Unit testing Cheat Sheet

by Alexandre Rosano via cheatography.com/193992/cs/40417/

About unit testing

Production code cleanliness cannot be greater than **test code cleanliness**.

Test code cleanliness mainly depends on its **readability**.

Unit tests can be viewed as **specifications** of the system.

FIRST principles

Fast : tests should be **fast**. If they're slow, they won't be run.

Independant : tests should **not depend on each other**, otherwise their result will be hard to analyze.

Repeatable : tests should be **repeatable in any environnement**. Their execution should not depend on the availability of a specific environnement.

Self-Validating : tests should **either pass or fail**. Evaluating their result mustn't be subjective.

Timely : tests should be written **just before** the **production code** is written. If they're written after, they will be difficult to write.

Test Driven Development

Test Driven Developement is a great way to write unit test in a **timely manner** and achieve a good **mutation score**.

1st rule : you may not write production code until you have written a failing unit test.

2nd rule : you may not write more of a unit test than is sufficient to fail, and not compiling is failing.

3rd rule : you may not write more production code than is sufficient to pass the currently failing test.

Obvious implementation : just **write it** and see if the tests pass.

Triangulate : **drive** implementation by using a set of **several examples**.

General rules

One test equals one behavior.

Write tests at the **uppermost level** of code. Changing implementation details should not break a test, only a **new behavioral need** (or a bug!) should.

Use **contract testing** to write **accurate assertions** and **reuse them** between several implementations tests.

Name tests in a **fluent way** or use the **pattern "Given - When - Then"**

The **body** of test methods should clearly show the **pattern "Given - When - Then"**.

"Hide wires" : **boilerplate** or **technical** code irrelevent to understanding a test should be **hidden**.

Hide irrelevant functional data. It is **noise** that lowers the test understandability.

Abstract **magic values** by giving them **meaningful names**. It makes tests more understandable.

Do **not** duplicate **production logic**. If the logic has a bug, it will be duplicated, making the test useless.

Do **not** use **conditional logic**. It lowers the readability of the test. **Split** the test instead.

Test doubles

Dummy : **fixed values not used** by tests.

Stub : **fixed values used** by tests.

Fake : **dynamic values used** by tests.

Spy : **fixed values used** by test ; provides **data** to **inspect its behavior**.

Mock : **fixed values used** by test ; provides **methods** to **inspect its behavior**.

Only use tests doubles **when necessary**. Do not over use them as it will **lower the readability** and **maintainability** of the tests.

Use **"manually created" fakes most of the time**. They are **reusable**, **decouple** test code from production code and **discourage** the use of assertions **based on behavior**.

Assertions

Prefer **assertions based on states** over assertions based on behavior. The later are less maintainable as they **couple** test code to production code.

Use an **expressive** assertion **framework**. Assertions should be understable quickly.

Only make assertions **regarding current case**. Tests should only fail regarding their case expectations.

Do **not aggregate** assertions, as the cause of a failing test would be hard to spot.

Assert **exceptions** as they also are **specifications** of the system.

Testing approaches

Use **property-based testing** to **spot values** that don't produce expected output.

Property-based testing and traditional **example-based testing** are **complementary**.

Use **"golden master testing"** to take a snapshot of **legacy code** outputs before **refactoring** it.

Parameterized tests can help dealing with **edge values**, by using a set of values covering the edges.

When **fixing a bug**, **start by writing a test** that shows it exists, then fix the bug.

Metrics

Do not rely on code coverage alone : it only shows if production code is executed by the tests.

Use **mutation testing** over **code coverage** to evaluate unit tests effectiveness.

Mutation testing also helps having a **minimal production code**, less production code meaning less possible mutations.

Credits

Test doubles definitions inspired from this post by Robert C. Martin

By **Alexandre Rosano**
cheatography.com/alexandre-rosano/

Published 6th October, 2023.
Last updated 5th August, 2024.