When Should a Test Be Automated?

Brian Marick
Testing Foundations
marick@testing.com

I want to automate as many tests as I can. I'm not comfortable running a test only once. What if a programmer then changes the code and introduces a bug? What if I don't catch that bug because I didn't rerun the test after the change? Wouldn't I feel horrible?

Well, yes, but I'm not paid to feel comfortable rather than horrible. I'm paid to be cost-effective. It took me a long time, but I finally realized that I was over-automating, that only *some* of the tests I created should be automated. Some of the tests I was automating not only did not find bugs when they were rerun, they had no significant prospect of doing so. Automating them was not a rational decision.

The question, then, is how to make a rational decision. When I take a job as a contract tester, I typically design a series of tests for some product feature. For each of them, I need to decide whether that particular test should be automated. This paper describes how I think about the tradeoffs.

Scenarios

In order for my argument to be clear, I must avoid trying to describe all possible testing scenarios at once. You as a reader are better served if I pick one realistic and useful scenario, describe it well, and then leave you to apply the argument to your specific situation. Here's my scenario:

1. You have a fixed level of automation support. That is, automation tools are available. You know how to use them, though you may not be an expert. Support libraries have been written. I assume you'll work with what you've got, not decide to acquire new tools, add more than simple features to a tool support library, or learn more about test automation. The question is: given what you have now, is automating this test justified? The decision about what to provide you was made earlier, and you live with it.

   In other scenarios, you might argue for increased automation support later in the project. This paper does not directly address when that's a good argument, but it provides context by detailing what it means to reduce the cost or increase the value of automation.

2. There are only two possibilities: a completely automated test that can run entirely unattended, and a "one-shot" manual test that is run once and then thrown away. These are extremes on a continuum. You might have tests that automate only

cumbersome setup, but leave the rest to be done manually. Or you might have a manual test that's carefully enough documented that it can readily be run again. Once you understand the factors that push a test to one extreme or the other, you'll know better where the optimal point on the continuum lies for a particular test.

3. Both automation and manual testing are plausible. That's not always the case. For example, load testing often requires the creation of heavy user workloads. Even if it were possible to arrange for 300 testers to use the product simultaneously, it's surely not cost-effective. Load tests need to be automated.

4. Testing is done through an external interface ("black box testing"). The same analysis applies to testing at the code level - and a brief example is given toward the end of the paper - but I will not describe all the details.

5. There is no mandate to automate. Management accepts the notion that some of your tests will be automated and some will be manual.

6. You first design the test and then decide whether it should be automated. In reality, it's common for the needs of automation to influence the design. Sadly, that sometimes means tests are weakened to make them automatable. But - if you understand where the true value of automation lies - it can also mean harmless adjustments or even improvements.

7. You have a certain amount of time to finish your testing. You should do the best testing possible in that time. The argument also applies in the less common situation of deciding on the tests first, then on how much time is required.

Overview

My decision process uses these questions.

1. Automating this test and running it once will cost more than simply running it manually once. How much more?

2. An automated test has a finite lifetime, during which it must recoup that additional cost. Is this test likely to die sooner or later? What events are likely to end it?

3. During its lifetime, how likely is this test to find additional bugs (beyond whatever bugs it found the first time it ran)? How does this uncertain benefit balance against the cost of automation?

If those questions don't suffice for a decision, other minor considerations might tip the balance.

The third question is the essential one, and the one I'll explore in most detail. Unfortunately, a good answer to the question requires a greater understanding of the product's structure than testers usually possess. In addition to describing what you can do with that understanding, I'll describe how to get approximately the same results without it.

What Do You Lose With Automation?

Creating an automated test is usually more time-consuming (expensive) than running it once manually.[1] The cost differential varies, depending on the product and the automation style.

- If the product is being tested through a GUI (graphical user interface), and your automation style is to write scripts (essentially simple programs) that drive the GUI, an automated test may be several times as expensive as a manual test.
- If you use a GUI capture/replay tool that tracks your interactions with the product and builds a script from them, automation is relatively cheaper. It is not as cheap as manual testing, though, when you consider the cost of recapturing a test from the beginning after you make a mistake, the time spent organizing and documenting all the files that make up the test suite, the aggravation of finding and working around bugs in the tool, and so forth. Those small "in the noise" costs can add up surprisingly quickly.
- If you're testing a compiler, automation might be only a little more expensive than manual testing, because most of the effort will go into writing test programs for the compiler to compile. Those programs have to be written whether or not they're saved for reuse.

Suppose your environment is very congenial to automation, and an automated test is only 10% more expensive than a manual test. (I would say this is rare.) That still means that, once you've automated ten tests, there's one manual test - one unique execution of the product - that is never exercised until a customer tries it. If automation is more expensive, those ten automated tests might prevent ten or twenty or even more manual tests from ever being run. What bugs might those tests have found?

So the first test automation question is this:

If I automate this test, what manual tests will I lose? How many bugs might I lose with them? What will be their severity?

The answers will vary widely, depending on your project. Suppose you're a tester on a telecom system, one where quality is very important and the testing budget is adequate. Your answer might be "If I automate this test, I'll probably lose three manual tests. But I've done a pretty complete job of test design, and I really think those additional tests would only be trivial variations of existing tests. Strictly speaking, they'd be different executions, but I really doubt they'd find serious new bugs." For you, the cost of automation is low.

[1] There are exceptions. For example, perhaps tests can be written in a tabular format. A tool can then process the table and drive the product. Filling in the table might be faster than testing the product manually. See [Pettichord96] and [Kaner97] for more on this style. If manual testing is really more expensive, most of the analysis in this paper does not apply. But beware: people tend to underestimate the cost of automation. For example, filling in a table of inputs might be easy, but automated results verification could still be expensive. Thanks to Dave Gelperin for pressing me on this point.

