

Regression Testing

Regression testing refers to any software testing that uncovers errors by retesting the modified program (Wikipedia). This form of testing often refers to large combined sets of test cases to detect regressions:

- of a bug fix that a developer has proposed.
- of related and unrelated other features that have been added.

Regression testing usually refers to system level (integration level) testing that runs the entire process.

Attributes of Regression Tests

Regression tests usually have the following attributes:

- **Automated:** no real reason to have manual regression tests.
- **Appropriately Sized:** too small and bugs will be missed. Too large and they will take a long time to run. Optimally, we want to run tests continuously.
- **Up-to-date:** ensure that tests are valid for the version of program being tested.

Automating Regression Tests

Regression tests often have a low yield in terms of finding bugs (and are boring to run). Automation is key.

Input

If the input is from a file, regression tests are easy to run (but should still be automatically triggered on a regular interval). There may still be a problem with validating output. We can also create special mocks that can take input from a file or other sources (e.g. scripting engines).

For UIs, the standard approach is to capture and replay events. This approach can be fragile! For example, tests may fail based on window placement or whitespace. For web applications, there is capture and replay for HTTP using Selenium. Mozilla has a project named Marionette¹ that is used to test Firefox and Thunderbird; it is like Selenium but also works on Chrome elements.

¹<https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette>

Output

Verifying output can be hard!! Problems can arise from issues such as resolution, whitespace, window placement etc.

Mozilla Case Study²

The case study presents an approach to testing Gecko based applications. Gecko is the layout engine for Mozilla applications like Firefox and Thunderbird.

In the past, a frame tree with coordinates for all UI elements was created and manual testers performed tests. Not optimal! The new approach was to capture screenshots after test cases and compare them with the expected screenshot. There were a few problems with the approach due to nondeterminism:

- Animated images: no way to ensure tests would function with animated images.
- Font Hinting: the same character would appear slightly differently after each run of the application.
- Other bugs: resolution problems, minor changes in layout etc.

The problem was “really hellish” and the partial solution was to enable logging in the application. The logs would essentially be compared with expected logs. This became very ugly given 1300 or so test cases; distributing across different computers helped.

Some Notes

- Sometimes it is a good idea to have 1 test case per bug fix. This can get unwieldy over time if redundant test cases are not removed.
- Test suites can be evaluated using coverage criteria (e.g. how many use cases were covered) or manually in order to get rid of redundant tests.
- There is some interest in test case prioritization but little practical application since it is unclear how to implement it.
- Expect low yields in terms of finding bugs so automation is key!
- Ensure regression tests are up-to-date. This can be a pain but is necessary since when software changes, test suits break or become incomplete. When a new software version comes out, try the old suite.
 - If there are no failures, new tests may need to be added to test new functionality.
 - If there are some failures, determine if software or test is broken. Tests can depend on inessential features of the output (e.g. order or text etc.).
- Try to get rid of irrelevant tests over time. For example, if you have a bunch of tests that are related to the same bug, keep only 1 test for the bug.

²http://robert.ocallahan.org/2005/03/visual-regression-tests_04.html

Industrial Best Practices

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. In order to commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. This ensures developers try to submit good code since if you break something, everyone knows about it :)
- **One-button Deploy:** If all tests have passed, one should be able to deploy to production with one command.
- **Back Button:** Systems should be designed so that it's possible to roll back changes.