CS 4530: Fundamentals of Software Engineering

Module 12.1: Testing Effectful Code

Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be prepared to:
  - Explain why you might need a test double in your testing
  - Use simple mocks and spies in your tests.

Remember: Assemble/Act/Assess

```
test('addStudent should add a student to the data
    // const db = new DataBase ()
    expect(db.nameToIDs('blair')).toEqual([])

    const id1 = db.addStudent('blair’);

    expect(db.nameToIDs('blair')).toEqual([id1])
  });
```

Assemble (and check that you’ve assembled it

Act (do the action that you are trying to test)

Assess: check to see that the response is correct

If the response is an answer, testing is (more or less) straightforward

f2c.test.ts

```
// System Under Test

/** given a temperature in Farenheit, returns the corresponding temperature in
 * Celsius
*/
function f2c(temperature:number): number {
    return (5/9*(temperature-32));
}

// Tests

describe ("tests for f2c", () => {
    test("32 F => 0 C", () => {
        expect(f2c(32)).toBe(0)
    })
    test("212 F => 100 C", () => {
        expect(f2c(212)).toBe(100)
    })
})
```

If you can look at the state of the object, it’s still easy (1)

clock1.test.ts

```
interface IPullingClock {
    reset():void      /** sets the time to 0 */
    tick():void       /** increments the time */
    getTime():number /** returns the current time */
}

class Clock1 implements IPullingClock { .... }

const c = new Clock1
```

If you can look at the state of the object, it’s still easy (2)

clock1.test.ts

```
describe("tests of Clock1", () => {
    test("after reset, clock should return 0", () => {
        c.reset();expect(c.getTime()).toBe(0)
    })
    test("after one tick, getTime should return 1", () => {
        c.reset(); c.tick()
        expect(c.getTime()).toBe(1)
    })
    test("after two ticks, getTime should return 2", () => {
        c.reset(); c.tick();c.tick()
        expect(c.getTime()).toBe(2)
    })
})
```

But what if you can’t look at its state?

- The action must have some visible effect on some other part of the system
- Look at the other part of the system
- Hopefully you can get access to the other part of the system.

If your code uses the observer pattern, you could supply your own observer

clockWithObserverPattern.test.ts

```
export interface IClockWithListeners {
    reset():void  // resets the time to 0
    tick():void  // increment time and notify all listeners
    // add a listener and initialize it with the current time
    addListener(listener:IClockListener):void
}

export interface IClockListener {
    // @param t - the current time, as reported by the clock
    notify(t:number):void
}

export class ProducerClock implements IClockWithListeners {
    // some implementation
}
```

Here is an observer you could use for testing.

clockWithObserverPattern.test.ts

```
import { IClockWithListeners, IClockListener } from "./clockWithObserverPattern";

class ClockListenerForTest implements IClockListener {
    private _time : number = 0
    constructor (private masterClock:IClockWithListeners) {
        masterClock.addListener(this)
    }
    notify (t:number) : void {this._time = t}
    getTime () : number {return this._time}
}
```