Or you might be a testing version 1.0 of a shrinkwrap product whose product direction and code base has changed wildly in the last few months. Your answer might be "Ha! I don't even have time to try all the *obvious* tests once. In the time I would spend automating this test, I *guarantee* I could find at least one completely new bug." For you, the cost of automation is high.

My measure of cost - bugs probably foregone - may seem somewhat odd. People usually measure the cost of automation as the time spent doing it. I use this measure because the point of automating a test is to find more bugs by rerunning it. Bugs are the value of automation, so the cost should be measured the same way.[2]

A note on estimation

I'm asking you for your best estimate of the number of bugs you'll miss, on average, by automating a single test. The answer will not be "0.25". It will not even be "0.25 ± 0.024". The answer is more like "a good chance at least one will be missed" or "probably none".

Later, you'll be asked to estimate the lifetime of the test. Those answers will be more like "probably not past this release" or "a long time" than "34.6 weeks".

Then you'll be asked to estimate the number of bugs the automated test will find in that lifetime. The answer will again be indefinite.

And finally, you'll be asked to compare the fuzzy estimate for the manual test to the fuzzy estimate for the automated test and make a decision.

Is this useful?

Yes, when you consider the alternative, which is to make the same decision - perhaps implicitly - with even less information. My experience is that thinking quickly about these questions seems to lead to better testing, despite the inexactness of the answers. I favor imprecise but useful methods over precise but misleading ones.

How Long Do Automated Tests Survive?

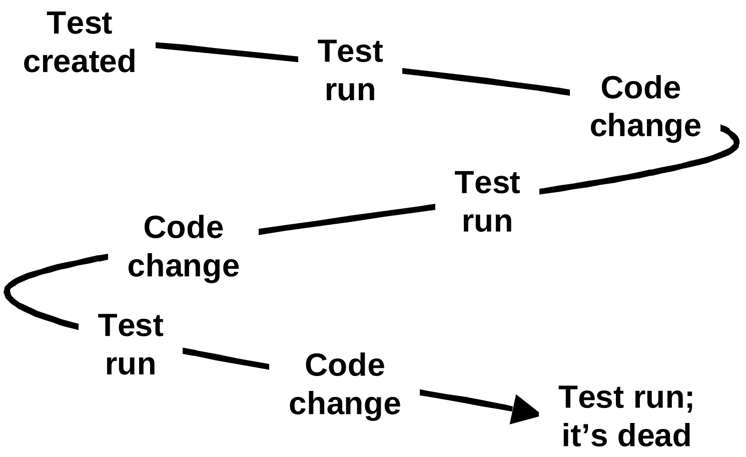
Automated tests produce their value after the code changes. Except for rare types of tests, rerunning a test before any code changes is a waste of time: it will find exactly the same bugs as before. (The exceptions, such as timing and stress tests, can be analyzed in the roughly same way. I omit them for simplicity.)

But a test will not last forever. At some point, the product will change in a way that breaks the test. The test will have to either be repaired or discarded. To a reasonable approximation, repairing a test costs as much as throwing it away and writing it from

[2] I first learned to think about the cost of automation in this way during conversations with Cem Kaner. Noel Nyman points out that it's a special case of John Daly's Rule, which has you always ask this question of any activity: "What bugs aren't I finding while I'm doing that?"

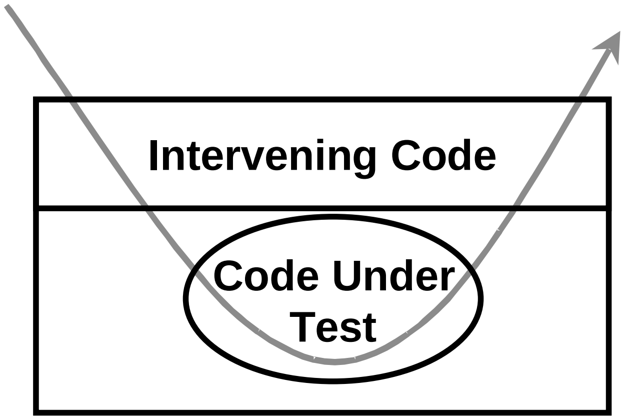
scratch[3]. Whichever you do when the test breaks, if it hasn’t repaid the automation effort by that point, you would have been better off leaving it as a manual test.

In short, the test’s useful lifespan looks like this:



When deciding whether to automate a test, you must estimate how many code changes it will survive. If the answer is "not many", the test had better be especially good at finding bugs.

To estimate a test’s life, you need some background knowledge. You need to understand something of the way code structure affects tests. Here’s a greatly simplified diagram to start with.



Suppose your task is to write a set of tests that check whether the product correctly validates phone numbers that the user types in. These tests check whether phone numbers have the right number of digits, don’t use any disallowed digits, and so on. If you

[3] If you’re using a capture/replay tool, you re-record the test. That probably costs more than recording it in the first place, when you factor in the time spent figuring out what the test was supposed to do. If you use test scripts, you need to understand the current script, modify it, try it out, and fix whatever problems you uncover. You may discover the new script can’t readily do everything that the old one could, so it’s better to break it into two scripts. And so on. If your testing effort is well-established, repairing a scripted test may be cheaper than writing a new one. That doesn’t affect the message of the paper; it only reduces one of the costs of automation. But make sure you’ve measured the true cost of repair: people seem to guess low.

understood the product code (and I understand that you rarely do), you could take a program listing and use a highlighter to mark the phone number validation code. I'm going to call that **the code under test**. It is the code whose behavior you thought about to complete your testing task.

