Science and Technology Facilities Council

Technical Report
DL-TR-2021-002

Software Outlook – Testing: overview and best practices

B Mummery

August 2021

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at:
https://epubs.stfc.ac.uk

DOI: 10.5286/dltr.2021002

ISSN 1362-0207

Software Outlook

Testing

Overview and Best Practices

Dr Benjamin Mummery
STFC Hartree Centre
August 2021

Contents

Figures

1 Introduction

In our experience, many developers have an idea of testing as something akin to flossing – everyone knows they *should* be doing it every single day, but it's annoying and uncomfortable and doesn't present an immediate benefit so most people do it every now and then and leave it at that, despite the protestations of their dentist.

As such, testing is one of the most neglected parts of software development. The perceived overhead to setting it up in the first place is compounded by the ever-growing backlog of code that would need to be tested if testing were to be brought in. And for those like ourselves who don't have formal training in software development, it can be daunting to even work out where to start.

This guide aims to address these problems by presenting a (we hope) accessible introduction to how one can approach testing in a targeted and efficient manner, as well as covering some of the more esoteric aspects of testing and disambiguating the common stumbling blocks.

1.1 How to Use This Guide

By its nature this guide is written for as broad an audience as possible. As such, you will inevitably find some sections redundant, while others will be of great use. While reading from start to finish is a valid approach and will (hopefully) make sense, it's more likely that you'll want to dip in and out of specific sections as needed. In order to facilitate this, we've tried to keep the content as clearly laid out and delineated as we can.

The main sections are as follows:

- **Why Test?** (page 5) In this section we lay out the primary benefits of software testing as well as addressing some of the common reasons given against it.
- **What Test?** (page 8) Here we give an overview of the various types of testing and what purposes they serve.
- **How Test?** (page 13) This section deals with some of the approaches to implementing testing, including an overview of common testing frameworks and automation tools, as well as concepts and rules of thumb relating to maximising the efficiency of tests.
- **When Test**? (page 20) In this section we cover when various types of test are best deployed, and how to prioritise tests to make the most of finite time and resources.
- **Who Test?** (page 24) Here we discuss where the primary responsibility for various forms and stages of testing should ideally lie.

1.2 TLDR

Testing offers many benefits but primarily it saves time overall by identifying and localising bugs early in the development process, preventing unintended breakages from changes, and facilitating asynchronous development by establishing the behaviour of as yet unwritten components in advance.

Testing can be split into a hierarchy depending on the size of the unit of code being tested, and into what we have termed 'Intents' which reflect the purpose behind each test. The hierarchy starts with unit tests that examine single functions and concluding with acceptance tests that examine the behaviour of the entire codebase in the environment to which it is ultimately to be deployed. Intents range from determining whether the code even compiles or runs (Build/Run) to examining the

usability and security of interfaces and APIs. Hierarchy and Intent for two axes that define a parameter space within which any given test can be placed. Approaching thinking about tests in this way helps to keep track of what units of code have been tested, and what they've been tested for, without necessarily having to keep track of the various (and often confusing) terminology around testing.

Various options exist for organising and automating tests, any one of which will go a long way to minimise the overheads testing entails. The longer you wait before implementing a testing scheme, the greater the overheads are and the less you get to reap the benefits of having it in place. Therefore, agreeing a concrete testing method and scheme should be an assumed part of setting up any project.

100% test coverage is a blissful utopian dream and is also completely impossible to achieve on anything but the most simplistic projects (i.e. full coverage becomes impossible as soon as testing becomes useful). Consequently, it is necessary to prioritise your tests. This means weighing up the pros and cons of each test and picking the best options. However, there is no quantitative way to compare the hassle of doing test x to the monetary cost of running test y, so there is no neat equation one can use to determine which is the best test to do at any given time. Rather, it is simpler to *qualitatively* rank the various options in terms of their benefits and costs, not thinking *too* hard about it, and work your way as far down the list as time and resources allow. This has the bonus advantage that adjusting the ranking is very quick and easy, making your prioritisation flexible and able to adapt to changing situations.

The best person to *do* each test tends to be the person best positioned to articulate what needs to change if the test fails. This usually breaks down along the testing hierarchy, so as a rule of thumb:

- Unit testing should be done by developers.
- Integration testing should be done by testers and developers.
- System testing should be done by testers.
- Acceptance testing should be done by testers and users.

There will, inevitably, be exceptions and modifications to these guidelines, but they're a good place to start.

2 Why Test?

Having a clear conception of what you want or expect to get out of testing is, in our experience, a useful starting point, so we recommend reading through this section even if you already have a sense of the benefits of testing.

2.1 "If it's not tested, it's broken."

This glib little phrase, or variations on it, gets trotted out a lot in discussions of testing and development, often to the eye-rolling annoyance of the listener (ourselves included). Despite its triteness, however, this saying aims to convey a vitally important principal, so we are going to spend the next 270 words arguing in support of it.

Clearly this statement is not *literally* true. It is entirely possible for a piece of code to be written with no errors, especially if that code is relatively simple. I am, for example, very confident that a python script reading simply

```
print("Hello, world!")
```

will, in fact, run without errors. It is, however, impossible to *know* that the code works. Especially if it is to be deployed in multiple environments on multiple operating systems over a long time period. This epistemic distinction has severe knock-ons. Any further development that relies on or incorporates the untested code is, by extension, at least partially untested. Any real-world application of it is similarly uncertain.

Choosing a piece of code for any purpose, be it a single component to be included in a larger project, or a finished product being employed for a real-world application, comes down to a simple calculation: are the potential benefits of using this worth the potential risks? "What if this piece of code doesn't work" is a perennial inclusion in the latter category, and sadly your confidence in your coding abilities counts for very little here. As Christopher Hitchens would say, that which can be asserted without evidence can be dismissed without evidence. The method by which risks due to errors in code can be shown not to apply is, of course, by explicitly testing the code's behaviour.

In summary, anyone using, maintaining, or adapting your code must consider, and plan for, the potential results of that code being broken, unless you have tests to demonstrate that it is not. Or, in other words: *if it's not tested, it's broken.*

2.2 You're already doing it

Nobody we have ever met would dream of writing a piece of code and handing it off without running it at least once themselves. Indeed, running the code you're writing is so ubiquitous an action that you likely don't even think about it. It's just a part of the process of coding: tap out a few lines, run it; try a different approach, run it; go for a coffee and can't quite get your train of thought back, run the code and see where it breaks. Congratulations, you're doing unit testing!

If this is the case, then why do you need this guide? Simply put, this sort of ad-hoc approach can always be improved by taking a more organised, systematic approach. The benefits of doing so include, but are not limited to:

- Recording the tests and their results so that they can be shared with others and you can see where you have, and have not, achieved good coverage.
- Minimising the wasted effort by avoiding repeat / low value tests.

- Maximising the value of tests by assessing which tests provide the most useful information.