Now we can test using the custom observer

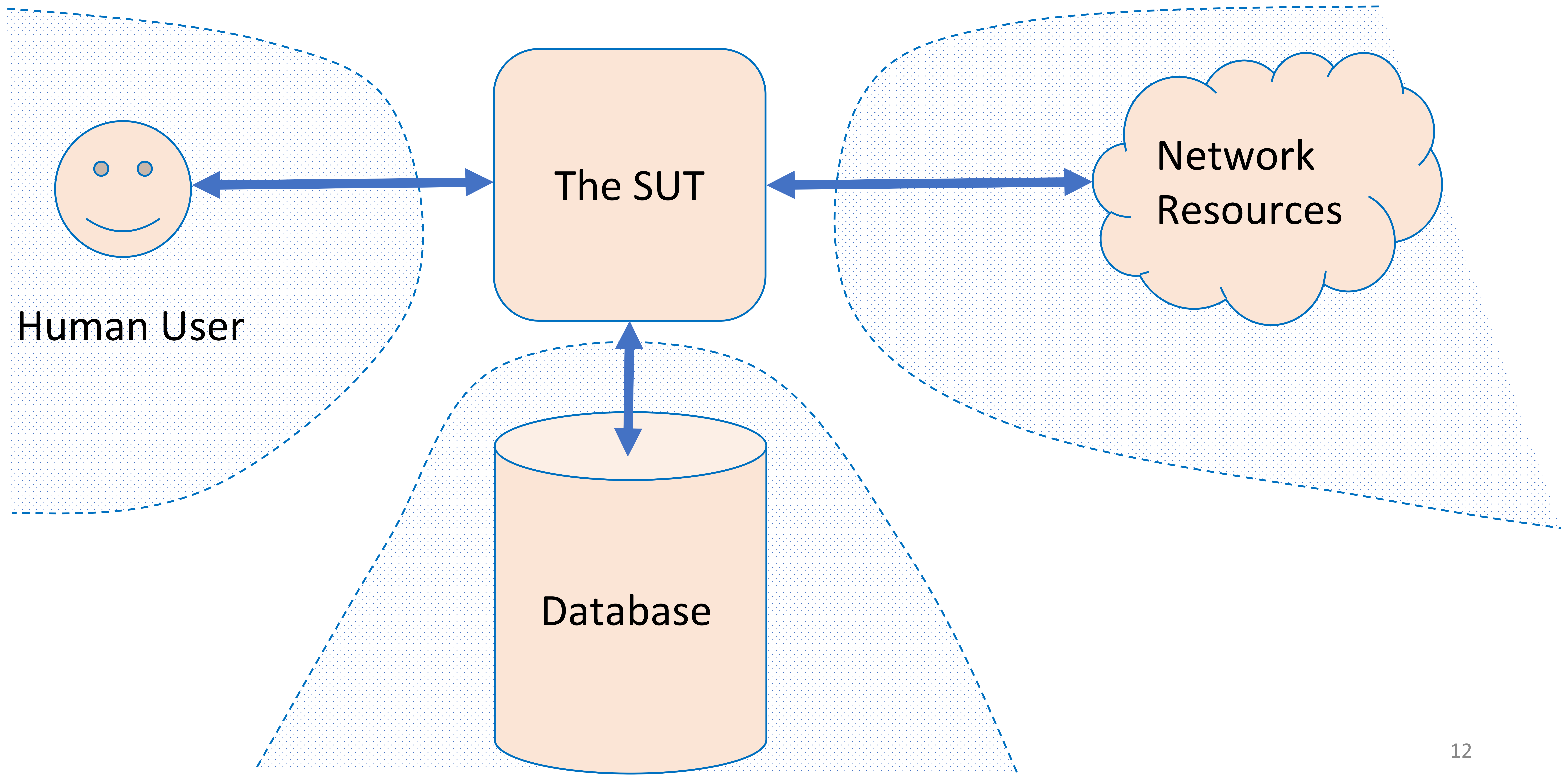
clockWithObserverPattern.test.ts

```
import { ProducerClock } from  "./clockWithObserverPattern";

const clock1 = new ProducerClock
const listener1 = new ClockListenerforTest(clock1)

describe("tests for ProducerClock", () => {
    test("after reset, listener should return 0", () => {
        clock1.reset()
        expect(listener1.getTime()).toBe(0)
      })
    test("after one tick, listener should return 1", () => {
        clock1.reset(); clock1.tick()
        expect(listener1.getTime()).toBe(1)
    })
    test("after two ticks, listener should return 2", () => {
        clock1.reset(); clock1.tick(); clock1.tick()
        expect(listener1.getTime()).toBe(2)
    })
})
```