In most cases, you don't exercise the code under test directly. For example, you don't give phone numbers directly to the validation code. Instead, you type them into the user interface, which is itself code that collects key presses, converts them into internal program data, and delivers that data to the validation routines. You also don't examine the results of the validation routines directly. Instead, the routines pass their results to other code, which eventually produces results visible at the user interface (by, for example, producing an error popup). I will call the code that sits between the code under test and the test itself the **intervening code**.

Changes to the intervening code

The intervening code is a major cause of test death. That's especially true when it's a graphical user interface as opposed to, say, a textual interface or the interface to some standard hardware device. For example, suppose the user interface once required you to type in the phone number. But it's now changed to provide a visual representation of a phone keypad. You now click on the numbers with a mouse, simulating the use of a real phone. (A really stupid idea, but weirder things have happened.) Both interfaces deliver exactly the same data to the code under test, but the UI change is likely to break an automated test, which no longer has any place to "type" the phone number.

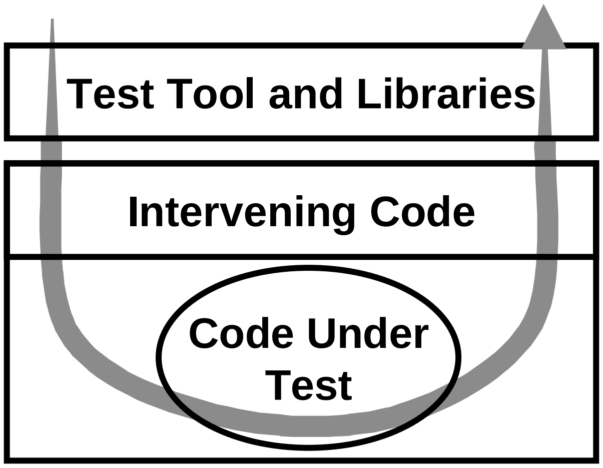
As another example, the way the interface tells the user of an input error might change. Instead of a popup dialog box, it might cause the main program window to flash red and have the sound card play that annoying "your call cannot be completed as dialed" tone. The test, which looks for a popup dialog, should consider the new correct action a bug. It is effectively dead.

"Off the shelf" test automation tools can do a limited job of preventing test death. For example, most GUI test automation tools can ignore changes to the size, position, or color of a text box. To handle larger changes, such as those in the previous two paragraphs, they must be customized. That is done by having someone in your project create product-specific **test libraries**. They allow you, the tester, to write your tests in terms of the feature you're testing, ignoring - as much as possible - the details of the user interface. For example, your automated test might contain this line:

```
try 217-555-1212
```

`try` is a library routine with the job of translating a phone number into terms the user interface understands. If the user interface accepts typed characters, `try` types the phone number at it. If it requires numbers to be selected from a keypad drawn on the screen, `try` does that.

In effect, the test libraries filter out irrelevant information. They allow your test to specify only and exactly the data that matters. On input, they add additional information required by the intervening code. On output, they condense all the information from the intervening code down to the important nugget of information actually produced by the code under test. This filtering can be pictured like this:



Many user interface changes will require no changes to tests, only to the test library. Since there is (presumably) a lot more test code than library code, the cost of change is dramatically lowered.

However, even the best compensatory code cannot insulate tests from all changes. It's just too hard to anticipate everything. So there is some likelihood that, at some point in the future, your test will break. You must ask this question:

How well is this test protected from changes to the intervening code?

You need to assess how likely are intervening code changes that will affect your test. If they're extremely unlikely - if, for example, the user interface really truly is fixed for all time - your test will have a long time to pay back your effort in automating it. (I would not believe the GUI is frozen until the product manager is ready to give me $100 for every future change to it.)

If changes are likely, you must then ask how confident you are that your test libraries will protect you from them. If the test library doesn't protect you, perhaps it can be easily modified to cope with the change. If a half-hour change rescues 300 tests from death, that's time well spent. Beware, though: many have grossly underestimated the difficulty of maintaining the test library, especially after it's been patched to handle change after change after change. You wouldn't be the first to give up, throw out all the tests and the library, and start over.

If you have no test libraries - if you are using a GUI test automation tool in capture/replay mode - you should expect little protection. The next major revision of the user interface will kill many of your tests. They will not have much time to repay their cost. You've traded low creation cost for a short lifetime.

Changes to the code under test

The intervening code isn’t the only code that can change. The code under test can also change. In particular, it can change to do something entirely different.

For example, suppose that some years ago someone wrote phone number validation tests. To test an invalid phone number, she used 1-888-343-3533. At that time, there was no such thing as an "888" number. Now there is. So the test that used to pass because the product correctly rejected the number now fails because the product correctly accepts a number that the test thinks it should reject. This may or may not be simple to fix. It’s simple if you realize what the problem is: just change "888" to "889".   But you might have difficulty deciphering the test well enough to realize it’s checking phone number validation. (Automated tests are notoriously poorly documented.) Or you might not realize that "888" is now a valid number, so you think the test has legitimately found a bug. The test doesn’t get fixed until after you’ve annoyed some programmer with a spurious bug report.

So, when deciding whether to automate a test, you must also ask:

How stable is the behavior of the code under test?

Note the emphasis on "behavior". Changes to the code are fine, so long as they leave the externally visible behavior the same.

Different types of products, and different types of code under test, have different stabilities. Phone numbers are actually fairly stable. Code that manipulates a bank account is fairly stable in the sense that tests that check whether adding $100 to an account with $30 yields an account with $130 are likely to continue to work (except when changes to the intervening code breaks them). Graphical user interfaces are notoriously unstable.

Additions to behavior are often harmless. For example, you might have a test that checks that withdrawing $100 from an account with $30 produces an error and no change in the account balance. But, since that test was written, a new feature has been added: customers with certain accounts have "automatic overdraft protection", which allows them to withdraw more money than they have in the account (in effect, taking out a loan). This change will not break the existing test, so long as the default test account has the old behavior. (Of course, new tests must be run against the new behavior.)

