Test Code Tangles

Test code is code.
What's considered bad in production code, is bad in test code.

Its intention should be immediately obvious.
Its functionality shouldn't be obscured.
Its results should be meaningful.
Its failure causes should be easy to find.

Hidden Arrange

Sometimes our tests rely on a non-obvious setup.

For example: The database implicitly gets set up with a test data set. The data is then used in the tests' asserts, but its source is not visible from the test itself.

This is problematic because it may hide failure causes.

Untangle with **Test Setup Helper Method**, **Test Data Builder**, **Test Data Manager**

Duplicate Arrange/ Assert Code

Creating and setting up test objects is duplicated in multiple tests.

Untangle with **Test Data Builder**, **Test Data Manager**

Magic Values

A magic value is basically some literal value appearing somewhere in the code without any hint why it was chosen or what it means.

Untangle with **Test Setup Helper Method**, **Explicit Constants**, **Test Data Builder**

```
var gilly = new Unicorn(
  "351d0356-6d5e-47d5-adbb-4909058fdf2f", // ??
  "Gilly", // I guess we use this all the time?
  ManeColor.RED, // Why not BLUE?
  154, // Is this important?
  12, // Why 12?
  today().minusYears(62).plusDays(1)
);

assertThat(gilly.age())
  .isEqualTo(61); // Not 62?!?
```

Long Arrange

When dealing with big data objects, we need to construct these objects to arrange our tests.

Quite often only few of the set properties are actually relevant for the test at hand. The other set properties are actually just set to satisfy the constructor of the class and often use Magic Values.

Untangle with **Test Setup Helper Method**, **Test Data Builder**

```
var gilly = new Unicorn(
  randomUUID(),
  "Gilly",
  ManeColor.RED,
  111,
  11,
  today().minusYears(62) // only relevant line!
);

assertThat(gilly.age()).isEqualTo(62);
```

Long Assert

Verifying big data objects can lead to a lot of simple assertions, which make the intention of the test hard to understand.

Untangle with **Verification Method**, **Test Data Builder** (asserting the result is equal to a built expected object)

```
var response = restTemplate
  .getForEntity(url, String.class);

var data = objectMapper
  .readTree(response.getBody());

assertThat(data.get("id").asText())
  .isEqualTo("4711");
assertThat(data.get("name").asText())
  .isEqualTo("Grace");
assertThat(data.get("maneColor").asText())
  .isEqualTo("RAINBOW");
assertThat(data.get("hornLength").asInt())
  .isEqualTo(42);
assertThat(data.get("hornDiameter").asInt())
  .isEqualTo(10);
assertThat(data.get("dateOfBirth").asText())
  .isEqualTo("1982-02-19");
```

Interdependent Test Cases

Test cases can be written in a way that they rely on other cases. Either by intention due to a very cumbersome setup, or accidentally when you assume the result of a series of test is actually the result of the arrange phase.

This is especially harmful as changing or removing one test case can make multiple other cases fail that dependent on it.

Untangle with **Test Data Manger**

Multiple Acts

When tests have multiple interactions with the unit under test, they usually have a lot of possible reasons to fail. This makes a failing test an ambiguous signal.

Untangle with **Split by Assumptions**, **Test Data Manager**

```
var postResponse = restTemplate
  .postForEntity(url, unicorn, String.class);

assertThat(response.getStatusCode())
  .isEqualTo(HttpStatusCode.valueOf(201));

var location = postResponse
  .getHeaders().get("Location")).get(0);
var getResponse = restTemplate
  .getForEntity(location, String.class);

assertThat(getResponse.getStatusCode())
  .isEqualTo(HttpStatusCode.valueOf(200));
```

Lying Names

Test case names are not executable code. Hence, their correctness is not checked automatically like the actual test code.

They often help though, to understand the general intention of a test case, and they are usually the data we get from test reports.

Often test case names are copy-pasted from other tests, and often we fail adjust them when the content of the test case changes.

Untangle with **Expressive & Consistent**, **Test Case Names**

```
@Test
void postInvalidUnicornYieldsA500Response() {
  var response = restTemplate
    .postForEntity(url, String.class);

  assertThat(response.getStatusCode())
    .isEqualTo(HttpStatusCode.valueOf(400));
  assertThat(response
      .getHeaders()
      .containsKey("Location"))
    .isFalse();
  assertThat(response.getBody())
    .contains("invalid unicorn");
}
```

Untangles

Arrange Act Assert

Tests should always have three parts (at most):

1. Arrange (or given) sets everything up for the test.
The code should be concise and focus on what makes the test case different from others.