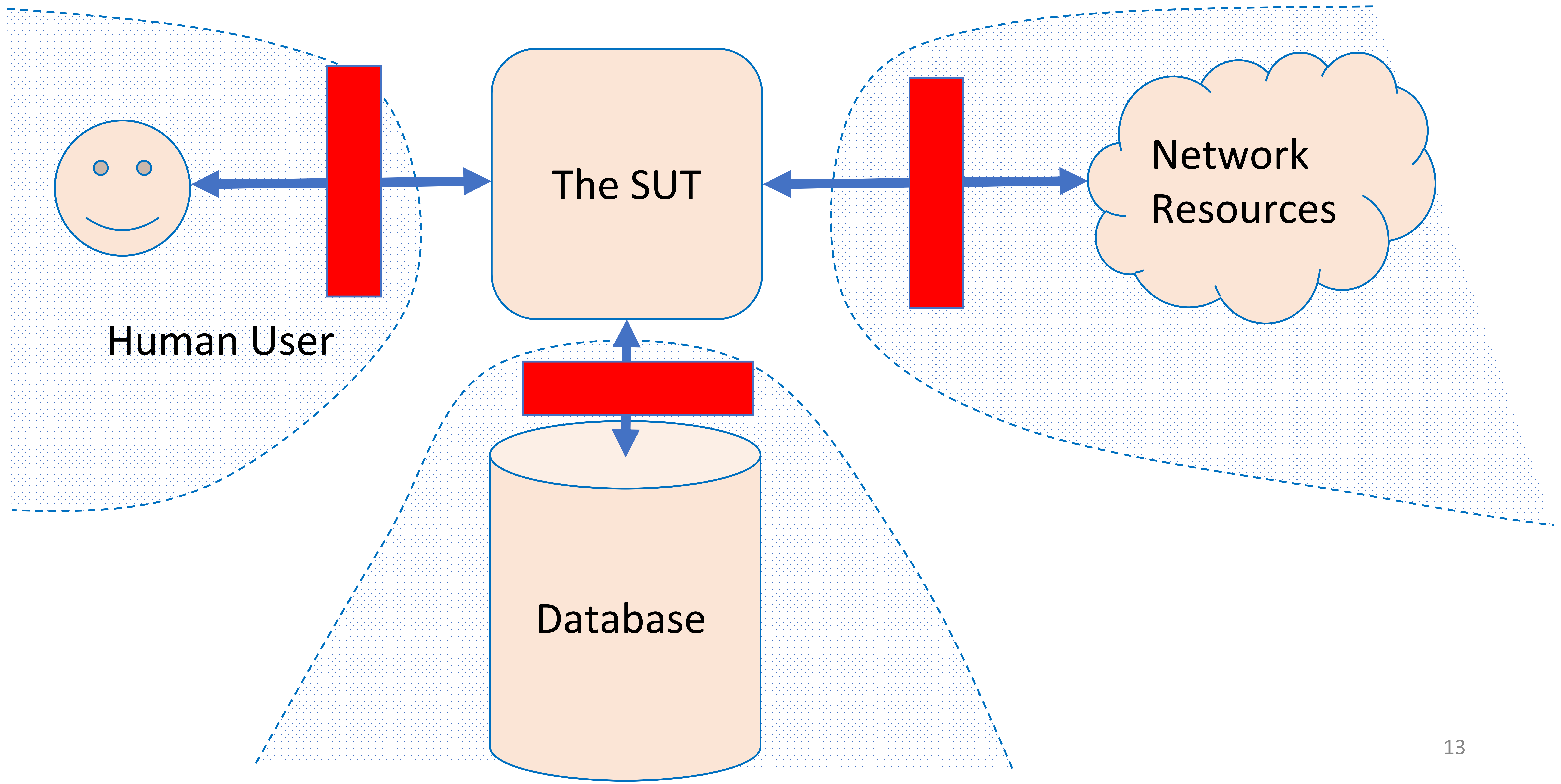
But what if you can’t do that?

- Existing code may have effects on other portions of the system, which you don’t control.

Your module may interact with uncontrollable things in the environment



Test doubles replace uncontrollable things with things that you do control



Test Doubles Intercept Calls to Methods

- Testing frameworks provide two common abstractions for doubles
  - The framework transparently modifies programs while running to intercept calls
- **Spies** invoke the original method, but record the parameters and call information
- **Mocks** do not invoke the original method
  - Default is to provide canned responses (Jest picks: `undefined`)
  - Also can provide a mock implementation to entirely replace the original method
- Other frameworks use terms like "fake" and "stub" for variants of these; we focus on Jest's features (spies, mocks)

A spy is a test double that monitors a real object call

- It remembers how the method was called, and what was returned;
  - For example: a particular method was called
    1. First with parameters "foo" and 42, and it returned 63
    2. Then with parameters "quux" and -88, and it returned "hark!"
- 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?

Spy
"remembers"

A mock is like a spy, but does not actually do the call

- It remembers how the method was called, ~~and what was returned;~~
  - For example: a particular method was called
    1. First with parameters "foo" and 42~~, and it returned 63~~
    2. Then with parameters "quux" and -88~~, and it returned "hark!"~~
- You can set up the mock to return what you want
  - Jest default is to return **undefined**

Simplest mock behavior in Jest

simpleMocks.test.ts

```
test("simplest mock behavior", () => {
    const mockFunction1 = jest.fn();

    const result1 = mockFunction1("17");
    const result2 = mockFunction1("42")

    expect(result1).toBeUndefined();
    expect(result2).toBeUndefined()

    expect(mockFunction1).toHaveBeenCalled();
    expect(mockFunction1).toHaveBeenCalledTimes(2);

    expect(mockFunction1).toHaveBeenCalledWith("17");
    expect(mockFunction1).toHaveBeenCalledWith("42")

});
```

