Software Testing, Quality Assurance and Maintenance Winter 2024

Lecture 2 — January 8, 2024

Patrick Lam *version 1*

Exploratory Testing

Exploratory testing is usually (but not always) carried out by dedicated testers. In that sense, it's somewhat different from the other testing activities in this course, which are more developer-focussed—our usual goal is learning, as developers, how to deploy better automated test suites for our software. Hallway usability testing, though, is an application of exploratory testing. Furthermore, the dedicated QA function is important, and we should learn about how it works.

Resources. James Bach has an introduction to exploratory testing:

- `https://www.satisfice.com/exploratory-testing`

There is an exhaustive set of notes on exploratory testing by Cem Kaner:

- `https://www.kaner.com/pdfs/QAIExploring.pdf`

> "Exploratory testing is simultaneous learning, test design, and test execution."

Contrast this to scripted testing: test design happens ahead of time and then test execution happens (repeatedly) throughout the product's development cycle. When we think of dedicated QA teams, we think they are manually executing scripted tests. In 2025, that is not an effective use of staff.

There is a continuum between scripted testing and exploratory testing. Good exploratory testing may use prepared scripts for certain tasks.

Scenarios where Exploratory Testing Excels. (from Bach's article)

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process. Exploratory testing should not be randomly bumbling around (we can call that "ad hoc testing")—the random approach finds bugs but isn't the most efficient at giving you an idea of how well the software works.

- Start with a charter for your testing activity, e.g. "Explore and analyze the product elements of the software." These charters should be somewhat ambiguous.

- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Exploratory testing shouldn't produce an exhaustive set of notes. Good testers will be able to reproduce the bugs that they encounter during their testing from brief notes. Taking full notes takes too long.

The output from exploratory testing is at least a set of bug reports. It may also include test notes, which include overall impressions and a summary of the test strategy/thought process. Artifacts such as test data or test materials are also both inputs and outputs from exploratory testing.

Primary vs contributing tasks. One way to classify tasks that software can do (or, in other words, its features) is *primary* vs *contributing.* A *primary* task is core functionality of the system; it's something that you would say "You Had One Job!" about. As examples, text editors must be able to load text files, add text, and save the text files. On the other hand, *contributing* tasks are secondary. A macro system for a text editor would be a contributing task. Being able to read email in your editor is definitely a contributing task. Sometimes it's not black-and-white. Spell-check can go either way.

In-class exercise: Exploratory testing of WaterlooWorks.

We will try out exploratory testing with WaterlooWorks. I believe that all non-exchange students here should have access to the system, although there may be no jobs visible right now.

The charter will be "Explore the overall functionality of WaterlooWorks". Summarize in one or two sentences what the purpose of WaterlooWorks is. Identify the tasks that WaterlooWorks should be able to do and classify them as primary or contributing. Identify areas of potential instability. Test each function and record results (bugs).

Of course, don't do things that have actual effects. Usually, testers would have access to a development server and could test those areas more aggressively. But we are working with production systems here.

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 comprehensive sets of test cases to detect regressions:

- of bug fixes 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 Marionette[1] that is used to test Firefox and Thunderbird; it is like Selenium but also works on Chrome elements.

Output

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

Mozilla Case Study[2]

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.

[1]`https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette`

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

Some Notes on Test Suites

- 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.
- One could have tiered level of tests, which are progressively more detailed; run the high-level suite often and more detailed tests as needed.
- 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.

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.