2. Act (or when) contains the actual test.
This part should be one line/one interaction with the unit under test. Otherwise, the number of potential outcomes quickly become confusing.

3. Assert (or then) checks the effects of act.
Ideally this only checks one aspect of the result, so the test can only fail for one obvious reason.

Note that arrange is optional. Act and assert can be combined in one line of code.

Expressive & Consistent Test Case Names

Choose a naming scheme that makes test case names clear, expressive, concise, unambiguous, and easily understandable.

Useful components of a naming scheme are <unitUnderTest>, <stateUnderTest>, and <expectedBehavior>.

Any naming scheme is better than no naming scheme at all!

```
class <ClassUnderTest>Test {
  void <methodUnderTest>_<stateUnderTest>() {
    … // see code for <expectedBehavior>
  }
}
```

Split with Assumptions

If you find **Multiple Acts** in a test case, you might want to split it with Assumptions. Copy the test, remove the second act from the original, and replace the copy's assertion code with an assumption.

```
var postResponse = restTemplate
  .postForEntity(url, unicorn, String.class);
// ↓ this is asserted somewhere else
assumeThat(postResponse.getStatusCode())
  .isEqualTo(HttpStatusCode.valueOf(201));
var location = postResponse
  .getHeaders().get("Location")).get(0);

var getResponse = restTemplate
  .getForEntity(location, String.class);

assertThat(getResponse.getStatusCode())
  .isEqualTo(HttpStatusCode.valueOf(200));
```

Test Data Manager

A test data manager directly interacts with the database to inject wanted test data:

```
testDataManager.withUnicorn(unicorn);

var respose = restTemplate
  .getForEnitiy(url, String.class);

assertThat(response.getBody())
  .contains(unicorn.name());
```

It can also be used to clean up and hence allows to test no data scenarios.

Explicit Constants

Explicitly named test data constants can help to understand the test code a lot. Choose good names (e.g. prefix with "SOME" to express that the actual value doesn't matter).

```
var someUnicorn =
  new Unicorn(
    SOME_ID, // "some" value ⇒ doesn't matter
    SOME_UNICORN_NAME,
    SOME_MANE_COLOR,
    SOME_VALID_HORN_LENGTH,
    SOME_VALID_HORN_DIAMETER,
    today.minusYears(62)); // this matters
```

Random Data Generators

Use random data generators for test data.

This may be necessary to resolve dependencies between tests (e.g. to ensure uniqueness of IDs).

It can also technically underline that the concrete value should not matter for the test.

```
var someUnicorn =
  new Unicorn(
    randomUUID(), // needs to be unique
    someValidName(), // value doesn't matter
    someManeColor(),
    someValidHornLength(),
    someValidHornDiameter(),
    today.minusYears(62)); // this matters
```

Test Setup Helper Method

Create test setup helper methods to avoid duplication of test object setup in a lot of tests.

```
private Unicorn createUnicornBornAt(
  LocalDate dateOfBirth) {
  return new Unicorn(
    randomUUID(), SOME_NAME, SOME_MANE_COLOR,
    SOME_HORN_LENGTH, SOME_HORN_DIAMETER,
    dateOfBirth);
}

@Test
void age() {
  var gilly = createUnicornBornAt(
    today.minusYears(62));

  assertThat(gilly.age())
    .isEqualTo(62);
}
```

Test Data Builder

Create a builder class that allows to create whatever object is required for the test. That builder class should contain or use random data generators or constants to fill all the fields that are deemed irrelevant for the test at hand.

```
var unicorn = new UnicornTestDataBuilder()
    .dateOfBirth(today.minusYears(62))
    .build();

assertThat(unicorn.age())
  .isEqualTo(62);
```

Verification Method

Group long asserts that check one logical thing in verification methods. This reduces the amount of code in the test itself, and the method name makes the intention more obvious.

```
assertThat(jsonContainsUnicorn(
  response.body(), unicorn)).isTrue();
```

AssertJ also allows to use your own Conditions and Custom Assertions, which can make the verification code even more elegant.

```
UnicornAssert.assertThat(unicorn))
  .isValid()
  .isOlderThan(62);
```