Where do we stand?

We now know the hurdle an automated test must leap: its value must exceed the value of all the manual tests it prevents. We’ve estimated the lifespan of the test, the time during which it will have opportunities to produce value. Now we must ask how likely it is that the test actually will produce value. What bugs might we expect from it?

Will the Test Have Continued Value?

The argument here is complicated, so I will outline it first.

1. The code under test has structure. As a useful approximation, we can divide it into feature code and support code.
2. Tests are typically written to exercise the feature code. The support code is invisible to the tester.
3. But changes to the feature code usually change behavior. Hence, they are more likely to end a test's life than to cause it to report a bug.
4. Most of a test's value thus comes from its ability to find bugs in changes to the support code.
5. But we don't know anything about the support code! How can we know if there will be future bugs for the test to find? How can we guess if the test will do a good job at finding those bugs?
    - There will be bugs if there's change. If there's been change in the past, there will likely be more change in the future.
    - We may have a hard time knowing whether a test will do a good job, but there's one characteristic that ensures it will do a <u>bad</u> one. Don't automate such tests.
6. The code under test interacts with the rest of the product, which can be considered still more support code. Changes to this support code also cause bugs that we hope automated tests will find.
    - We can again identify a characteristic of low-value tests. High-value tests are unlikely to be feature-driven tests; rather, they will be task-driven.

More terminology

To understand what makes an automated test have value, you need to look more closely at the structure of the code under test.

Suppose the code under test handles withdrawals from a bank account. Further suppose that each test's **purpose** has been concisely summarized. Such summaries might read like this:

- "Check that a cash withdrawal of more than $9,999 triggers a Large Withdrawal audit trail record."
- "Check that you can completely empty the account."
- "Check that overdrafts of less than $100 are automatically allowed and lead to a Quick Loan tag being added to the account."
- "Check that you can withdraw money from the same account at most four times a day."

Now suppose you looked at a listing of the code under test and used a highlighter to mark the code that each test was intended to exercise. For example, the first test purpose would highlight the following code:

```
if (amount > 9999.00) audit(transaction, LARGE_WITHDRAWAL);
```

When you finished, not all of the code under test would be highlighted. What's the difference between the highlighted and unhighlighted code? To see, let's look more closely at the last purpose ("check that you can withdraw money from the same account at most four times a day"). I can see two obvious tests.

1. The first test makes four withdrawals, each of which should be successful. It then attempts another withdrawal, which should fail.
2. The next makes four withdrawals, each of which should be successful. It waits until just after midnight, then makes four more withdrawals. Each of those should again succeed. (It might also make a fifth withdrawal, just to see that it still fails.)

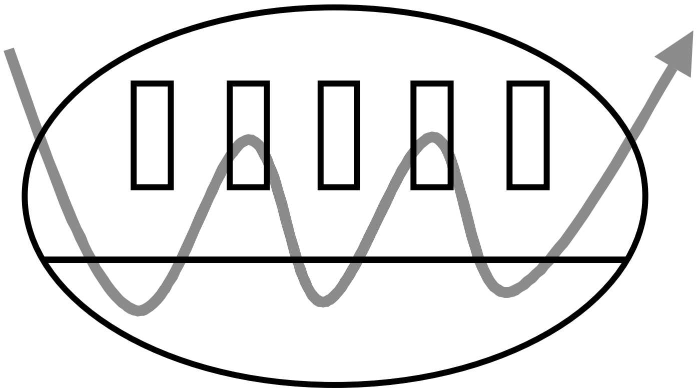
How does the code under test know that there have been four withdrawals from an account today? Perhaps there's a chunk of code that maintains a list of all withdrawals that day. When a new one is attempted, it searches for matching account numbers in the list and totals up the number found.

How well will the two tests exercise that searching code? Do they, for example, check that the searching code can handle very long lists of, say, 10,000 people each making one or a couple of withdrawals per day? No. Checking that wouldn't occur to the tester, because the searching code is completely hidden.

As a useful approximation, I divide the code under test into two parts:

1. The **feature code** directly implements the features that the code under test provides. It is intentionally exercised by tests. It performs those operations a user selects (via the intervening code of the user interface).
2. The **support code** supports the feature code. Its existence is not obvious from an external description of the code under test; that is, from a list of features that code provides. It is exercised by tests written for other reasons, but no test specifically targets it.

Here's a picture:



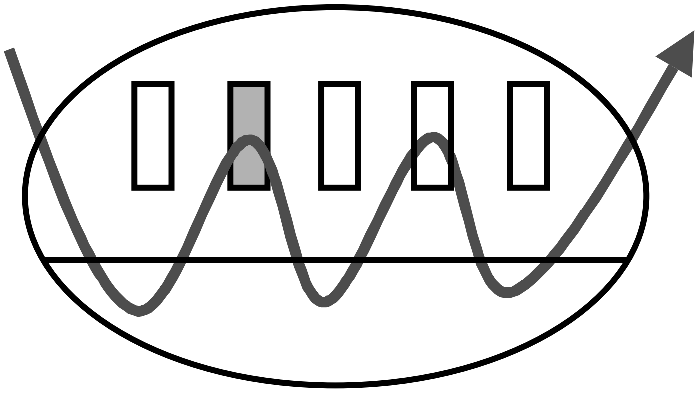
Here, the support code lies beneath the horizontal line. The feature code lies above it. There are five distinct features. The particular test shown happens to exercise two of them.

(Note that I am describing strictly the code under test. As we'll see later, the rest of the system has some additional relevant structure.)