2.3 Writing for Tests Makes You a Better Coder

Beyond simply ensuring that your software is functional and robust, keeping testing in mind can actually serve to improve the quality of your code *even if you never run a single test*.[1] To illustrate, consider the following example: the single responsibility principal for software development states that each module, class, or function in a computer program should have responsibility over, and encapsulation of, a single part of the program's functionality. Deciding when a function you are writing needs to be split into two in order to satisfy this principal can be difficult especially in the early stages of development when things are more fluid. However, we find that keeping unit testing in the back of your mind can provide a much more concrete metric: a single responsibility function should require a single test to check that its core functionality is working correctly. If you cannot test the full functionality in one go, it is very likely that you should be splitting the different testable cases into different functions.[2] Testing also improves the design of the code - it forces the dev to think about the abstractions and how the code is used from the calling-code perspective.

In other words, the principals of software testing and good software development are synergistic. Doing one supports the other, and vice versa.

2.4 Testing Serves the Design Process

Test-driven development (TDD) is the practice of writing tests for your code first, and then developing the code until it passes the test. While this is not always a suitable scheme, especially when developing in a highly experimental context, it does illustrate a useful paradigm: *decide what you need your code to do **before** you start writing it*. This approach can be applied at all levels – everything from the entire codebase down to individual functions can, and should, have their final functionality cemented (and, importantly, recorded) as early as possible. This is invaluable for keeping sight of the bigger picture, spotting potential problems early in the development process, and, perhaps most importantly, facilitating collaborative development by making explicit the parameters within which each unit of development lies. You can develop around functions or modules that don't even exist yet because you know how the finished unit will behave.

As an almost incidental bonus, having taken this approach makes it almost trivially easy to then add in tests that express the requirements for your code.

2.5 Testing Saves Time

This is really the most important point on this list, and we could probably have begun and ended here and left it at that. One of the most common reasons for not testing that we have heard in the process of researching this guide is that it presents too great a demand for time and effort, on top of the already stringent existing demands of software development. We argue that this perception, while understandable, is an artefact of the view of testing as a monolithic, alien object that lies *external* to the practice of software development. And, indeed, when testing is treated in this manner it *is* a

[1] Although, to be clear, we do not advocate for this approach, and you should definitely run tests.

[2] This does not mean only one test should exist for each component – code needs to be tested for multiple things each of which takes its own test. However, the behaviour of the function under normal circumstances should be summarisable in a single test.

significant overhead. However, as we will cover below (Section 4), testing is at its best when it is incorporated as an intrinsic part of the development process.

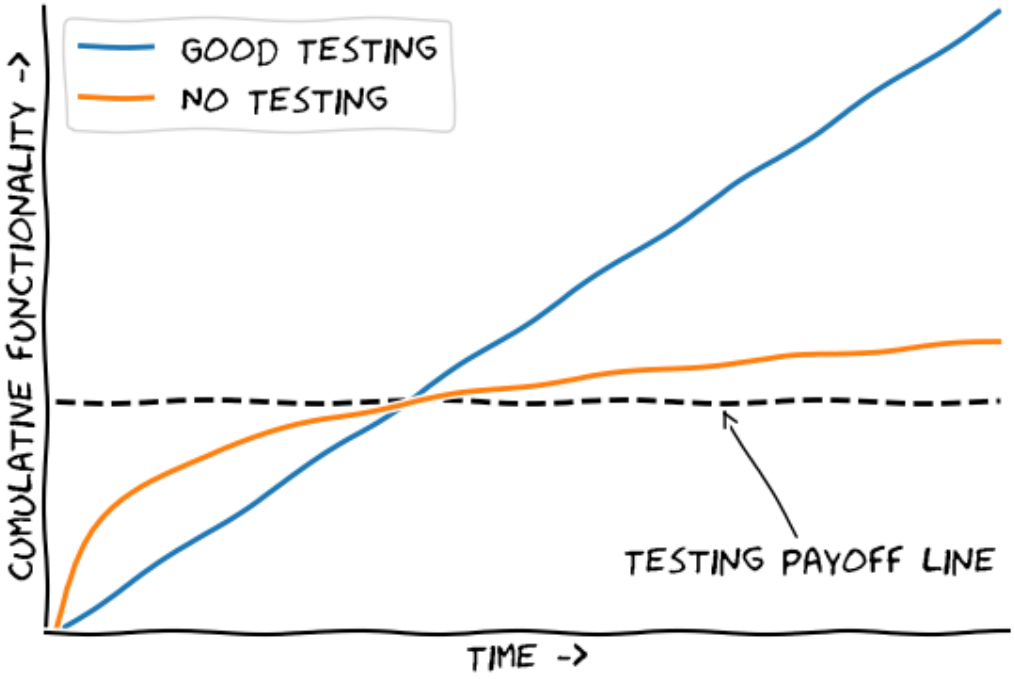

FIGURE 2-1: THE CUMULATIVE BENEFIT OF TESTING OVER TIME

Incorporating testing into your development practices almost invariably results in a net *saving* in terms of time and effort. This is because running tests once they are implemented is very low effort (to the point where it can be largely automated) and means that errors are spotted almost immediately. Effective unit testing isolates the error usually to the point of an individual function. This means fewer bug fixes, fewer clashes between developers, and much less time spent tracing a bug through the code. Like flossing, testing is about prevention rather than correction – a small discomfort now saves a lot of pain later.

In this regard, testing dovetails nicely with the concept of Design Stamina.[3] Through this lens, neglecting testing accrues a "Technical Debt" comprised of all of the tiny conflicts, bugs, and 'temporary' hacks that the system has accumulated. Adding new functionality is both harder, since you will be forced to deal with at least some of the accumulated technical debt, and makes adding *further* functionality harder as well, since your changes will add to the total technical debt. This relationship is illustrated in Figure 2-1. Since neglecting testing leaves more time for adding functionality, progress is initially faster without it. However, as the technical debt accumulates, progress slows (as illustrated by the orange line in the figure). Conversely, taking the time to invest in testing (and other good design practices) keeps the technical debt low, allowing development to continue at a pretty-much constant rate that rapidly outstrips the untested case. If your project is *very* limited in scope (for example a proof-of-concept) and you are *certain* that no further development is ever going to depend on it, then you can operate more efficiently without testing. In *all* other cases, your project will cross the "testing payoff line"[4] and implementing testing will pay dividends.

[3] For a more detailed discussion of Design Stamina, see https://martinfowler.com/bliki/DesignStaminaHypothesis.html

[4] Note that the threshold is a value on the functionality axis, not the time axis. The complexity of the software is the deciding factor, rather than the amount of time being spent on it.

3 What Test?

The vast majority of resources we have found that discuss what different types of tests exist take the form of long, exhaustively detailed lists of subtly distinct terminology. While these are very useful for ensuring a common language when talking about tests, we find them to be particularly off-putting to the uninitiated [5] and suspect that documents of this type are one of the main reasons that implementing testing can seem particularly daunting.

For completeness, we include a glossary of testing terminology in Appendix I to be used as a reference. However, an in-depth discussion of the taxonomy of testing methods would likely be lengthy, dry, and ultimately unhelpful in the context of this guide.

In lieu of the traditional lists, we will instead discuss the various forms of tests in the context of two aspects: The Testing Hierarchy and what we shall be referring to as Testing Intents[6]. Broadly speaking, the former tells you what unit of code is to be tested, the latter tells you what you are testing it for.

3.1 The Testing Hierarchy

Tests fall into four categories: Unit Testing, Integration Testing, System Testing, and Acceptance Testing. These form a hierarchy with each stage contingent on the previous level of testing being passed. For intuitive understanding, we will consider these categories in the context of manufacturing a retractable ballpoint pen.

