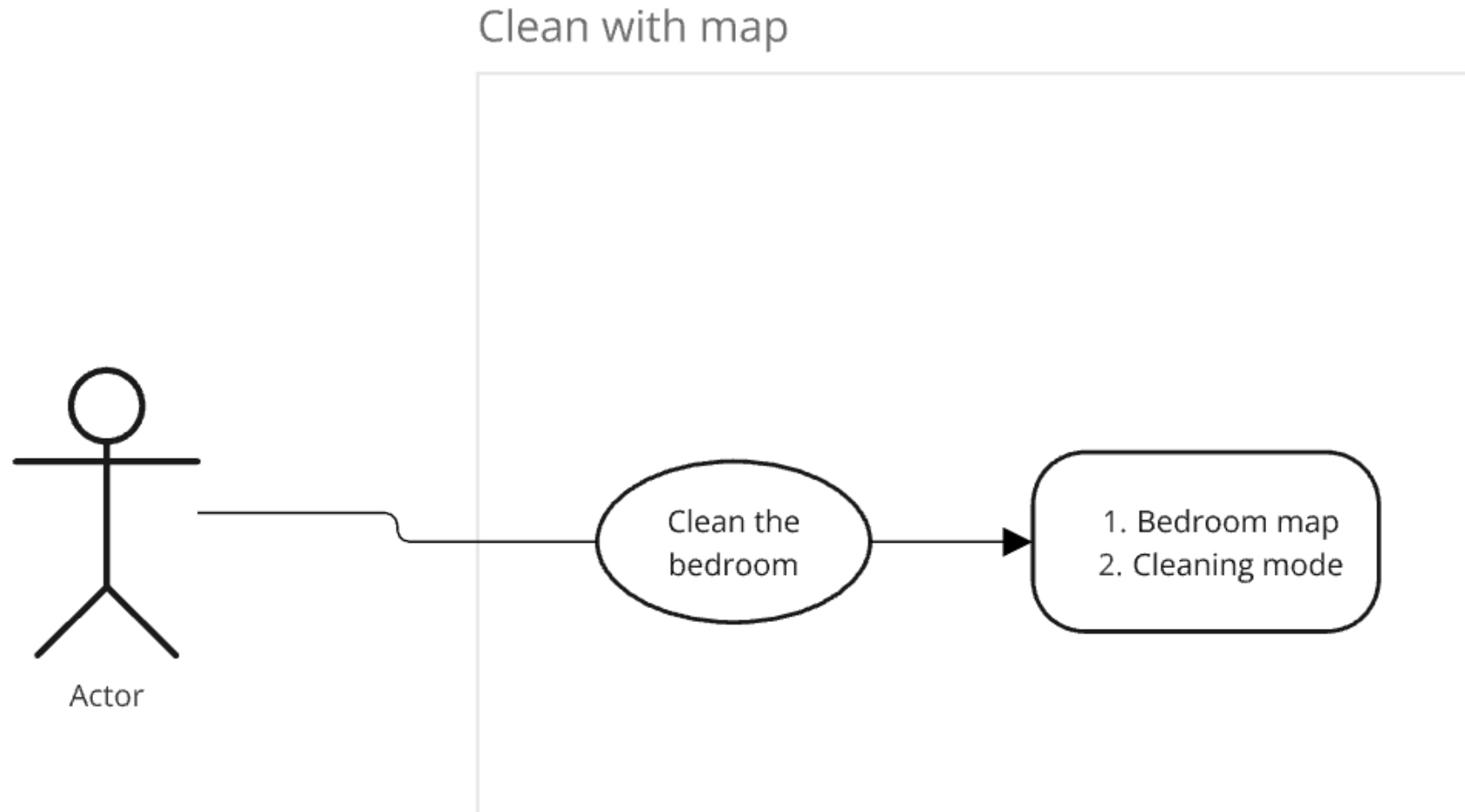
Use cases



Interaction with robot



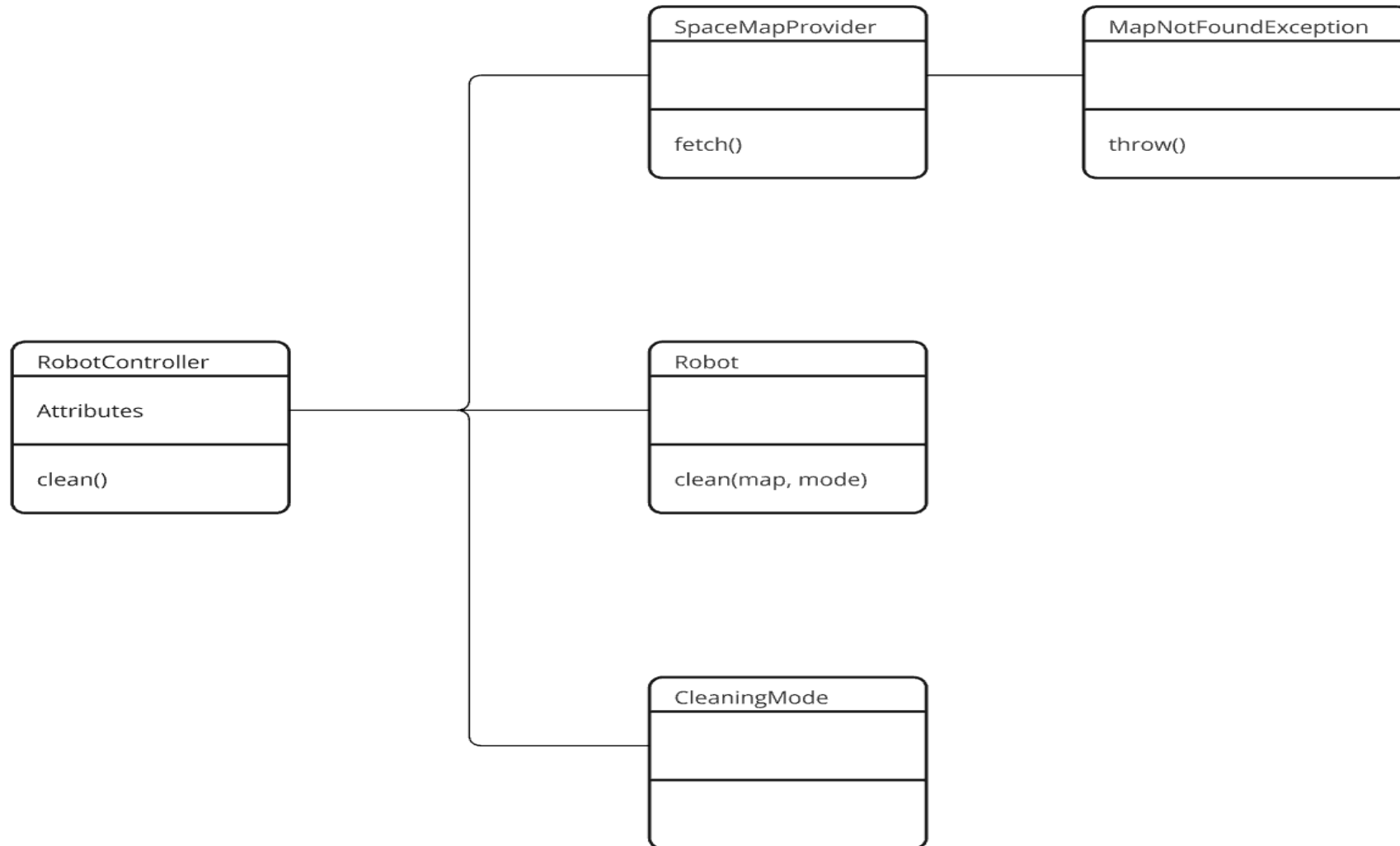
Use case – Clean with a given map



Scenarios – Cleaning pre-requisites

- Use case: Clean with provided map
 - Given a map is provided
 - Action: Start cleaning
 - Verification: Robot status should be cleaning
- Use case: Throw exception when map is not found
 - Given a map is not found
 - Action: Start cleaning
 - Verification: Exception thrown that Map is not found

Code Structure



Stub

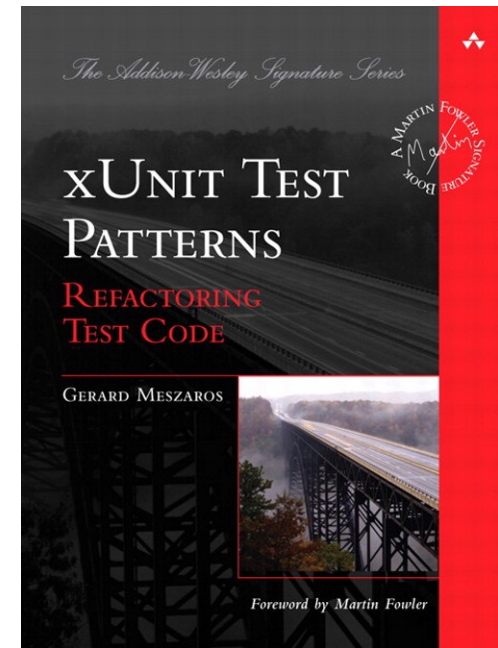
A filler object to satisfy behavioral needs of code through canned responses. Used as an alternative to expensive integration testing for testing dependency responses

Benefits of Stubs

- **Isolation:** The SUT is isolated from external dependencies, making tests faster and more reliable.
- **Controlled Data:** You can control the output of the stub to test various conditions without involving the real dependency.
- **Simplicity:** Stubs are simple to implement and provide a lightweight way to handle external dependencies in tests.