The effects of change within the code under test

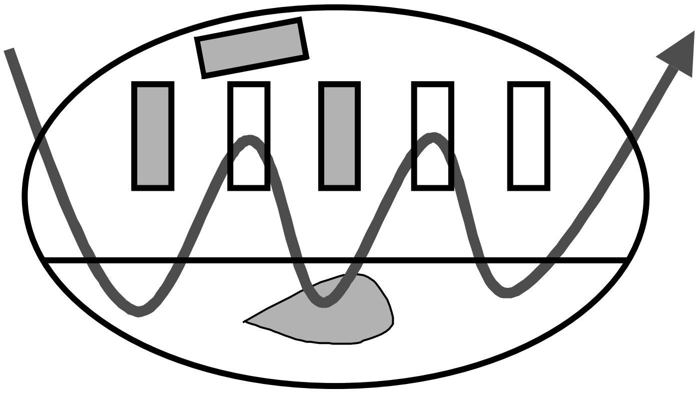
Given this structure, what can we say about the effects of change? What types of change cause tests to have value?

Suppose some feature code is changed, as shown by the gray box in this picture:



It is very likely that this change will break tests that exercise the feature code. Most changes to feature code are intended to change behavior in some way. Thus, if you're hoping for your automated test to discover bugs in changed feature code, you stand a good chance of having the test die just as it has its first chance to deliver value. This is not economical if the cost of automating the test was high.

What changes to the code under test *should* leave a given test's behavior alone? I believe the most common answer is changes to support code made in support of changes to other feature code. Consider this picture:



Two features have changed. One new feature has been added. To make those changes and additions work, some support code - exercised only accidentally by any tests - has been changed. That changed support code is also used by unchanged features, and it is exercised by tests of those unchanged features. If - as is certainly plausible - the support code was changed in a way that lets it work for the new code, but breaks it for the old code, such tests have a chance of catching it. But only if they're rerun, which is more likely if they're automated.

This, then, is what I see as the central insight, the central paradox, of automated testing:

An automated test's value is mostly unrelated to the specific purpose for which it was written. It's the accidental things that count: the untargeted bugs that it finds.

The reason you wrote the test is not the reason it finds bugs. You wrote the test to check whether withdrawing all money from a bank account works, but it blows up before it even gets to that step.

This is a problem. You designed a test to target the feature code, not the support code. You don't know anything about the support code. But, armed with this total lack of knowledge, you need to answer two sets of questions. The first one is:

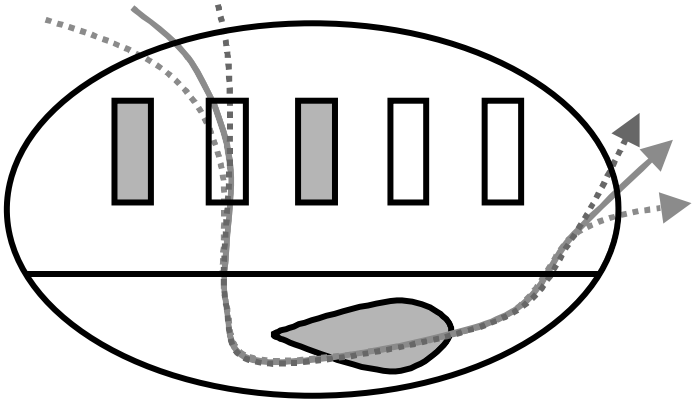
How much will the support code change? How buggy are those changes likely to be?

Unless there's a reasonable chance of changes with bugs, you won't recover your costs.

Not easy questions to answer. I'll discuss one way to go about it shortly, by means of an example.

Support code changes aren't enough. The test also must be good at detecting the bugs that result. What confidence can you have in that?

Suppose that you have three tests targeted at some particular feature. For example, one test tries withdrawing all the money in the account. Another withdraws one cent. Another withdraws all the money but one cent. How might those tests exercise the code? They may exercise the feature code differently, but they will likely all exercise the support code in almost exactly the same way. Each retrieves a record from the database, updates it, and puts it back. Here's a picture:



As far as the support code is concerned, these three tests are identical. If there are bugs to be found in the support code, they will either all find them, or none of them will. Once one of these tests has been automated, automating the other two adds negligible value (unless the feature code is likely to change in ways intended to preserve behavior).

That given, here's a question to ask when considering whether to automate a test:

If you ignore what the test does to directly fulfill its purpose, are the remainder of the test actions somehow different from those done by other tests?

In other words, does the test have seemingly irrelevant variety? Does it do things in different orders, even though the ordering shouldn't make any difference? You hope a test with variety exercises the support code differently than other tests. But you can't know that, not without a deep understanding of the code.

This is a general-purpose question for estimating a test's long-term value. As you get more experience with the product, it will be easier for you to develop knowledge and intuition about what sorts of variations are useful.

An example

By this point, you're probably screaming, "But how do I use this abstract knowledge of product structure in real life?" Here's an example that shows how I'd use everything presented in this paper so far.

Suppose I were testing an addition to the product. It's half done. The key features are in, but some of the ancillary ones still need to be added. I'd like to automate tests for those key features now. The sooner I automate, the greater the value the tests can have.

But I need to talk to people first. I'll ask these questions:

- Of the programmer: Is it likely that the ancillary features will require changes to product support code? It may be that the programmer carefully laid down the support code first, and considers the remaining user-visible features straightforward additions to that work. In that case, automated tests are less likely to have value. But the programmer might know that the support code is not a solid infrastructure because she was rushed to get this alpha version finished. Much rework will be required. Automated tests are more clearly called for. Or the programmer might have no idea - which is essentially the same thing as answering "yes, support code will change".