Jest’s Mock API: https://jestjs.io/docs/mock-function-api

You can customize your mock in many ways

simpleMocks.test.ts

```
test("customizing mock functions", () => {

    // you can specify the the return value
    const mockFunction3 = jest.fn();
    mockFunction3.mockReturnValue("baz");

    expect(mockFunction3(17)).toBe("baz");
    expect(mockFunction3).toHaveBeenCalledWith(17);

    // or give the mock an implementation
    const mockFunction2 = jest.fn()
    mockFunction2.mockImplementation((n: number) => n + n);

    expect(mockFunction2(3)).toBe(6);
    expect(mockFunction2(14)).toBe(28)
    expect(mockFunction2).toHaveBeenCalledWith(3);
    expect(mockFunction2).toHaveBeenCalledWith(14);

    // you can also reset the mock's history
    mockFunction2.mockReset()
    expect(mockFunction2).not.toHaveBeenCalledWith(14);
});
```

Let’s mock the http client from the async module

echo.ts

```typescript
import axios from 'axios'

export async function echo(str: string) : Promise<string> {
    const res =
        await axios.get(`https://httpbin.org/get?answer=${str}`)
    return res.data.args.answer
}
```

Pattern: use **.spyon** to spy on a single method

echo.test.ts

```
import axios from 'axios'
import { echo } from './echo'

describe("tests for echo", () => {

    beforeEach(jest.resetAllMocks)

    test('just spying on a function runs the original', async () => {
        jest.resetAllMocks()
        const spy1 = jest.spyOn(axios, 'get')
        const str = '43'
        const correctURL = `https://httpbin.org/get?answer=${str}`
        await expect(echo(str)).resolves.toEqual(str);
        expect(spy1).toBeCalledWith(correctURL);
        expect(spy1).toBeCalledTimes(1)
        expect.assertions(3)
    })
```

Spying on a function runs the original

echo.test.ts

```
import axios from 'axios'
import { echo } from './echo'

describe("tests for echo", () => {

    beforeEach(jest.resetAllMocks)

    test('just spying on a function runs the original', async () => {
        jest.resetAllMocks()
        const spy1 = jest.spyOn(axios, 'get')
        const str = '43'
        const correctURL = `https://httpbin.org/get?answer=${str}`
        await expect(echo(str)).resolves.toEqual(str);
        expect(spy1).toBeCalledWith(correctURL);
        expect(spy1).toBeCalledTimes(1)
        expect.assertions(3)
    })
```

Pattern: add a mock response to turn a spy into a mock

echo.test.ts

```
test('mocking the http call doesn\'t actually do a live call', async () => {
        jest.resetAllMocks()
        const spy1 = jest.spyOn(axios, 'get')

        // have the mock return this
        const mockAnswer = '777'
        const mockResponse = { data: { args: { answer: mockAnswer } } }
        spy1.mockResolvedValue(mockResponse)  // don't run the original!

        const realInput = '43'  // put this in the URL
        const realQuery = `https://httpbin.org/get?answer=${realInput}`

        // 'echo' takes the realInput, but returns the mockAnswer,
        // so the http call must not have taken place
        await expect(echo(realInput)).resolves.toEqual(mockAnswer);
        expect(spy1).toBeCalledWith(realQuery);
        expect(spy1).toBeCalledTimes(1)
        expect.assertions(3)
    })
```

This pattern creates close coupling between the SUT and the test

echo.test.ts

```
test('mocking the http call doesn\'t actually do a live call', async () => {
        jest.resetAllMocks()
        const spy1 = jest.spyOn(axios, 'get')

        // have the mock return this
        const mockAnswer = '777'
        const mockResponse = { data: { args: { answer: mockAnswer } } }
        spy1.mockResolvedValue(mockResponse)  // don't run the original!

        const realInput = '43'  // put this in the URL
        const realQuery = `https://httpbin.org/get?answer=${realInput}`

        // 'echo' takes the realInput, but returns the mockAnswer,
        // so the http call must not have taken place
        await expect(echo(realInput)).resolves.toEqual(mockAnswer);
        expect(spy1).toBeCalledWith(realQuery);
        expect(spy1).toBeCalledTimes(1)
        expect.assertions(3)
    })
```

Pattern: spy on one method of a class to replace it with a mock.

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

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.player.id);
      });
```

Learning Objectives for this Lesson

- You should now be prepared to:
  - Explain why you might need a test double in your testing
  - Use simple mocks and spies in your tests.