Types of stubs

- Responder- Returns a canned response
- Saboteur – Returns a canned exception



Scenarios – Cleaning pre-requisites

- Use case: Clean with provided map
 - Given a map is provided
 - Action: Start cleaning
 - Verification: Robot status should be cleaning
- Use case: Throw exception when map is not found
 - Given a map is not found
 - Action: Start cleaning
 - Verification: Exception thrown that Map is not found
- Challenge: How to deal with CleaningMode?
 - Not required as a pre-requisite testing, but expected in instantiation

Dummy

A filler object to satisfy structural needs of code under test.
Used as a test helper and doesn't impact the behavior of code under test.

Benefits of Dummy

- **Simplifies** test setup
- Keeps the test **focused** on the relevant behavior
- **Avoids unnecessary** logic or operations
- **Speeds up** test execution
- **Reduces dependencies**, making tests more isolated
- **Improves readability** of test code

Dummy is not same as Null object

- A Dummy object is not used by the SUT, so its behaviour is irrelevant.
By contrast, a Null Object is used by the SUT
 - The null object is designed to do nothing.
 - Though may direct the logical flow

Scenarios – Cleaning with different modes

- Use case: Default cleaning mode
 - Given the cleaning mode is default and map is provided
 - Action: Start cleaning
 - Verification: Robot should clean once for the given map
- Use case: Deep cleaning mode
 - Given the cleaning mode is deep cleaning and map is provided
 - Action: Start cleaning
 - Verification: Robot should clean twice for the given map
- Challenge: How do I know if Robot cleaned once or twice? And for what map?

Spy

Spy is a stub that also records the interaction with caller objects

Benefits of Spy

- **Verifies** method calls and interactions.
- Tracks internal **method invocations** without breaking **encapsulation**
- Allows for **non-intrusive** observation of object behavior
- Facilitates partial mocking for specific methods
- Suitable for **interaction-based** testing
- Useful to test **indirect outputs**

Mock

Mock is a spy that verifies the interactions with caller object and fails if it doesn't meet expectations. Unlike Spies, Mock doesn't call the real implementation

Mock vs Spy

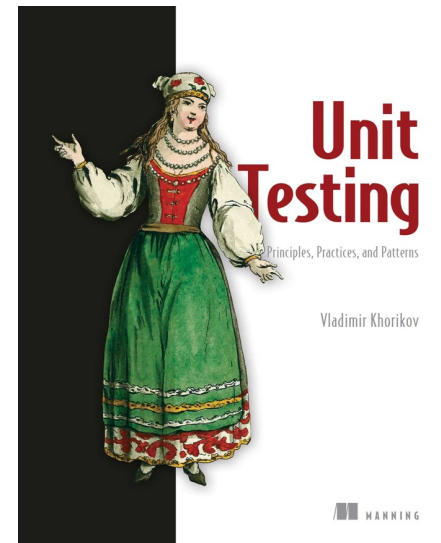
Feature	Mock	Spy
Definition	A mock is a fully simulated object that replaces the real object in a test. It doesn't call the real methods of the object.	A spy is a partial mock that wraps around a real object, allowing real methods to be called, but you can still verify interactions.
Purpose	Simulate behaviour of a dependency and verify specific method calls without affecting the actual implementation.	Track the behavior of a real object and verify interactions, while still calling the real methods.
Verification	Verification is primarily focused on method calls (how many times, with what arguments, etc.)	Records the interactions but does not verify. Provides a way to make assertions
Behaviour Simulation	Requires manual simulation of behavior using stubbing, defining how methods should act.	Can rely on the real behavior of the object, with options to override specific methods.
Complexity	Slightly simpler to implement since real methods aren't executed. Behavior is fully controlled by the test setup.	A bit more complex since it involves both executing real methods and verifying interactions.

Fake

A Fake object has a working implementation and behavior. Though meant for testing purpose and not suitable for production. For instance, using in-memory database instead of connecting to real database instance for operations

Benefits of using fake

- Closer to real implementation
- Not tied to structure of the implementation
- Helps focuses on behaviour instead of structure of program



Summary

- Unit tests help
 - Build developer confidence in code
 - Makes future changes easier
 - Helps to improve software quality
 - Helps to understand the system behaviour
 - Strengthens deployability of the code
- Use Test Doubles Wisely
 - Fakes: Simulate realistic behavior (e.g., in-memory databases).
 - Stubs: Provide fixed responses for isolation.
 - Mocks: Verify interactions and method calls.
 - Spies: Record method call details (e.g., call counts).
 - Dummy: As a test helper

Q & A

Let's connect

- Linked In: <https://www.linkedin.com/in/ritesh-mehrotra/>
- Subscribe to my blog: <https://neatstack.substack.com>