- Of the product manager or project manager: Will this addition be an important part of the new release? If so, and if there's a hot competitive market, it's likely to change in user-visible ways. How much has the user interface changed in the past, and why should I expect it to change less often in the future? Are changes mostly additions, or is existing behavior redone? I want a realistic assessment of the chance of change, because change will raise the cost of each automated test and shorten its life.

- Of the person who knows most about the test automation toolset: What's its track record at coping with product change? What kinds of changes tend to break tests (if that's known)? Are those kinds possibilities for the addition I'm testing?

I should already know how many manual tests automating a test will cost me, and I now have a very rough feel for the value and lifetime of tests. I know that I'll be wrong, so I want to take care not to be disastrously wrong. If the product is a shrinkwrap product with a GUI (where the cost of automation tends to be high and the lifetime short), I'll err on the side of manual tests.

But that doesn't mean I'll have no automated tests. I'll follow common practice and create what's often called a "smoke test", "sniff test", or "build verification suite". Such suites are run often, typically after a daily build, to "validate the basic functionality of the system, preemptively catching gross regressions" [McCarthy95]. The smoke test suite "exercises the entire system from end to end. It does not have to be an exhaustive test, but it should be capable of detecting major problems. By definition, if the build passes the smoke test, that means that it is stable enough to be tested and is a good build." [McConnell96]

The main difference between my smoke test suite and anyone else's is that I'll concentrate on making the tests exercise different paths through the support code by setting them up differently, doing basic operations in different orders, trying different environments, and so forth. I'll think about having a variety of "inessential" steps. These tests will be more work to write, but they'll have more value. (Note that I'll use variety in even my manual testing, to increase the chance of stumbling over bugs in support code.)

My smoke test suite might also be smaller. I may omit tests for some basic features if I believe they wouldn't exercise the support code in a new way or their lifetime will be shorter than average.

I now have an automated test suite that I know is suboptimal. That's OK, because I can improve it as my knowledge improves. Most importantly, I'll be tracking bug reports and fixes, both mine and those others file against the code under test. From them I'll discover important information:

- what seemingly irrelevant factors actually turn up bugs. I might discover through a bug report that the number of other people who've withdrawn money is, surprisingly, relevant because of the structure of the support code. I now know to include variations in that number in my tests, be they manual or automated. I've gained a shallow understanding of the support code and the issues it has to cope with. Shallow, but good enough to design better tests.

- where bugs lie. By talking to the developer I'll discover whether many bugs have been in the support code. If so, it's likely that further bugs will be, so I'll be encouraged to automate.

- how stable the code's behavior really is. I'm always skeptical of claims that "this interface is frozen". So if it really was, I'll have underautomated. After the smoke test is done, I'll keep testing - both creating new tests for the key features and testing the ancillary features as they are completed. As I grow more confident about stability, I'll automate more of those tests (if there's enough time left in the project for that to be worthwhile).

Over time, my decisions about whether to automate or run a test manually will get better. I will also get a larger test suite.

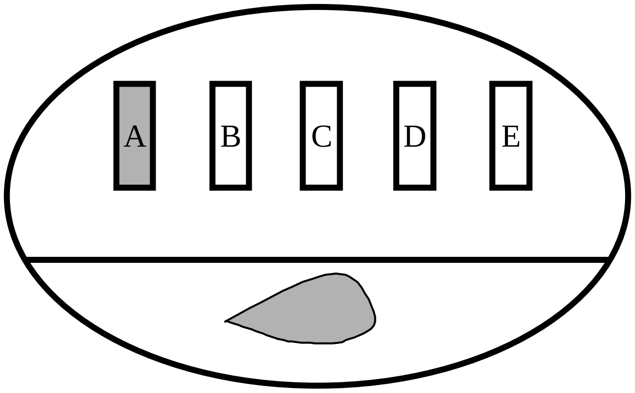
When feature code changes, I hope that the smoke tests are unaffected. If the change passes the smoke test, I must now further test it manually. That involves rerunning some older manual tests. If they were originally documented in terse, checklist form, they can readily be rerun. These new executions won't be the same as the earlier ones; they may be rather different. That's good, as they can serve both to check regressions and perhaps find bugs that were there all along. I must also create new tests specific to the change. It is unlikely that old tests will do a thorough job on code that didn't exist when they were first designed. I'll decide whether to automate the new tests according to the usual criteria.

Sometimes, product changes will break tests. For example, I'd expect the development for a new major release to break many old tests. As I looked at each broken test, I'd decide anew whether automation was justified. I have seen testing organizations where vast amounts of effort are spent keeping the automated test suites running, not because the tests have value, but because people really hate to throw them away once they're written.

Example: Automated product-level tests after code-level testing

What if the code under test contains no untargeted code? What if there are specific tests, including code-level tests, for every line of code in the system, including support code that would be invisible to a product-level "black box" tester? In that case, fewer product-level tests are needed. Many bugs in changed support code will be caught directly (whether by automated tests or by manual tests targeted at the changes). A few are still useful.

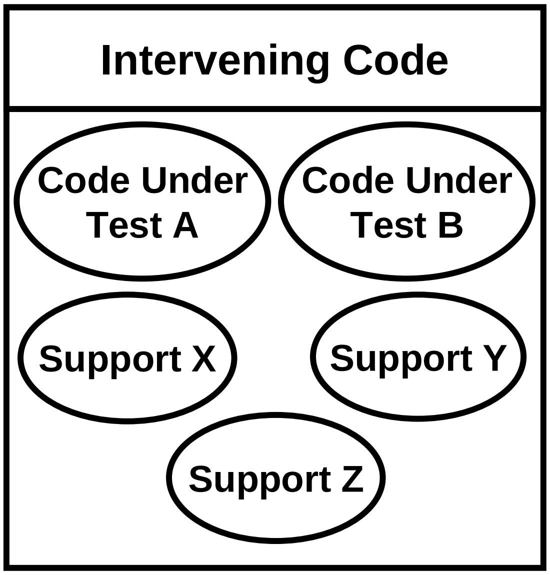
A common type of bug is the missing change, exemplified by this picture:



Feature code A has been changed. Some support code has changed as well. Some facet of its behavior changed to support A's new behavior. Unbeknownst to the programmer, feature code E also depended on that facet of behavior. She should have changed E to match the support code change, but overlooked the need. As a consequence, E is now broken, and that brokenness will not be caught by the most thorough tests of the support code: it works exactly as intended. A test for E will catch it. So an automated test for E has value, despite the thoroughness of support code testing.

