

CS 4530: Fundamentals of Software Engineering

Lesson 5.3 Testing Systems

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
 - Explain why you might need a "test double" in your testing
 - Explain the differences between different kinds of test "doubles" such as "stubs, mocks, spies, fakes"

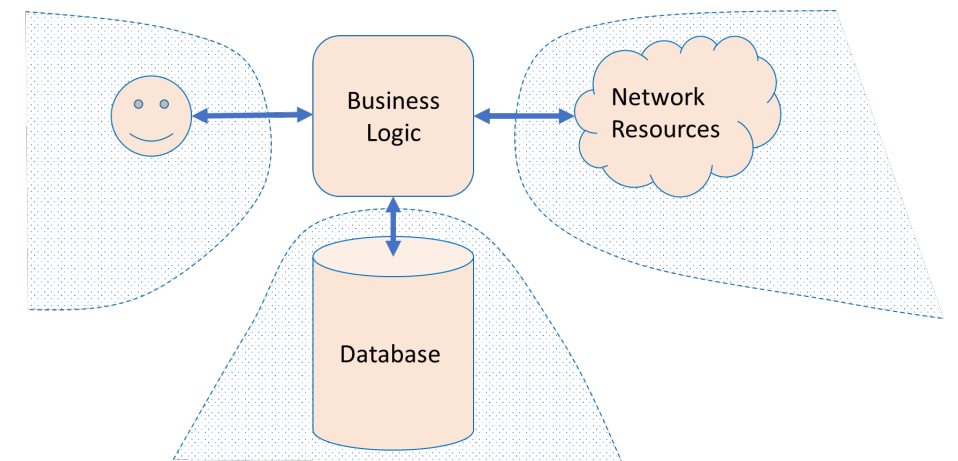
Review: What is the purpose of Test Suite?

- Test Driven Development
 - Does the SUT satisfy its specification? (“functional testing”)
- Regression Test
 - Did something change since some previous version?
 - Prevent bugs from (re-)entering during maintenance.
- Acceptance Test
 - Does the SUT satisfy the customer (requirement testing)
 - Validation: Are we building the right system ?