One huge advantage of this hierarchy is that the structure and ordering of the tests mirrors that of the development process. Unit Testing correlates with the development of individual components, For collaborative development, a unit test acts as an explicit contract that states what the code it tests must do. As writing a test is faster than writing the code it measures, this can often be done first. Development on other components that depend upon or interact with the unit can then proceed on the basis of the test, before the unit itself is complete, with the certain knowledge of how the unit will behave.

Unit tests are easy to write, easy to maintain, usually cheap to run, and present a prime candidate for automation. Since each test is concerned with a specific section of code, it is trivial to know which tests need to be rerun when a modification is made, and which are still current. This also makes it simple to determine how complete your test coverage is. Finally, the specificity of Unit Testing renders the results very transparent – when a test fails, the location of the fault is highly localised and easy to find.

Integration Testing with that of combinations of components, System Testing with that of complete systems, and Acceptance Testing with the final delivery. This synchronicity means that for any alteration to the codebase, you can see at a glance what level of testing you need to be concerned with to confirm that your modification works as expected.

[5] Ourselves included.

[6] In researching this guide, we were unable to find an existing term for this concept. If you believe you know of one, please let us know!

Unit Testing

Test an individual component in isolation.

Unit testing is the most basic form of testing. It verifies individual functions, methods, and modules in isolation. In our pen analogy, this would mean examining the spring, catch, ball, ink reservoir, clip, etc. on their own, confirming that they meet manufacturing requirements.

Unit testing can, and probably should, be built into your codebase from the ground up. For collaborative development, a unit test acts as an explicit contract that states what the code it tests must do. As writing a test is faster than writing the code it measures, this can often be done first. Development on other components that depend upon or interact with the unit can then proceed on the basis of the test, before the unit itself is complete, with the certain knowledge of how the unit will behave.

Unit tests are easy to write, easy to maintain, usually cheap to run, and present a prime candidate for automation. Since each test is concerned with a specific section of code, it is trivial to know which tests need to be rerun when a modification is made, and which are still current. This also makes it simple to determine how complete your test coverage is. Finally, the specificity of Unit Testing renders the results very transparent – when a test fails, the location of the fault is highly localised and easy to find.

Integration Testing

Test the communication/interaction between two components.

Integration testing operates on the level of subsystems and focusses on testing the interfaces between modules. For the case of our pen, this would be the equivalent of confirming that the various pieces of the body screw together snugly, and that the rubberised grip does not rotate freely around the body of the pen.

As with Unit Testing, specificity is vital. Each integration test should be concerned with one specific interaction, minimising the potential ambiguity of the output.

System Testing

Test a specific behaviour of the complete system.

Once the final component has been integrated (and integration testing is complete), the system can be tested in its entirety. The full suite of functions and expected behaviours can be examined. This is typically performed by the developers in a mock-up of the environment in which the system is to be deployed.

In our pen analogy, system testing is the equivalent of taking the fully assembled pen through its paces – trying it on different types of paper, checking that it fits a range of hand sizes, and that the body, spring, catch, etc. function together to deploy and retract properly.

Acceptance Testing

Test the entire system against user needs.

Acceptance testing is the final stage of testing. Rather than testing for strict functionality as in the case of system testing, this confirms that the system meets the needs of the user, be they functional, security, usability, whatever. This is where you check that the software doesn't just work, it works in the way that the user requested, on their systems, and to their satisfaction.

In our pen example, this is where you get potential customers, marketing execs, and anyone else who might need to sign off[7] on the pen to agree that yes, it is the correct weight, it does write properly in their preferred brand of notebook, and it is indeed the perfect shade of rose pink.

3.2 Testing Intents

While The Testing Hierarchy (3.1) describes what part of the code is to be tested, the Intent expresses what you are testing for. Unlike the hierarchy these are not a strict sequence, although there are often advantages to be gained from certain orderings.

The use of this system also helps to keep tests *semantic*. As with well-written functions, a well-written test tests a single concept about the software, an idea that meshes nicely with the concept of Testing Intents. The advantage of semantic testing is that it maximises the information gained from a failed test. Rather than simply learning that *something* is wrong, you are told what specific expected behaviour of the software has not been met.

Note that the intents we list here are, in our experience, the most commonly applicable, but they are by no means an exhaustive parameter space, nor are they categorical in their scope. These are a loose conceptual guideline to aide in breaking down the monolith of 'Testing' into usable chunks.

Build/Run

Does the code crash?

The most basic level of test possible: does the code run and produce results. This is less a formal test and more of a sanity check to confirm that further testing is worthwhile. The most widely used form of this is Smoke Testing, an analogue of the engineering practice of "turn it on and see if it catches fire". However, this can be expanded to include techniques such as trying a range of input parameters to confirm that edge cases do not cause errors.

To continue testing our pen, this would be the equivalent of giving the spring an experimental squeeze to see if it snaps before you take the step of putting it into your carefully calibrated and expensive to use Automatic Spring Springiness Testing Machine[8], or doing a few random scribbles with the first prototype pen.

Regression

Does the code do the same thing it did yesterday?

Regression tests compare the behaviour of their subject to the behaviour of its previous versions. The primary use of these tests is to detect when previously functional code has been hampered by subsequent changes. Note that this is explicitly agnostic to the *correctness* of the behaviour, these tests care only about the consistency over time.

In our pen example, this would be comparing the mechanical resistance[9] of the new batch of springs to the old batch to see if there is a difference.

Analytical Result

Does the code give the right answer?

[7] Pun intended
[8] Patent pending
[9] As determined, of course, by the Automatic Spring Springiness Testing Machine

Tests that fall into this category confirm that for a given input, the code being tested produces the correct output. This can include testing that the code produces known correct results on prepared datasets, and that edge cases give correct responses.

Continuing the pen example from above, this would be comparing the ink mixture we have developed to the colour specified in the initial design to make sure that we're getting the required resultant shade.

Performance

Is the code suitably efficient?

Performance tests are not concerned with what the code does, but rather how it does it. Specifically, does the code fulfil the operational requirements under a specific workload. This can include examining how the code scales (either up or down), how changes to the configuration impact the performance, how responsive interfaces remain under load, and even sequentially ramping up the load on the software to determine the upper limit with which it can cope.

Performance testing can be used to demonstrate that the software meets whatever performance criteria are required, to indicate where effort might be best spent by identifying areas of the system or workflow that have the most to gain in terms of efficiency, or to compare two viable approaches to determine which offers the best performance.

For our pen, this would mean testing how long one can continuously write with it before the ink runs out, determining the range of writing angles and pressures that are viable, and seeing how many clicks you can go through before the spring breaks.

Usability

Can humans actually use the code?

Usability testing examines how well the software relates to the people with whom it will interact. The most common way to think of this is in the context of graphical user interfaces (GUIs), where how a human intuitively interprets abstract symbols or unfamiliar jargon, how layout and colour convey information to the user, and how well the interface meets accessibility requirements such as adapting to colour blindness, partially sighted users, or screen readers, are considerations that are almost universal. However, the Usability intent extends to any human interaction with the code, including whether documentation and help hints are clear and understandable, how self-explanatory command-line arguments and options are, and how well errors and warnings are explained to the user. It also includes how well any APIs for the system are implemented and how easy they are to use.