More about code structure: the task-driven automated test

My picture of the structure of the code was oversimplified. Here's a better one:



Not all support code is part of the code under test. Hidden from external view will be large blocks of support code that aid many diverse features. Examples include memory management code, networking code, graphics code, and database access code.

The degree to which such support code is exercised by feature tests varies a lot. I have been in situations where I understood the role played by Support X. I was able to write tests for Code Under Test A that exercised both it and Support X well. That, combined with ordinary smoke tests for Code Under Test B (which also used X), gave me reasonable confidence that broken changes to Support X would be detected.

That's probably the exception to the rule. The question, then, is once again how you can exercise well both support code and interactions between support code and feature code without knowing anything about the support code.

The situation is exacerbated by the fact that support code often contains persistent state, data that lasts from invocation to invocation (like the records in a database). As a result, a bug may only reveal itself when Code Under Test A does something with Support Y (perhaps changing a database record to a new value), then Code Under Test B does something else that depends - incorrectly - on that changed state (perhaps B was written assuming that the new value is impossible).

The tests you need are often called task-driven tests, use-case tests, or scenario tests. They aim to mimic the actions a user would take when performing a typical task. Because users use many features during normal tasks, such tests exercise interactions that are not probed when each feature is tested in isolation. Because scenario tests favor common tasks, they find the bugs most users will find. They also force testers to behave like users. When they do, they will discover the same annoyances and usability bugs that will frustrate users. (These are cases where the product performs "according to spec", but the spec is wrong.)

This style of testing is under-documented compared to feature testing. My two favorite sources describe the use of scenarios in product design and marketing. [Moore91] discusses target-customer characterization. [Cusumano95] describes activity-based planning. See [Jacobson92] for an early description of use cases, including a brief discussion of their use in testing. See also [Ambler94], [Binder96], and [Beizer90] (chapter 4).

Some subset of these tests should be automated to catch interaction bugs introduced by later change. As before, how many should be automated depends on your expectation about how many such bugs will be introduced in the future and not caught by feature testing, weighed against how many such bugs you might find now by trying more of the manual task-driven tests.

Note that task-driven tests are more likely to be broken by changes to the product, because each one uses more of it than any single feature test would. You're likely to automate fewer of them than of feature tests.

Secondary Considerations

Here are some other things I keep in mind when thinking about automation.

- Humans can notice bugs that automation ignores. The same tools and test libraries that filter out irrelevant changes to the UI might also filter out oddities that are signs of bugs. I've personally watched a tester notice something odd about the way the mouse pointer flickered as he moved it, dig into the symptom further, and find a significant bug. I heard of one automated test suite that didn't notice when certain popup windows now appeared at X,Y coordinates off the visible screen, so that no human could see them (but the tool could).

- But, while humans are good at noticing oddities, they're bad at painstaking or precise checking of results. If bugs lurk in the 7th decimal place of precision, humans will miss it, whereas a tool might not. Noel Nyman points out that a tool may analyse more than a person can see. Tools are not limited to looking at what appears on the screen; they can look at the data structures that lie behind it.

- The fact that humans can't be precise about inputs means that repeated runs of a manual test are often slightly different tests, which might lead to discovery of a support code bug. For example, people make mistakes, back out, and retry inputs, thus sometimes stumbling across interactions between error-handling code and the code under test.

- Configuration testing argues for more automation. Running against a new OS, device, 3rd party library, etc., is logically equivalent to running with changed support code. Since you know change is coming, automation will have more value. The trick, though, is to write tests that are sensitive to configuration problems - to the differences between OSes, devices, etc. It likely makes little sense to automate your whole test suite just so you can run it all against multiple configurations.

- If the test finds a bug when you first run it, you know you'll need to run it again when the bug is ostensibly fixed. That in itself is probably not enough to tip the scales toward automation. It does perhaps signal that this part of the code is liable to have future changes (since bugs cluster) and may thus motivate you to automate more tests in this area, especially if the bug was in support code.

- If your automation support is strong enough that developers can rerun tests easily, it may be faster to automate a test than write a detailed description of how to reproduce a bug. That level of automation support is rare, though. Developers sometimes have difficulty using the test tools, or don't have them installed on their machine, or can't integrate them with their debugger, or can't find the test suite documentation, or have an environment that mysteriously breaks the test, and so forth. You can end up frustrating everyone and wasting a lot of time, just to avoid writing detailed bug reports.

- It's annoying to discover a bug in manual testing and then find you can't reproduce it. Probably you did something that you don't remember doing. Automated tests rarely have that problem (though sometimes they're dependent on parts of the environment that change without your noticing it). Rudimentary tracing or logging in the product can often help greatly - and it's useful to people other than testers. In its absence, a test automation tool can be used to create a similar log of keystrokes and mouse movements. How useful such logs are depends on how readable they are - internally generated logs are often much better. From what I've seen, many testers could benefit greatly from the lower-tech solution of taking notes on a pad of paper.