These purposes are copied from Lesson 5.2

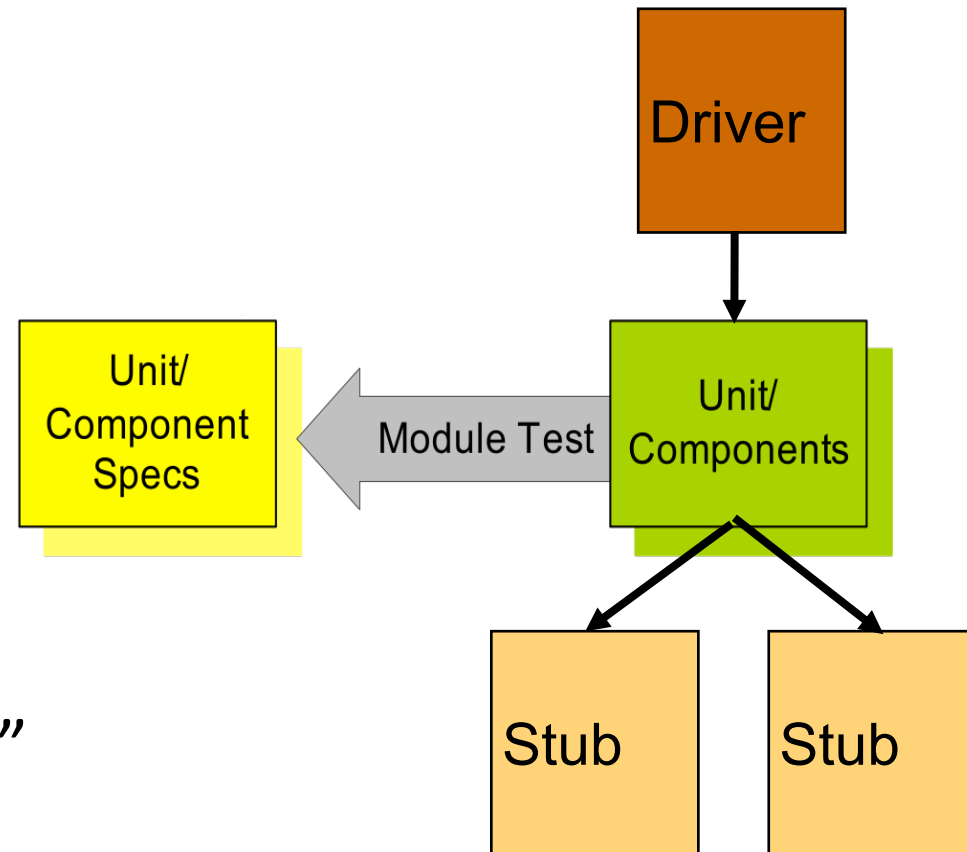
Large Systems are Hard to Test

- Database component
 - Contents may need to reflect/simulate real-world;
 - Data may be expensive/proprietary/confidential.
- Network connections
 - "Real" connections may be slow/flaky/disrupted;
 - Resources may have changed since test was written.
- Environment
 - Interactions with OS, locale or other software.
- Human actors
 - Ultimately unpredictable.



