Mocking

ERIN KEITH

Goals

1. Introduce Mocking
2. Explore examples

Do *not* test

A testing unit should focus on one tiny bit of functionality and "prove it correct".

- Unit tests are simple and localized
- They should **NOT** involve
  - multi-threading
  - I/O
  - database connections
  - web services
  - these would be considered integration tests
- They should be fast
- They run in isolation
- Order shouldn't matter
- They should not depend on global state

Testing continued

After you've pulled out as much of that as you can, there may still be code which needs to be tested

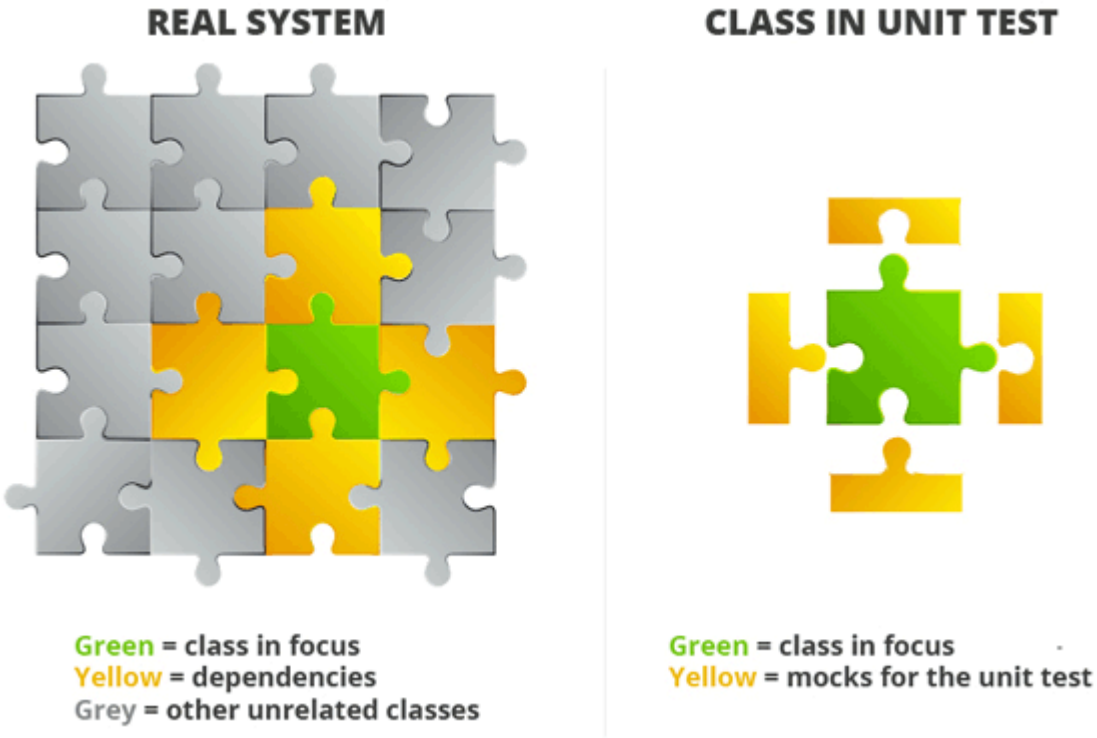
- but depends on libraries or modules from outside of your project
- and is therefore out of your control

Examples

- File IO
- API calls
- Database connections

Mocking

Allows us to “fake” the behavior of an object our code depends on.



Mocking

Allows us to “fake” the behavior of an object our code depends on.

- add fake object attributes and set their values
- add fake return values for object functions that are called
  - or exceptions
- verify that the mock object’s function(s) called

Mocking

Allows us to "fake" the behavior of an object our code depends on.

```
from datetime import datetime

def is_weekday():
    today = datetime.today()
    # Python's datetime library treats
Monday as 0 and Sunday as 6
    return (0 <= today.weekday() < 5)

# Test if today is a weekday
assert is_weekday()
```

Mocking

```
from datetime import datetime
from unittest.mock import Mock

# Save a couple of test days
wednesday = datetime(year=2025, month=1, day=1)
sunday = datetime(year=2025, month=1, day=5)

# Mock datetime to control today's date
datetime = Mock()

datetime.today.return_value = wednesday
# Test Wednesday is a weekday
assert is_weekday()

datetime.today.return_value = sunday
# Test Sunday is not a weekday
assert not is_weekday()
```

Patching

Allows us to "fake" the behavior of an module our code depends on.