- An automated test suite can explore the whole product every day. A manual testing effort will take longer to revisit everything. So the bugs automation does find will tend to be found sooner after the incorrect change was made. When something that used to work now breaks, a first question is "what code changes have been made since this last worked?" Debugging is much cheaper when there's only been a day's worth of changes. This raises the value of automation.

  Note that the really nasty debugging situations are due to interactions between subsystems. If the product is big, convoluted, and hard to debug, automated tests will have more value. That's especially true of task-driven tests (though, unfortunately, they may also have the shortest expected life).

- After a programmer makes a change, a tester should check it. That might include rerunning a stock set of old tests, possibly with variations. It certainly includes devising new tests specifically for the change. Sometimes communication is poor: testers aren't told of a change. With luck, some automated tests will break, causing the testers to notice the change, thus be able to test it and report bugs while they're still cheapest to fix. The smaller the automated test suite, the less likely this will happen. (I should note that test automation is an awfully roundabout and expensive substitute for basic communication skills.)

- Because test automation takes time, you often won't report the first bugs back to the programmer as soon as you could in manual testing. That's a problem if you finally start reporting bugs two weeks after the programmer's moved on to some other task.

- It's hard to avoid the urge to design tests so that they're easy to automate, rather than good at finding bugs. You may find yourself making tests too simplistic, for example, because you know that reduces the chance they'll be broken by product changes. Such simplistic tests will be less likely to find support code bugs.

- Suppose the product changes behavior, causing some automated tests to report failure spuriously. Fixing those tests sometimes removes the spurious failures but also greatly reduces their ability to find legitimate bugs. Automated test suites tend to decay over time.

- Automated tests, if written well, can be run in sequence, and the ordering can vary from day to day. This can be an inexpensive way to create something like task-driven tests from a set of feature tests. Edward L. Peters reminded me of this strategy after reading a draft of this paper. As Noel Nyman points out, automated tests can take advantage of randomness (both in ordering and generating inputs) better than humans can.

- You may be designing tests before the product is ready to test. In that case, the extra time spent writing test scripts doesn't count - you don't have the alternative of manual testing. You should still consider the cost of actually getting those scripts working when the product is ready to test. (Thanks to Dave Gelperin for this point.)

- An automated test might not pay for itself until next release. A manual test will find any bugs it finds this release. Bugs found now might be worth more than bugs found next release. (If this release isn't a success, there won't be a next release.)

Summary

This paper sketches a systematic approach to deciding whether a test should be automated. It contains two insights that took me a long time to grasp - and that I still find somewhat slippery - but which seem to be broadly true:

1. The cost of automating a test is best measured by the number of manual tests it prevents you from running and the bugs it will therefore cause you to miss.

2. A test is designed for a particular purpose: to see if some aspects of one or more features work. When an automated test that's rerun finds bugs, you should expect it to find ones that seem to have nothing to do with the test's original purpose. Much of the value of an automated test lies in how well it can do that.

For clarity, and to keep this paper from ballooning beyond its already excessive length, I've slanted it toward particular testing scenarios. But I believe the analysis applies more broadly. In response to reader comments, I will discuss these issues further on my web page, <http://www.stlabs.com/marick/root.htm>.

Acknowledgements

A version of this paper was briefly presented to participants in the third Los Altos Workshop on Software Testing. They were Chris Agruss (Autodesk), James Bach (SmartPatents), Karla Fisher (Intel), David Gelperin (Software Quality Engineering), Chip Groder (Cadence Design Systems), Elisabeth Hendrickson (Quality Tree Consulting), Doug Hoffman (Software Quality Methods), III (Systemodels), Bob Johnson, Cem Kaner (kaner.com), Brian Lawrence (Coyote Valley Software Consulting), Thanga Meenakshi (Net Objects), Noel Nyman (Microsoft), Jeffery E. Payne (Reliable Software Technologies), Bret Pettichord (Tivoli Software), Johanna Rothman (Rothman Consulting Group), Jane Stepak, Melora Svoboda (Facetime), Jeremy White (CTB/McGraw-Hill), and Rodney Wilson (Migration Software Systems).

Special thanks to Dave Gelperin, Chip Groder, and Noel Nyman for their comments.

I also had students in my University of Illinois class, "Pragmatics of Software Testing and Development", critique the paper. Thanks to Aaron Coday, Kay Connelly, Duangkao Crinon, Dave Jiang, Fred C. Kuu, Shekhar Mehta, Steve Pachter, Scott Pakin, Edward L. Peters, Geraldine Rosario, Tim Ryan, Ben Shoemaker, and Roger Steffen.

Cem Kaner first made me realize that my earlier sweeping claims about automation were rooted in my particular environment, not universal law.

References

[Ambler94]
Scott Ambler, "Use-Case Scenario Testing," Software Development, July 1995.

[Beizer90]
Boris Beizer, *Software Testing Techniques (2/e),* Van Nostrand Reinhold, 1990.

[Binder96]
Robert V. Binder, "Use-cases, Threads, and Relations: The FREE Approach to System Testing," Object Magazine, February 1996.

[Cusumano95]
M. Cusumano and R. Selby, *Microsoft Secrets,* Free Press, 1995.

[Jacobson92]
Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, 1992.

[Kaner97]
Cem Kaner, “Improving the Maintainability of Automated Test Suites,” in *Proceedings of the Tenth International Quality Week* (Software Research, San Francisco, CA), 1997. (http://www.kaner.com/lawst1.htm)

[Moore91]
Geoffrey A. Moore, *Crossing the Chasm,* Harper Collins, 1991.

[Pettichord96]
Bret Pettichord, “Success with Test Automation,” in *Proceedings of the Ninth International Quality Week* (Software Research, San Francisco, CA), 1996. (http://www.io.com/~wazmo/succpap.htm)