18F

Automated Testing Playbook

APIs and Legacy Systems

Technical impediments to the automated testing of existing systems and how to overcome them.

WORK IN PROGRESS: This material is currently in draft form and under active development. See the GitHub issues page to examine progress.

Fix the Build!

"*...if people feel they don't have the power to change a bad situation, **then they do not think about it**.*" — *Saul Alinsky*, Rules for Radicals, *p. 105*

A slow, difficult and/or incorrect build leads to unhappy programmers and bad testing experiences. They are powerless to change their circumstances.

A fast, correct build leads to happy programmers and good testing experiences. **They have the power to move fast and try new things**.

Often the first step towards improving automated testing discipline is to fix the build system. That isn't to say that educational outreach isn't also necessary, but it often necessary to fix the build system before people will attempt to write automated tests. After all, trying to

deliver a product is difficult enough; why add the pain of learning a new skill on top of that when the build system adds so much friction to the learning process?

Dangle the Carrot of Revolution

Both the speed of the build system and correctly-declared dependencies are critical to successful automated testing practices. By the same token, correctly-declared and properly-maintained dependencies are often required to make use of better, faster build tools in the first place.

By dangling the promise of a faster build in front of reluctant developers, they will willingly solve many of the same problems that make tests slow to build and execute in the first place. By removing the "I don't have time to test" excuse imposed by a slow build system, you are free to educate your developers to then eliminate the "My code is too hard to test" excuse borne of inexperience.

References:

- Large Scale Culture Change: Google and the US Government

Work with the Existing Code

It's called "*soft*ware" for a reason. Despite the overall complexity of a large program, nearly every program can succumb to the classic divide-and-conquer strategy: smaller pieces can be extracted to test in isolation.

As mentioned earlier, "golden files" can be used to ensure feature parity as a temporary measure when refactoring to get parts of a program under test. Michael Feathers's book *Working Effectively with Legacy Code* contains this and a host of other strategies for getting an unwieldy code base under test.

Rewriting to APIs

This refers not to the popular modern practice of exposing an Application Programming Interface (API) as an accessible HTTP (or, preferably, HTTPS) endpoint that provides structured data in JSON format, but to identifying a "seam" at which an existing subsystem integrates with the larger system. Inject an interface to make this subsystem easily swappable, if needed. Use a "golden file" or other means to check for feature parity, if necessary, then test the extracted component in isolation via its internal API.

xUnit is a Concept, Not a Framework

Kent Beck's original xUnit concept was implemented in Smalltalk, an Object-Oriented language, and has been replicated in virtually every OO language since. However, should you find yourself working in a non-OO code base, such as one written in C, the same xUnit concepts can be replicated *without* an explicit framework following the Pseudo-xUnit Pattern:

- *Test Fixtures*: Defined using bare structs passed to `setup()` and `teardown()` functions. Fixtures can "inherit" functionality by composing new structs, and `setup()`/`teardown()` functions in terms of the "parent" version.
- *Execution function*: An `execute()` function (or set of functions) that contains the relevant assertions and error message formatting.
- *Test Cases*: Functions that follow the pattern: `setup(); execute(); teardown()`.
- *Test Suites*: Sets of test case functions that all start with a common prefix.
- *Test Runner*: `main()`

For an example, see OpenSSL's ssl/heartbeat_test.c.