This is not limited to external interactions with the software. Usability is a factor for any interaction with the code, including maintenance, ongoing development, or simply skimming through the source code to understand exactly how it functions. As such, documentation, formatting, and intuitive variable naming all have a bearing.

To finish out the pen analogy that we have been gradually driving into the ground, usability testing is analogous to looking at how easy it is for a user to grasp the correct usage of the pen. This may sound trite, however the frustration that comes from a pen that is *almost* symmetrical so that you cannot effortlessly tell which end you write with, or which feels like you should click when you should really twist, is almost impossible to adequately put into words. It would also involve looking at how easy the pen is to reconstruct after you've absent-mindedly dismantled it during a dull meeting.

3.3 Overview, or "Zen and the Art of Software Testing"

The Hierarchy level vs intent scheme we've described above does a passable job at laying out the parameter space that individual types of test populate. Sadly, as a consequence of the language around testing being almost invariably descriptive rather than prescriptive, many types of test overlap or buck these simple axiomatic categories.[10]

Rather than get bogged down in discussing whether the commonly understood definition of Unit Testing as a type of test aligns perfectly with the definition of Unit Testing as a hierarchy level, and debating the exact functional difference between Regression Testing and Analytical Result testing, we suggest a far more zen approach:

Do not worry about what the test you are doing is called, rather worry about what it is testing, and what it is testing for.

The Hierarchy level - intent matrix is best used not to taxonomize the various types of test, but rather as a mental model of what your testing needs to address. Any individual test will test a certain unit of code (thereby falling into the testing hierarchy) for some property (therefore corresponding to an intent), and the matrix allows you to conceptualise where gaps in your testing scheme might lie. The matrix itself isn't even set in stone: you can add or adjust categories on either axis to suit your needs. Either way it provides a tool to visualise your testing coverage.

As a visual aide we present below an example matrix populated with a few of the more common testing types. For a full explanation of these types see Appendix I.

		Hierarchy level			
		Unit	**Integration**	**System**	**Acceptance**
Intent	**Build**	Smoke Test			
	Regression				
	Analytical Result	Assert Test			Operational Test
	Performance				
	Usability				Accessibility Test

[10] This is one of the reasons why we have banished a discussion of individual test types to the appendix rather than include them here.

4 How Test?

An enormous number of frameworks and tools exist to support testing practices, and we lack the space to go into them all in detail here. We will, however, cover some of the more widely used tools and give an overview and comparison of these tools.

4.1 Automation (CI/CD)

The time and effort cost of testing is the primary barrier to entry for the vast majority of users. Luckily, the majority of tests are a) useful to run repeatedly over the course of development, and b) simple to automate. Assuming that you are already using a version control system such as Git or Subversion, you should be in the habit of making regular (and sensible) commits.[11] Continuous Integration, Deployment, and Delivery (CI/CD) systems exist that integrate with VCSs, allowing you to run a suite of tests automatically with every commit, and, depending on your preference, passing some or all of your test suite can be a requirement for permitting the commit to actually be made. This means that once the system is in place, the additional overheads for unit testing are kept very minimal.

CI/CD actually has a broader scope than simply automating testing – many common tasks can be incorporated into the pipeline, and there is a wide degree of customisability. A CI/CD pipeline can be as simple as running a set of unit tests on your local machine before each commit, or as complex as deploying a suite of virtual machines on a cloud server to carry out a gauntlet of testing, profiling, and formatting, before carrying out the actual deployment of the (passing) software, all without you needing to lift a finger.

4.1.1 Example CI/CD Tools

The below are some of the more common CI/CD tools currently used. These tools provide a framework for automating many common tests, although none can fully eliminate the need for human intervention.

CircleCI

CircleCI is primarily cloud-based, freeing the user from the need to host a server in order to run it, although an on-prem solution is available to those who need it. It integrates smoothly with both Github and Bitbucket, making it trivial to have a CI/CD pipeline that is triggered on commit. Out of the box it supports Go (Golang), Haskell, Java, PHP, Python, Ruby/Rails, and Scala, and when using CircleCI's provided cloud hosting testing environments can be hosted on AWS, Azure, Heroku, or Docker. Alerts can be delivered through services including Jira, HipChat, and Slack. CircleCI uses a simple YAML configuration file, making it comparatively quick and easy to get set up.

The free plan supports Ubuntu (12.04, 14.04) and Windows environments, however those looking to test on MacOS will require one of the paid plans. These also provide concomitant increases in the permitted number of concurrent jobs, with the free plan allowing for a single job to be run at any time, and in the total usage limits[12]. They also provide the facility to cache docker layers, meaning that subsequent CircleCI runs can reuse the image rather than rebuilding each time. On large projects this presents a significant time saving.

[11] For our companion guide on the use of version control systems, see https://epubs.stfc.ac.uk/work/47984368

[12] CircleCI makes use of a weekly-resetting credits system that helps to both simplify calculating what you can and cannot do in a given week, and to obscure the actual monetary cost of any individual action.

CircleCI's out-of-box language support is comparatively limited, which can present a considerable hurdle to projects using unsupported languages. Customisations are possible to allow bespoke support, however implementing these can often require third-party software.

Above the free option, the Performance plan starts at $15/month for three users, allows credits to be purchased as in blocks of 25,000 for $15, and permits up to 80 concurrent jobs. Additional users can be added at $15/month each. The Scale plan dispenses with the standardised pricing model, allowing the user to negotiate a plan that exactly meets their requirements.

Travis CI

Travis CI is very similar to CircleCI in overall design[13]. It is webhosted, uses a YAML configuration file that makes it simple to get set up and running, integrates with Github and Bitbucket, and supports using Docker to run tests. The primary differences are the out of box language support, which in Travis CI's case is significantly broader than that offered by CircleCI, encompassing Android, C, C#, C++, Clojure, Crystal, D, Dart, Elixir, Elm, Erlang, F#, Generic, Go, Groovy, Haskell, Haxe, Java, Javascript (with Node.js), Julia, Matlab, Minimal, Nix, Objective-C, Perl, Perl6, PHP, Python, R, Ruby, Rist, Scala, Smalltalk, Swift, and Visual Basic.

Travis CI's pricing plan offers a few more options than that of CircleCI, but in general tends to work out on the slightly more expensive side.

Travis CI has two pricing models which are billed depending on either concurrency or usage. For concurrency, a series of tiered plans is provided depending on the number of concurrent jobs required. These plans allow for unlimited Windows, Linux, or FreeBSD build time, with MacOS build time requiring an additional investment of approximately $0.03 per build minute.[14] Usage-based plans have either no or very high limits on concurrency, but use a system of credits to bill the user per unit of build time. They also place limits on the number of users by billing the plan holder for a license fee for each unique user who triggers builds in a given billing period. How much this licence fee will set you back is dependent on the plan you are on.

The free plan is a default usage-based plan where the user receives a one-time allotment of 10,000 credits. These credits cannot be replenished, but for open-source projects an allotment of public-repo-specific credits may be requested from Travis CI and granted at their discretion. Paid usage-based plans are a little more obscured due to the fact that they are negotiated as bespoke solutions for your specific needs. This can only be done by contacting Travis CI directly.