- can be used as a decorator or context manager

Imagine we have a project that we want to test with the following structure:

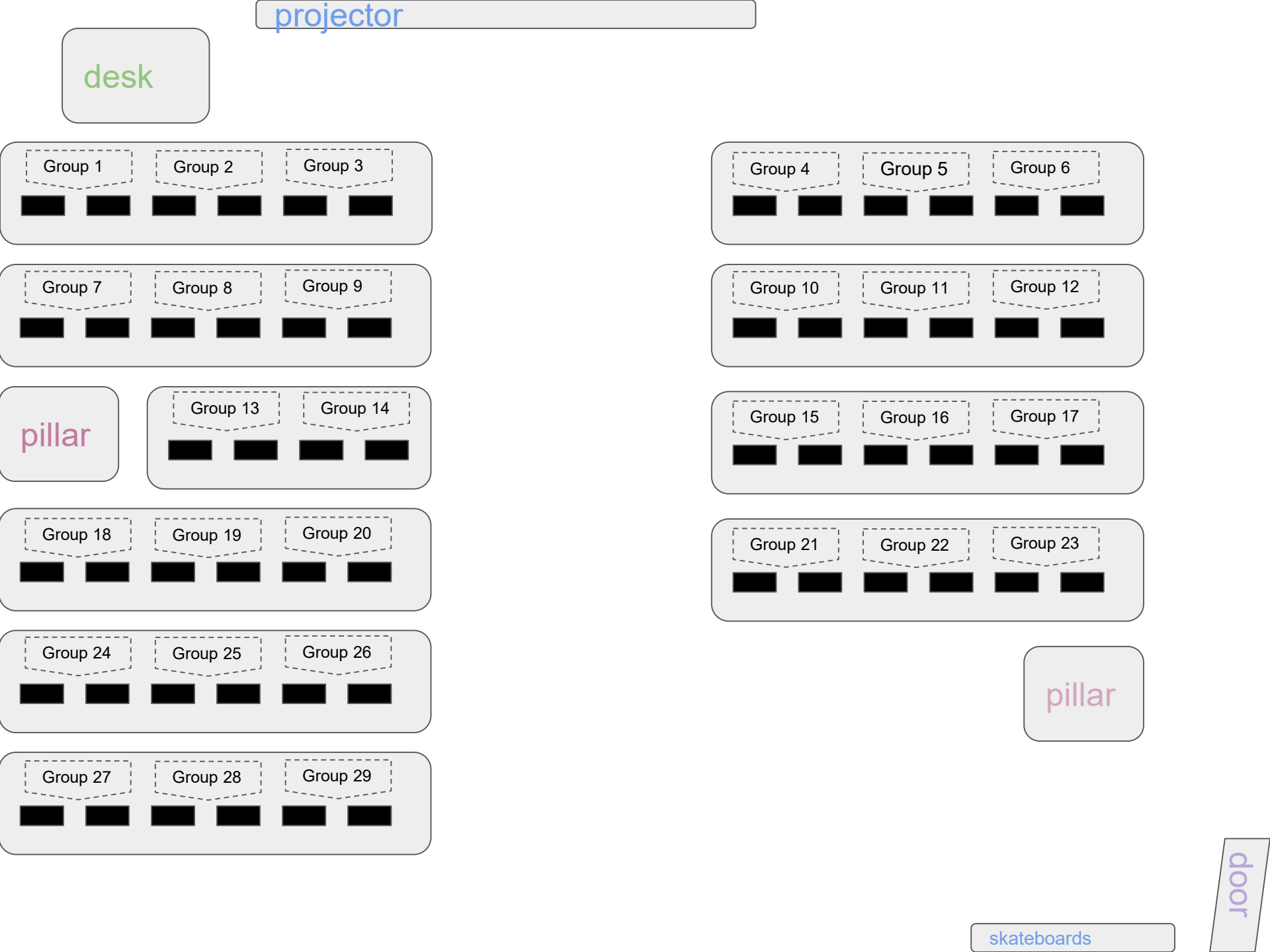
```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using `patch()`. The problem is that when we import module b, which we will have to do when it imports `SomeClass` from module a. If we use `patch()` to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```


projector
desk
Group 1
Group 2
Group 3
Group 4
Group 5
Group 6
Group 7
Group 8
Group 9
Group 10
Group 11
Group 12
pillar
Group 13
Group 14
Group 15
Group 16
Group 17
Group 18
Group 19
Group 20
Group 21
Group 22
Group 23
Group 24
Group 25
Group 26
pillar
Group 27
Group 28
Group 29
door
skateboards