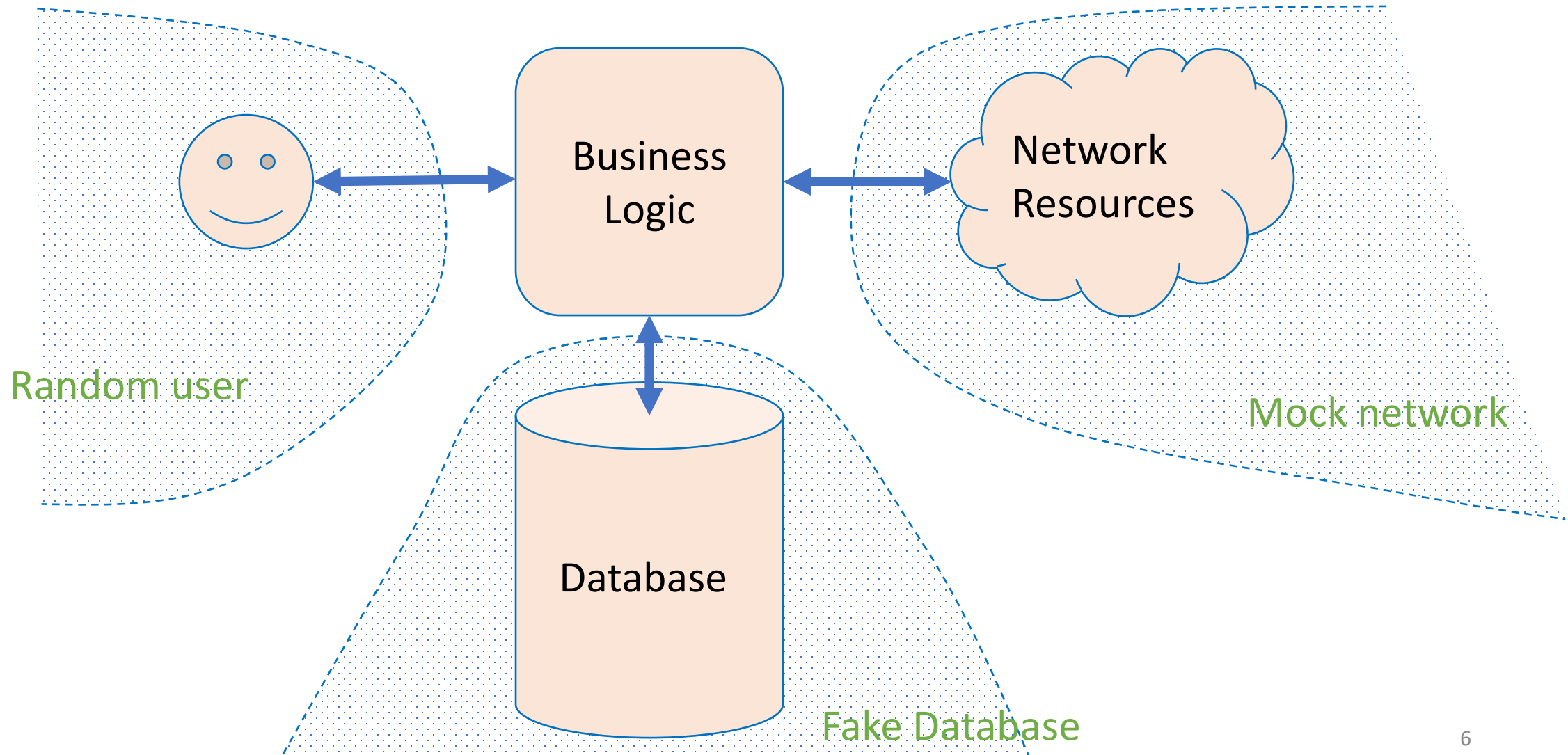
Unit Testing is not sufficient

- You are used to using Drivers and Stubs in your tests

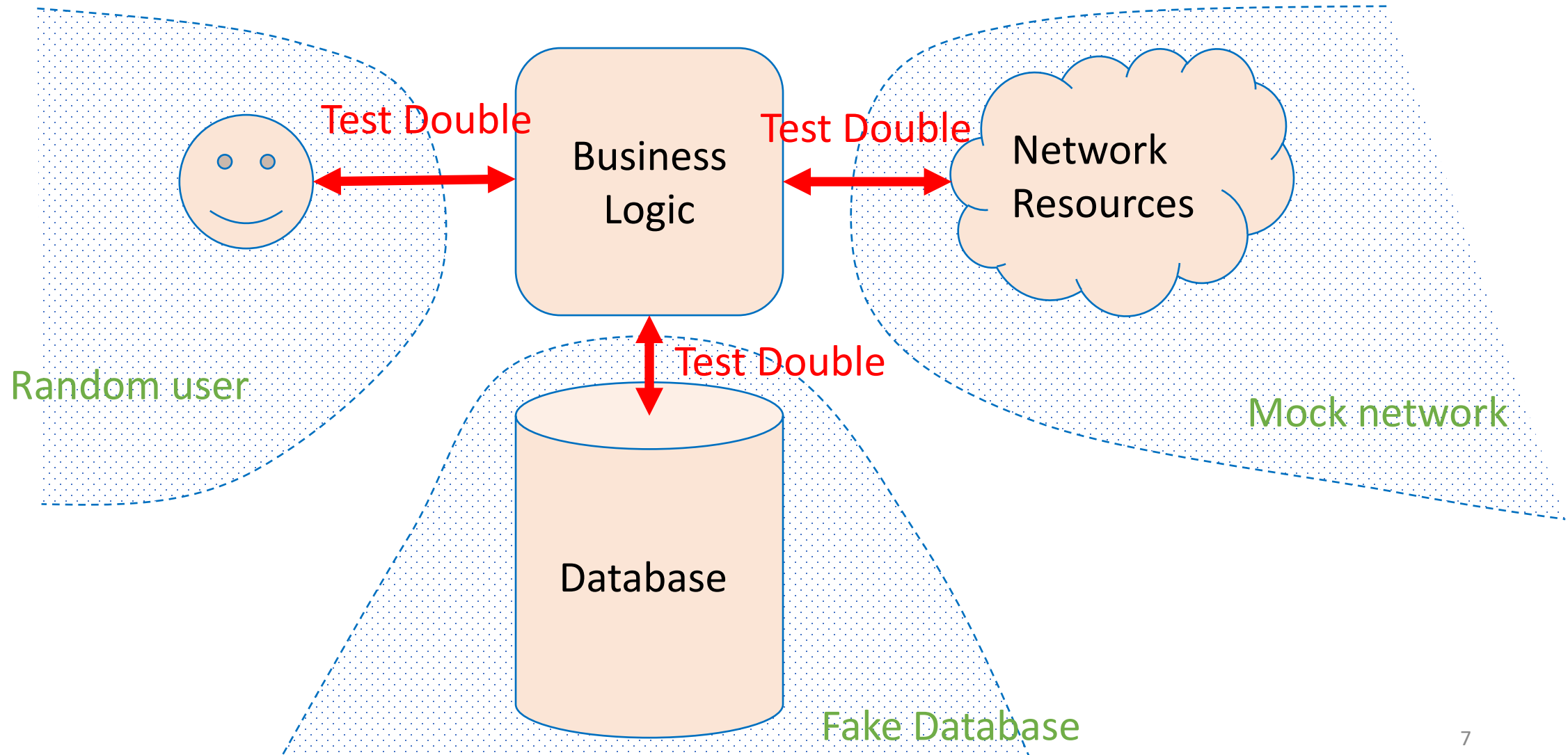


- Overall systems are “a little more” complicated

Test Doubles replace uncontrollable pieces of the environment



What are Test Doubles?



Test Stub is a Double that just supplies the same interface

- Supply an object with the same interface:
 - Same methods;
 - Default result values (i.e., **canned answers**).
- The stub gets the test to run:
 - If the client blindly uses the stub, it can proceed;
 - If the client expects something specific from the object, the test will likely fail.

Test Stub Example

```
final class Service {  
    public function doSomething(UserModelInterface user): Int {  
        /* Do things */  
        return user.uuid;  
    }  
}  
  
final class ServiceTest extends TestCase {  
    public function testDoSomething(): void {  
        // The service needs a implementation `UserModelInterface`.  
        String uuid = (new Service()).doSomething(new UserStub());  
        self.assertStringContainsString('0000-000-000-00001', uuid);  
    }  
}  
  
interface UserModelInterface {  
    public function getUuid(): String;  
}  
  
final class UserStub implements UserModelInterface {  
    public function getUuid(): String {  
        return '0000-000-000-00001';  
    }  
}
```

getUuid() is a
stub

Sometimes Test Stub is not enough

- You might want your stub to do at least two more things:
 1. Remember how the stub was used; (“memory”)
 2. Program the responses of the stub for different situations.

Test Spy is a stub that remembers how the object was called

- Test can check what happened earlier;
 - For example: a particular method should be called