The main draw of Travis CI is the out-of-box support for a Build Matrix. This allows you to specify sets of language, package, or OS versions within which you wish to test. Tests can then be automatically run with each option, or combination of options, vastly simplifying the process of ensuring backwards (or forwards) compatibility, as well as flexibility with regards to dependencies and environment.

[13] Although they differ notably in their opinions about whether their name should be one word or two.

[14] Note that this is the minimum cost as MacOS build time is paid for in credits which can be purchased in a minimum unit of 25,000 for $15 meaning that you will inevitably end up paying for more than you use, although your total wasted expenditure should never exceed $15 as you can request a refund for unused credits.

Jenkins

Jenkins is the most complex to set up of the tools discussed here, but also the most flexible. As might be expected of free software, there is less direct support available but also a greater emphasis on flexibility and self-modification. Jenkins's plugin architecture makes the act of extending Jenkins for your needs significantly simpler than it is for CircleCI or TravisCI, and is streamlined further by the cornucopia of plugins available on the Jenkins Update Center. Of particular note is the Pipeline, a suite of plugins that implements a generic CI/CD pipeline that can then be adapted to your specific requirements.

This extensibility is a double-edged sword. On the one hand, it allows Jenkins to automate almost any automatable test in almost any set of circumstances. On the other, it means that the time and effort required to get Jenkins up and running is significantly longer than that of either CircleCI or Travis CI, both of which prioritise quick and simple setup. As such, the overheads of Jenkins make it overkill for many smaller projects, especially if those can be served by the Travis CI or CircleCI free plan.

Atlassian Bamboo

Bamboo attempts to offer a competitor to Jenkins that removes a lot of the hassle of setting up and customising Jenkins at the cost of some customisability and, well, cost. Extensions that are commonly added to Jenkins via plugins such as integration with other Atlassian products Bitbucket and Jira are included by default, as is a broadly helpful GUI that assists in setting up the build and deployment stages. Bamboo is particularly geared towards the 'Deployment' part of CI/CD, allowing, for example, your build to be pushed as far as the App Store automatically if desired. It also boasts integration between Mercurial and Git branches, and built-in Git branching workflows.

Bamboo is priced on the maximum number of build agents made available to you, rather than the number of users or jobs. Unlike the other paid offerings on this list, a one-off initial payment is required in order to secure the license and use of the remote agents in perpetuity. Ongoing costs come from the optional (but highly recommended by Atlassian) software maintenance renewal. After the first year, you will no longer have access to new software releases/enhancements, Atlassian's support team, critical bug fixes, and security patches unless you cough up 50% of the then-current price of your tier every 12 months.

4.1.2 Summary and Recommendations

For smaller, simpler projects, CircleCI is often going to be the best fit. Its free plan including enterprise projects, and design for fast, easy setup are both major draws, and the time and money overheads involved in setting up and hosting something like Jenkins are unlikely to be cost-effective at this scale.

Slightly larger projects or those that require testing on multiple environments may benefit more from Travis CI than CircleCI, primarily due to the former's provision of Build Matrix support.

For larger projects, the flexibility and customisability of Jenkins, combined with the fact that it is free, makes it an obvious frontrunner. At these scales, the additional cost of hosting a server for Jenkins, and the time investment in setting it up, are more than offset by the extremely high degree of specificity that can be achieved.

All of these options benefit from the fact that automated testing scales extremely well, and provides an invaluable safety net for catching bugs, conflicts, and regressions before these make it anywhere near deployment.

<table>
<tr><th>Tool</th><th>Hosting</th><th>Supported Languages</th><th colspan="2">Price</th></tr>
<tr><td rowspan="3">CircleCI</td><td rowspan="3">Cloud- or self-hosted</td><td rowspan="3">Go, Haskell, Java, PHP, Python, Ruby, Scala.</td><td colspan="2">Free: limited weekly usage; 1 concurrent job; Docker, Linux and Windows support.</td></tr>
<tr><td colspan="2">Performance ($15/month for 3 users + $15/month/additional user): 80 concurrent jobs; Docker, Linux, Windows, MacOS support; docker layer caching</td></tr>
<tr><td colspan="2">Scale (custom pricing): customisable concurrent jobs; Docker, Linux, MacOS, Windows, GPU-Nvidia support; self-hosted runners; docker layer caching</td></tr>
<tr><td rowspan="6">TravisCI</td><td rowspan="6">Cloud-hosted, self-hosting available as part of Enterprise plan</td><td rowspan="6">Android, C, C++, C#, Clojure, Crystal, D, Dart, Elixir, Elm, Erlang, Generic, Go, Groovy, Hack, Haskell, Haxe, Java, Julia, Nix, Node_js, Objective-C, Perl, Perl6, PHP, Python, R, Ruby, Rust, Scala, Shell, Smalltalk</td><td rowspan="4">Concurrency-based: unlimited Linux, Windows, FreeBSD build time. Paid MacOS Builds via credits (1 minute of build time costs 50 credits, 25,000 credits costs $15, making for an average cost of $0.03/build minute)</td><td>$69/month: 1 concurrent job</td></tr>
<tr><td>$129/month: up to 2 concurrent jobs</td></tr>
<tr><td>$249/month: up to 5 concurrent jobs</td></tr>
<tr><td>$759/year: up to 5 concurrent jobs</td></tr>
<tr><td rowspan="2">Usage based: very high or no limit on concurrency, user-license cost per active user, build time is paid for in credits which can be purchased for $15 for 25,000.
OS | Credits / build minute
Linux | 10
FreeBDS | 10
Windows | 20
MacOS | 50</td><td>Free: starting allotment of 10,000 credits, no user limit, cannot purchase additional credits but can request open-source credits.</td></tr>
<tr><td>Enterprise: scalable depending on demand. Also offers on-prem solution. Contact Travis CI directly to obtain.</td></tr>
<tr><td>Jenkins</td><td>Self-hosted</td><td>Any</td><td colspan="2">Free</td></tr>
<tr><td rowspan="9">Bamboo</td><td rowspan="9">Cloud- or self-hosted (unlimited local agents as standard)</td><td rowspan="9">Any</td><td>0 Remote Agents, up to 10 jobs</td><td>30-day Free Trial</td></tr>
<tr><td>1 Remote Agent</td><td>$1,500 (+ $750/year)</td></tr>
<tr><td>5 Remote Agents</td><td>$4,000 (+ $2,000/year)</td></tr>
<tr><td>10 Remote Agents</td><td>$7,500 (+ $3,750/year)</td></tr>
<tr><td>25 Remote Agents</td><td>$14,500 (+ $7,250/year)</td></tr>
<tr><td>100 Remote Agents</td><td>$29,100 (+ $14,550/year)</td></tr>
<tr><td>250 Remote Agents</td><td>$72,700 (+ $36,350/year)</td></tr>
<tr><td>500 Remote Agents</td><td>$109,100 (+ $54,550/year)</td></tr>
<tr><td>1000 Remote Agents</td><td>$167,300 (+ $83,650/year)</td></tr>
</table>

4.2 Testing Frameworks

Testing frameworks are designed to allow you to write and automate a large number of tests in a simple, systematic, and cohesive fashion. While there are far too many for us to cover here, we will give a brief overview of some of the more popular options for C, C++, C#, Java, JavaScript, and Python, as well as some language-agnostic tools. The highly specific nature of unit tests makes these the most viable for automation, so many of these frameworks bias heavily towards unit testing. Sources for this information can be found in Section **Error! Reference source not found.** of **Error! Reference source not found.**.

