Google

Testing on the Toilet

Only Verify State-Changing Method Calls

Which lines can be safely removed from this test?

```
@Test public void addPermissionToDatabase() {
  new UserAuthorizer(mockUserService, mockPermissionDb).grantPermission(USER, READ_ACCESS);

  // The test will fail if any of these methods is not called.
  verify(mockUserService).isUserActive(USER);
  verify(mockPermissionDb).getPermissions(USER);
  verify(mockPermissionDb).isValidPermission(READ_ACCESS);
  verify(mockPermissionDb).addPermission(USER, READ_ACCESS);
}
```

The answer is that **the calls to verify the non-state-changing methods can be removed**.

Method calls on another object fall into one of two categories:

- ***State-changing***: methods that have side effects and change the world outside the code under test, e.g., `sendEmail()`, `saveRecord()`, `logAccess()`.
- ***Non-state-changing***: methods that return information about the world outside the code under test and don't modify anything, e.g., `getUser()`, `findResults()`, `readFile()`.

You should usually **avoid verifying that non-state-changing methods are called**:

- *It is often redundant*: **a method call that doesn't change the state of the world is meaningless on its own**. The code under test will use the return value of the method call to do other work that you can assert.
- *It makes tests brittle*: tests need to be updated whenever method calls change. For example, if a test is expecting `mockUserService.isUserActive(USER)` to be called, it would fail if the code under test is modified to call `user.isActive()` instead.
- *It makes tests less readable*: the additional assertions in the test make it more difficult to determine which method calls actually affect the state of the world.
- *It gives a false sense of security*: just because the code under test called a method does not mean the code under test did the right thing with the method's return value.

Instead of verifying that they are called, **use non-state-changing methods to simulate different conditions in tests**, e.g., `when(mockUserService.isUserActive(USER)).thenReturn(false)`. Then write assertions for the return value of the code under test, or verify state-changing method calls.

Verifying non-state-changing method calls may be useful if there is no other output you can assert. For example, if your code should be caching an RPC result, you can verify that the method that makes the RPC is called only once.

With the unnecessary verifications removed, the test looks like this:

```
@Test public void addPermissionToDatabase() {
  new UserAuthorizer(mockUserService, mockPermissionDb).grantPermission(USER, READ_ACCESS);

  // Verify only the state-changing method.
  verify(mockPermissionDb).addPermission(USER, READ_ACCESS);
}
```

That's much simpler! But remember that instead of using a mock to verify that a method was called, **it would be even better to use a real or fake object** to actually execute the method and check that it works properly. For example, the above test could use a fake database to check that the permission exists in the database rather than just verifying that `addPermission()` was called.

More information, discussion, and archives: `testing.googleblog.com`

You can learn more about this topic in the book Growing Object-Oriented Software, Guided by Tests. Note that the book uses the terms “command” and “query” instead of “state-changing” and “non-state-changing”.