 1. First with parameters “foo” and 42;
 2. Then with parameters “quux” and -88.
- A spy can be useful in conjunction with the “real” environment:
 - What was sent on the network?
 - How many times a problem was logged?
 - What was inserted in the database?
- But most often used with a “mock.” (we will discuss this later)

Spy
“remembers”

Test Spy Example

```
interface Logger {  
    public function log(String message): void;  
}
```

```
final class LoggerSpy implements Logger {  
    public Array messages = [];  
    public function log(string message): void {  
        this.messages[] = message;  
    }  
}
```

```
final class UserNotifier {  
    public function __construct(private Logger logger) {}  
    public function registerUser(UserModelInterface user): void {  
        this.logger.log("Notifying the user: {user.name()}");  
        // ...  
    }  
}
```

```
final class UserNotifierTest extends TestCase {  
    public function testLogMessage(): void {  
        LoggerSpy logger = new LoggerSpy();  
        UserNotifier notifier = new UserNotifier(logger);  
        User user = new User(name = 'Jesus');  
        notifier.registerUser(user);  
        self.assertStringContainsString(  
            "Notifying the user: {user.name()}",  
            first(logger.messages)    );    }}
```

Logger
"remembers"
messages

Test Mock is a Double that has Scripted results

- A test mock has scripted results:
 - If such-and-such a method is called
 - return some particular value.
- A complex mock can have many scripts:
 - Multiple methods;
 - Different results for subsequent calls.
- Useful mocking assumes we know how mocked object will be used.
- If a “mock” has real logic, it becomes a “fake” (we will discuss this later).