4.2.1 C / C++

Boost.Test

Boost.Test is part of the widely-used set of boost libraries. As with boost's provision for operations such as linear algebra, it is a flexible set of interfaces providing a comparatively easy methodology for writing, organising, and executing test programs.

Cache2

Primarily a C++ unit testing framework, Cache2 also provides limited support for basic benchmarking features. Like Boost.Test it bills itself on a shallow learning curve and intuitive functionality.

Embunit

Embunit abstracts a large amount of the actual writing of tests, instead providing a graphical user interface within which you can specify a sequence of actions that constitute your test. The actual source code for the test is then generated automatically by the tool, allowing tests to be created without actually having to write code.

GoogleTest / gtest

GoogleTest is a unit-testing library based on the xUnit[15] architecture. Its main selling points are the automatic invocation of a debugger on test failure, the simplicity of its usage (often a single command is all that is required to run the test suite thanks to a suite of built-in macros) and the availability of a graphical user interface to monitor the progress of tests.

4.2.2 C#

NUnit

An open source framework of the xUnit family, NUnit was originally ported from JUnit although subsequent development has led to it being rewritten from the ground up with a host of new features. Although its primary intent is to provide unit testing for Mono and the .NET Framework, this also permits for easy unit testing of C#.

4.2.3 Java

JUnit

JUnit is the Java-flavoured[16] variant of the xUnit family. The latest release, JUnit 5, provides both the test framework itself, JUnit Jupiter, as well as JUnit Vintage which allows for backwards compatibility with tests written with previous JUnit versions.

[15] Growing out of Smalltalk's SUnit architecture, the xUnit family are a series of unit testing frameworks adapted for various programming languages. These include NUnit and JUnit which we discuss here.
[16] Mmm, coffee…

TestNG

TestNG is similar to JUnit and NUnit in a lot of ways, but aims to provide additional functionality not offered by its xUnit counterparts. Thses additional functions include annotations, more flexibility in terms of multithreading tests, support for data-driven testing, and extensibility through greater support for tools and plug-ins such as IDEA, Maven, and Eclipse.

JavaScript

HtmlUnit

HtmlUnit is not a test framework in the strictest sense, but serves as a mechanism by which a browser can be simulated for testing purposes. It is intended to be used in partnership with a framework such as JUnit or TestNG, allowing their framework to be applied to browser-based applications. A number of open-source tools such as JWebUnit and Celerity use HtmlUnit as their underlying browser.

Jest

Shipping as an NPM package that can be easily be added to an existing JavaScript project. Jest's primary selling point is its approach to "auto mocking" – Jest mocks every dependency of the tested code except the unit itself. While this means that unit testing with Jest is exceptionally clean (and saves a lot of writing time and boilerplate code in a project where you need to mock a lot), it also means that Jest is computationally expensive compared to other similar tools.

4.2.4 Python

PyTest

PyTest is the most widely used, and best supported, python testing framework. Any condition that can be expressed as an assert statement can be tested, meaning that while PyTest excels at unit testing it can, with minimal additional work, scale up to integration and system tests. Of particular note are the very low barriers to entry – tests are written in python and while there are a few specifics to learn the curve is shallow. Test discovery is handled for you, you need only title your test files "test_*.py" and PyTest will find them.

Beyond the basics, PyTest allows for a number of customisations that vastly improve quality of life such as grouping tests together into classes, specifying messages to return on test failure so that you know exactly what has gone wrong, and setting conditions for when specific tests or classes will or will not run. However, by far the most valuable of these is the pytest-xdist plugin that allows tests to be run in parallel, vastly decreasing testing time for large test suites.

4.2.5 Multi-Language

Valgrind

Although billed primarily as a memory mismanagement detector, Valgrind is more specifically a wrapper around a collection of open-source tools including Memcheck, Cachegrind, Callgrind, Massif, Helgrind, DRD, Lackey, Nulgrind, and DHAT. While regularly running Memcheck alongside unit tests to immediately detect memory mismanagement is a useful technique, Valgrind's main strength is in simplifying a broad swathe of system testing.

Gherkin/Cucumber

While most of the testing frameworks discussed above take a bottom-up approach to testing, Gherkin and Cucumber take the opposite approach – starting with the features that are essential to the end user and defining tests to meet them. Gherkin is a language designed to be readable by non-technical parties while following a set of very rigid rules. This allows behaviour to be defined without specifying how it should be implemented, making it invaluable for agreeing a concrete set of requirements

without needing technical knowledge on the part of everyone involved, making it vital for any business-driven development model. Gherkin forces behaviours to be expressed in the form of Given-When-Then statements, but permits any wording to follow each of those. For example:

```
Given Press start button
When Process is stopped
Then Process starts
```

Cucumber is a framework that allows the abstract concepts expressed in Gherkin to be linked to steps in testing code and runs those steps following the Gherkin file. This means that tests can *explicitly* be written to (and automated) to demonstrate that business requirements have been met.

Selenium

Selenium grew out of a need to test browser interactions for an internal Time and Expenses application at ThoughtWorks, but rapidly evolved into the core of an ecosystem of open source testing tool. Selenium automates browser interactions as a tool for automating the system and acceptance testing of web applications. It can also be used to automate repetitive or tedious browser tasks.

5 When Test?

While it is easy to say in the abstract that you should always be running tests, or getting the testing in place as early as possible, it can often be tricky to determine the correct time to carry out any specific test.

We discussed the various benefits of testing in Section 2, and we hope it was clear from that that the potential benefits of testing, as well as the potential detriments, increase exponentially over the development of a project. To briefly summarise:

- Frequent testing minimises the time before problems are detected, limiting the amount of further work that may now have to be changed to accommodate the fix.
- Writing an individual test is quick and easy, but the more code is written before you start the greater the backlog you then need to work through.
- Automation only needs to be set up once, and can then work passively.
- If you define tests before writing the code, development that depends on that code can use the tests as a basis, meaning they don't have to wait for the code to be complete.

As such, the second-best time to start thinking about testing is always "now", and the best time is always "yesterday".

Having decided to test in the abstract, however, we must next decide what specifically to test, and how.

5.1 The Cost-Benefit Matrix

All tests are valuable, but some are more valuable than others.

5.1.1 Rationale

As we have alluded to previously, it is impossible to test everything. Sooner or later, you will run out of time, money, imagination, or all three. As such it is necessary to prioritise tests so that you know where it is worthwhile to commit resources.

Ideally we would want a calculation of the relative value of each test. Since different tests provide different benefits and incur different costs, the value can be conceptualised as the ratio of the benefit to the cost. Once computed, we could simply work our way down the list from most to least valuable, stopping whenever we run out of resources.

Sadly, this ideal scenario bears almost no resemblance to real life. In sections below we list some of the more common considerations on each side (5.1.3, 5.1.4). The scientifically minded reader will note that few or none of these parameters share standardised units nor conversions, that the vast majority are entirely subjective, and that accurately weighing the cost/benefit for any test is consequently impossible. An alternative approach is therefore necessitated.

