COMP 3550

8.3 — USING MOCKS AND FAKES IN COMPLEX TEST SUITES

Week 8: Advanced Testing

QUICK RECAP OF TEST DOUBLES

Type	Purpose	Example
Dummy	Placeholder, passed but never used	new DummyLogger()
Stub	Returns canned responses	when(repo.find()).thenReturn(null)
Fake	Working lightweight implementation	In-memory DB or FakeEmailService
Mock	Verifies interactions (e.g., was it called?)	verify(service).sendEmail()
Spy	Wraps a real object to observe behavior	spy(realService) with partial mocking

TEST DOUBLES IN INTEGRATION CONTEXTS

Guiding Principle:
Use real implementations for the components you're testing, and test doubles only for the components outside the scope.

Example Scenario:
You're testing RecipeService, which saves to a real test database and sends notifications via an external NotificationService.

```
// Integration Test Setup
RecipeRepository repo = new RealRecipeRepository(testDb);
NotificationService notifier = new FakeNotificationService(); // doesn’t send real emails

RecipeService service = new RecipeService(repo, notifier);
```

REPLACING EXTERNAL APIS WITH FAKES

Why Replace External APIs in Tests?

- External systems are:
  - Slow
  - Unreliable
  - Expensive (or have rate limits)

REPLACING EXTERNAL APIS WITH FAKES

Why Replace External APIs in Tests?

- External systems are:
  - Slow
  - Unreliable
  - Expensive (or have rate limits)
- Fakes let you:
  - Test without a real server
  - Inspect internal state
  - Simulate failures or edge cases

EXAMPLE: FAKEEMAILSERVICE

Instead of sending real emails, store them in a test-accessible

```
public class FakeEmailService implements EmailService {
    private List<Email> sentEmails = new ArrayList<>();

    @Override
    public void send(Email email) {
        sentEmails.add(email);
    }

    public List<Email> getSentEmails() {
        return sentEmails;
    }
}
```

Now your integration test can assert:

```
assertEquals(1, fakeEmailService.getSentEmails().size());
assertTrue(fakeEmailService.getSentEmails().get(0).getTo().contains("test@"));
```

USING MOCKITO (OR SIMILAR TOOLS)

Matchers: Flexible argument matching

```
when(repo.findById(anyInt())).thenReturn(recipe);
```

USING MOCKITO (OR SIMILAR TOOLS)

Matchers: Flexible argument matching

```
when(repo.findById(anyInt())).thenReturn(recipe);
```

Stubbing: Simulate method behavior

```
when(repo.save(any())).thenThrow(new RuntimeException("DB down"));
```

USING MOCKITO (OR SIMILAR TOOLS)

Matchers: Flexible argument matching

```
when(repo.findById(anyInt())).thenReturn(recipe);
```

Stubbing: Simulate method behavior

```
when(repo.save(any())).thenThrow(new RuntimeException("DB down"));
```

Verification: Check what was called

```
verify(repo).save(any(Recipe.class));
```

COMMON PITFALLS:

Don’t Do This...

- Over-mock everything
- Forget to verify important interactions
- Mock internal details of logic

Instead...

- Only mock collaborators, not the class under test
- Use verify() when interaction matters
- Mock only the public behavior you rely on

PROJECT PAUSE & REFLECT

Replace one flaky or slow dependency in your tests with a fake or mock.

Did the test speed or reliability improve?