Mock has “scripted answers” and is used for “behavior verification”

Jest supports Mocks

Jest's Mock API: <https://jestjs.io/docs/mock-function-api>

- Replacing TwilioVideo with Mock

```
const mockTwilioVideo = mockDeep<TwilioVideo>();  
jest.spyOn(TwilioVideo, 'getInstance').mockReturnValue(mockTwilioVideo);
```

You will see more of
these in HW3

- Jest Tests can be written

```
it('should use the coveyTownID and player ID properties when requesting a video token',  
  async () => {  
    const townName = `FriendlyNameTest-${nanoid()}`;  
    const townController = new CoveyTownController(townName, false);  
    const newPlayerSession = await townController.addPlayer(new Player(nanoid()));  
    expect(mockTwilioVideo.getTokenForTown).toBeCalledTimes(1);  
    expect(mockTwilioVideo.getTokenForTown).toBeCalledWith(townController.coveyTownID, newPlayerSession.playerID);  
  });
```

Here is another Example of Mock /1

```
describe('conversationAreaCreateHandler', () => {
```

```
    const mockCoveyTownStore = mock<CoveyTownStore>();
```

```
    const mockCoveyTownController = mock<CoveyTownController>();
```

```
    beforeAll(() => {
```

```
        // Set up a spy for CoveyTownStore that will always return our mockCoveyTownStore as the  
        singleton instance
```

```
        jest.spyOn(CoveyTownStore, 'getInstance').mockReturnValue(mockCoveyTownStore);  
    });
```

```
    beforeEach(() => {
```

```
        // Reset all mock calls, and ensure that getControllerForTown will always return the same  
        mock controller
```

```
        mockReset(mockCoveyTownController);
```

```
        mockReset(mockCoveyTownStore);
```

```
        mockCoveyTownStore.getControllerForTown.mockReturnValue(mockCoveyTownController);
```

```
    });
```

```
    . . .
```

spying on
getInstance()
method

Here is another Example of Mock /2

.....

```
it('Checks for a valid session token before creating a conversation area', ()=>{
  const coveyTownID = nanoid();
  const conversationArea :ServerConversationArea = { boundingBox: { height: 1, width: 1, x:1, y:1 }, label:
nanoid(), occupantsByID: [], topic: nanoid() };
  const invalidSessionToken = nanoid();
  // Make sure to return 'undefined' regardless of what session token is passed
  mockCoveyTownController.getSessionByToken.mockReturnValueOnce(undefined);
  requestHandlers.conversationAreaCreateHandler({
    conversationArea,
    coveyTownID,
    sessionToken: invalidSessionToken,
  });
  expect(mockCoveyTownController.getSessionByToken).toBeCalledWith(invalidSessionToken);
  expect(mockCoveyTownController.addConversationArea).not.toHaveBeenCalled();
});
});
```

If SessionToken is invalid, don't call
addConversationArea()

Test Fake is a Mock with semi-real implementation

- A *fake* has an implementation of the object being replaced
 - A *low-fidelity* fake implements things partially
 - Enough to work for the test.
 - A *high-fidelity* fake implements most aspects:
 - Usually all functional aspects;
 - Usually not as efficiently or as scalable.
- The purpose of the fake is to avoid processes/network/cost:
 - So the test can be cheap and deterministic.
- Transcript Server you used in **Activity 4.1** was a Fake

Fake has
"semi-real
implementation"

How do you provide a Test Double for a User?

- To replace a user, we can program a “Bot”
 - Randomly use mouse, press buttons;
 - Arbitrary text;
 - Fast or slow.
- Smarter (“Fuzzing”)
 - Capture real actions;
 - Then make targeted mutations.
 - (This applies also to programs taking text input.)
- Expected result can only be imprecise:
 - e.g., “not crash” or “not leak secrets”.

Weaknesses of Test Doubles

- The Mock/Fake may not behave correctly
 - The test may assume wrong behavior;
 - Particularly an issue if original object changes
 - Mocks have to be maintained as well!
 - Solution: Test the mock/fake against a higher fidelity fake, or against the real thing.
- The SUT may use a different algorithm:
 - The Spies expect a particular usage of double;
 - The test is “brittle” because it depends on internal behavior of SUT;

Review: Learning Objectives for this Lesson

- You should now be able to:
 - Explain why you might need a "test double" in your testing
 - Explain the differences between different kinds of test "doubles" such as "stubs, mocks, spies, fakes"
- **For Further Reading**
 - Check out Martin Fowler's article, "Mocks Aren't Stubs" <https://martinfowler.com/articles/mocksArentStubs.html>
 - "xUnit Test Patterns: Refactoring Test Code" by Gerard Meszaros