There are plenty of resources out there to help with test prioritisation[17], with numerous schemes existing each with their own particular focus. While these are helpful for thinking about how various considerations contribute to either the cost or benefit, in our experience they provide little help in actually evaluating those considerations. Rather than go into these specific examples, we will instead be discussing a more general technique – the cost benefit matrix.

[17] Also called "test case prioritisation"

5.1.2 Overview

The cost-benefit matrix is a rough sketch of the cost-benefit space and the relative positions of tests within it. For illustration, we include a cartoon of this space in Figure 5-1. Note that the two axes lack units – as mentioned above a quantitative measurement of either property is all but impossible. What concerns us is the *relative* positions of potential tests along these axes.

Once the matrix is populated, we can divide it into regions, which we have here labelled for how frequently the tests that fall within their boundaries should be run: "Always", "Usually", "Occasionally", and "If the opportunity arises". Using these regions allows you to avoid having to make fine distinctions between the values of tests in similar areas – if they're both in the "Always" region, they get done on every commit. Where you draw these boundaries will depend on the specifics of your circumstance, but in general we try to space them so that they are more-or-less evenly populated.

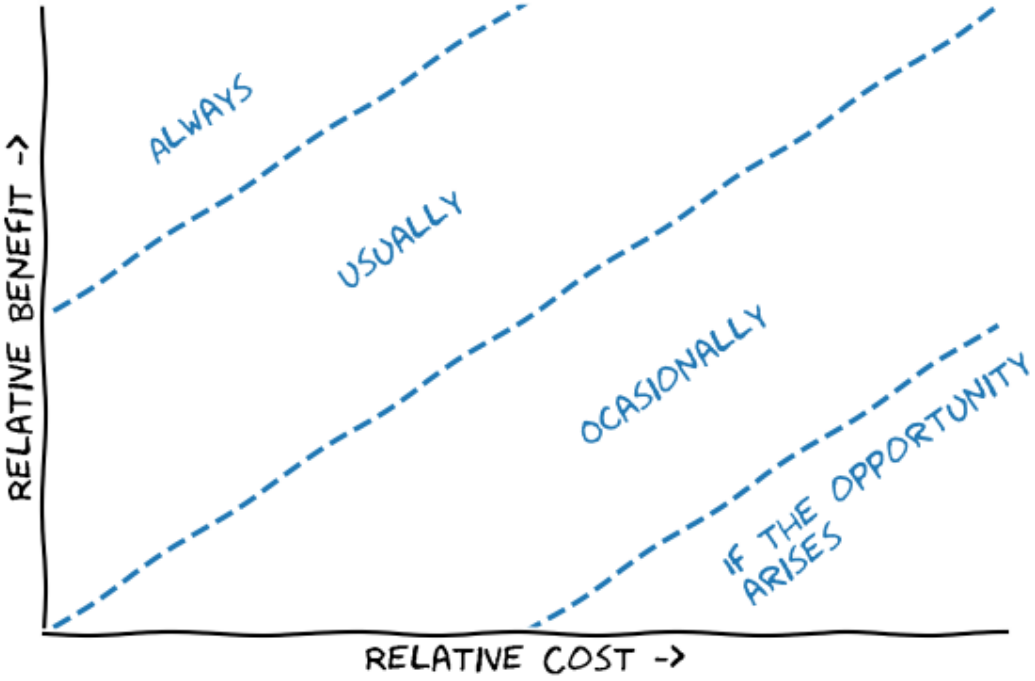

FIGURE 5-1 THE COST-BENEFIT MATRIX

This matrix allows for a systematic approach to a rough ranking of tests, enabling you to see at a glance which tests present the most value, and which can safely be deprioritised. It also helps keep track of how things evolve, for example if new information means that a specific test becomes less beneficial than another, but the cost doesn't change, this can be easily reflected in the matrix with any change to its relative value being inferred from there.

From personal experience, this sort of prioritisation matrix works best as a living document, or ideally a bunch of post-its stuck to a whiteboard. As you carry out testing, you can adjust your initial cost and benefit estimates to better reflect your reality.

5.1.3 Factors contributing to Cost

Here we list some of the more common factors contributing to the cost of tests. Note this is by no means comprehensive, and that not all considerations apply in all cases.

Financial Cost of Hardware

Compute cycles cost money, but this can usually be negligible in the early stages of development. Once you're into testing at scale, however, and are potentially spinning up cloud computing or clusters, this becomes a significant consideration.

Financial Cost of People

Depending on your circumstances, you may be contracting additional people to do QA testing, vulnerability scanning, etc., all of whom are eventually going to want paying in order to support their crippling addictions to food, water, and housing.

Opportunity Cost

Time and resources being finite,[18] the choice to do something is, by its nature, the choice to *not* do something else. As such, the cost of any choice includes the benefit you don't get by doing this over the other option or options.

Hassle

Sometimes a test is simply annoying or fiddly to do.

5.1.4 Factors contributing to Benefit

Listed below are some of the more common factors contributing to the benefit of a given test.

Information

How informative is the test? A test that simply tells you whether an error has occurred at some point in software provides very little information, whereas a test that tells you that a specific function of a specific module returned a known result to an accuracy of three significant figures when you require four is going to save you a lot of time in debugging.

Fail-Fast

How many steps does the test have to go through before telling you that it has failed, and how long do those steps take? A test that can return a negative result quickly, thereby saving you time and potentially removing the need to do more expensive tests on code that you now realise you need to rewrite anyway, is a pretty useful thing to keep around.

Damage Prevention

If a system that would fail this test were released, how bad would the fallout be? Or, put another way, how much is the peace of mind that this test brings worth to you?

Longevity

All code has a lifespan. Eventually, further development, bug fixes, or simply growing obsolete and being replaced will alter every single line of a program[19]. As such, any test specific to that code shares its lifespan. In a decision between a test that is likely to stick around for several years, and one that will need replacing in a week, the former offers greater benefit.

---

[18] [CITATION NEEDED]

[19] We will leave the philosophical discussion regarding the Ship of Theseus for another time.

Likelihood of Failure

There are always a few tests that will never seem to go right. Perhaps the behaviour they test for is highly sensitive to initial conditions, or it requires an interaction with a third-party API that seems particularly capricious, or it just seems to be entirely populated by gremlins. While it can be tempting to avoid these tests and the headaches they bring, these are actually some of the most valuable tests in your arsenal – the tests that are failed the most often are usually the most sensitive.

5.2 Milestones

As mentioned above, the cost-benefit matrix, and indeed any testing scheme, works best as a living document. Unforeseen circumstances are always going to arise, and your strategy must adapt to cope with them. However, you don't have infinite time to spend constantly reviewing and updating your testing strategy. As such, it can be helpful to set milestones in a project.

Each milestone reached is an opportunity to reassess the progress and circumstances of the project and, if necessary, adjust your testing strategy. It is also a good place to run any tests floating around in the "occasionally" region, and even consider those in the "If the opportunity arises" camp.

Milestones work better for assessing the testing scheme than fixed-time intervals because the main factor driving changes to testing is changes to the codebase. However, it is useful as a fallback to set a maximum time between reassessments.

6 Who Test?

While the glib answer to this question would be “You! And also, Everyone!”, there is a little more nuance that we should consider. Specifically, different levels of test are best performed by different people in different contexts.

The two extremes are Unit Testing and Acceptance Testing. The former is almost solely the domain of developers and maintainers – a test that tells you that a specific method of a specific class of a specific module throws an exception if called on a Saturday when the current clock time is an odd number is not going to mean much to the end-user, but is probably quite helpful to the developer. At the other end of the scale, while developers can (and should) test the code in as close to the deployment environment as possible and compare it very closely to the user’s specifications, whether or not it actually works correctly in a real environment and actually fits what the user wanted is a question best answered by having the end-user actually try to use the software.

In general, the best person to be performing any given test correlates with its place in the testing hierarchy, although we’re using ‘person’ in a very loose sense here. While we use the terms “developer”, “tester”, and “user” to refer to individuals with different interests and domains of responsibility, these may not be disparate individuals but rather ‘hats’ that a person can wear in different contexts.

Unit Testing – Developers

The limited scope allows all of the unit tests for a given chunk of code to be carried out in isolation, and the high degree of granularity is usually only relevant to people who are actively engaged with the nuts and bolts of the code.

Integration testing - testers and developers

It is very seldom that a developer is responsible for a single unit within a system, and much of the reasoning behind unit testing being carried out by developers also applies – units are often developed in parallel, and their interaction is a crucial part of their behaviour, and testing specific interactions between units has only slightly less granularity than unit testing.

At the same time, a broader approach to integration testing is often needed. Complex networks of modules developed by different teams don’t fall neatly under the purview of either team, and are better handled by those with a responsibility for more general testing.

In general, the two sides are best served by two different approaches. ‘Bottom-up’ integration testing, starting with the most simple combinations of units and building in complexity from there, is the most valuable to developers as it intuitively extends from unit testing and maps closely to the process of actually building the code. The mirroring ‘top-down’ approach, beginning with the most complex interactions and progressively getting more and more detailed, is often the best use of testers’ time. This allows unexpected emergent behaviour to be spotted early, and maximises the benefit of having a tester role with a broader oversight than individual developers or developer teams.

System Testing – Testers

As with top-down integration testing, system testing is best handled by testers. The task of setting up and running numerous testing environments, scaling the workload up and down, and checking against functional requirements does not overlap comfortably with development tasks, and needs a broad oversight of the entire project to perform effectively.

Acceptance Testing – Testers and Users

The scope of acceptance testing is similar to that of system testing, so the practical considerations that position testers as the ideal practitioners still apply. However, regardless of how detailed your understanding of your end-user requirements might be, the only way to be *certain* that the software meets the user's needs is to get a user to try it out. As a result it is valuable to have either members of the actual end-user pool, or individuals who are representative of their general level of expertise and needs, involved in acceptance testing.

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook Activity.

Appendix I. Glossary of Testing Terminology

I.i Types of Testing

Please note that as with many areas of software development, testing is an evolving field, and the definitions below therefore have a somewhat limited half-life. It is also worth noting that these terms are not axiomatic, but rather descriptions of related, overlapping categories. Many are concepts that have been inherited from the broader engineering discipline before having their meanings adapted in order to fit their application to software, while others are ideas unique to software engineering, but whose meanings have naturally shifted over time. These processes are stochastic and local, meaning that it is not uncommon to find conflicting definitions of these terms. In other words, even though we have endeavoured to be as general as possible, it is likely that you will find at least one of these definitions to be unsatisfactory. Sorry.

A/B Testing

Testing Level: Any
Testing Intent: Performance
Run a controlled experiment to determine if a proposed change is preferable to the current approach – literally choose between option A and option B.

Accessibility Testing

Testing Level: Acceptance
Testing Intent: Usability
A form of Non-Functional acceptance testing that confirms that the software meets accessibility requirements, such as the Web Accessibility Initiative.[20]

Alpha Testing

Testing Level: Acceptance
Testing Intent: Analytical Result, Performance, or Usability
Alpha testing is performed by developers or internal to identify potential issues and bugs that would otherwise be faced by the end users. As part of Acceptance Testing it is done towards the end of development. It is usually followed by Beta testing, which is similar in scope but performed by real users.

Assert Test

Testing Level: Unit
Testing Intent: Analytical Result
Assert Tests are statements that assert that some property of the software should have a certain quality. This can be as simple as “variable x exists” and indicate an abstract conceptual understanding about how the code should behave or operate, against which the concrete reality of the code can be compared.

[20] https://www.w3.org/WAI/

Beta Testing

Testing Level: Acceptance
Testing Intent: Analytical Result, Performance, or Usability
Beta testing is performed by a selection of real users to identify issues or bugs that escaped detection in the preceding Alpha Testing.

Compatibility Testing

Testing Level: Acceptance
Testing Intent: Analytical Result or Performance
Compatibility Testing checks that the software works correctly when used on different hardware, operating systems, software, or network environments.

Concurrent Testing

Testing Level: Any
Testing Intent: Performance
Concurrent Testing is a subset of Performance Testing that examines the behaviour of the code in concurrent computing environments.

Conformance Testing

Testing Level: Any
Testing Intent: Acceptance Testing
Conformance Testing (also called Compliance Testing) examines the software in the context of determining its compliance with a set of standards and regulations, for example those defined by IEEE, W3C or ETSI.

Destructive Testing

Testing Level: Unit, Integration, or System
Testing Intent: Analytical Result or Performance
Destructive Testing is any testing approach wherein the software is intentionally made to fail in order to determine where and how that failure takes place, and what fallout may result from it.

Functional Testing

Testing Level: Integration or System
Testing Intent: Analytical Result
Functional testing is a form of black-box testing where a software component or system is tested against its functional specifications – in other words it asks whether the code does what it is supposed to do, while being agnostic to *how* it does so.

Installation Testing

Testing Level: Acceptance Testing
Testing Intent: Usability
As the name suggests, installation testing involves running through the installation procedure for your software on a variety of platforms and environments to verify that it installs correctly.

Internationalization and Localisation

Testing Level: Acceptance Testing
Testing Intent: Usability
If software needs to be used outside of the culture and language in which it was created, then the accuracy of any translations, and the ease of switching languages, needs to be tested.

Non-Functional Testing

Testing Level: Acceptance
Testing Intent:
A form of Acceptance Testing**Error! Reference source not found.** that refers to any test that confirms concordance with a non-function requirement. For example: Accessibility Testing.

Operational Testing

Testing Level: Acceptance
Testing Intent: Usability
A form of Acceptance Testing that confirms that the day-to-day operations of the software such as its general performance, backup, recovery, and maintenance procedures, meet requirements. It is ideally performed by the teams, or team members, who are responsible for supporting, maintaining, or operating the software.

Sanity Testing

Testing Level: Any
Testing Intent: Build
Sanity testing can be considered the 0^{th} order of testing – it encompasses any test that is performed to determine whether carrying out further tests is viable.

Security Testing

Testing Level: System
Testing Intent: Performance
Uncover security vulnerabilities, often using known attack patterns and threats.

Smoke Testing

Testing Level: Unit
Testing Intent: Build
Confirms that a build is successful. Commonly used as the most basic form of Sanity Testing since if code fails a smoke test there it is unlikely that more complex tests are even possible.

The name originates from a product development practice best summed up as “turn it on and see whether or not it catches fire.”