Unit Testing

Table of Contents

Introduction

This document focuses on **automated unit tests**.

We use the term "**unit testing**" to refer to the process of software developers creating and using automated unit tests to check that small units of code works as they expect.

In particular:

- Unit tests are automated
- Unit tests check that a small part of the software system works correctly
- Unit tests are small (typically less than 20 lines of code)
- Unit tests run quickly (milliseconds per test)
- Unit tests are typically written by developers
- Unit tests use test dummies to simulate any external dependencies (e.g., database)
- Unit tests are committed to the code base along with the code changes they are testing

Writing unit tests **at the same time** as writing the code helps inform the design of the code and ensure that the code is easily testable

Organizations like Google do not allow developers to submit code without unit tests.

These tests are typically organized into **test suites** of related tests. Unit tests help a developer appropriately evolve the design of their code. While developing their code on their workstation, developers run unit tests and test suites to validate that their code works and has not broken other unit tests. Similarly, automated continuous integration / continuous deployment (CI/CD) pipelines run test suites as one necessary (but not sufficient) step to validate that committed code has not broken the system.

As described below, such automated test suites provide a great deal of value to individual developers and to the organizations that they work for. This is one reason unit testing has become widely adopted across software development organizations.

Learning Objectives

- Understand what unit testing is, and how unit testing is similar and different from other types of testing
- Understand the history of unit testing, and how it fits into the overall software development life cycle
- Understand how the Formality Continuum helps show how unit testing, manual testing, exploratory testing, and rapid software testing relate to each other
- Understand the Test-Driven Development (TDD) process for creating and using unit tests
- Understand how refactoring relates to unit testing and changing a system's functionality
- Understand the ethics of testing one's own code

Preview

This document starts with some long excerpts describing why and how Google dramatically changed its testing practices in the 2000s. This framing will help you understand some of the larger organizational and contextual issues that relate to testing.

Next we frame the nature of software testing, including a couple of models that describe how different types of tests and testing relate to each other. The Testing Pyramid illustrates provides a way to think about automating tests, and how unit tests form the foundation of effective software testing helping to ensure that the code does not have internal bugs.

At this point, we dive into some of the history of software testing, which goes back to the early 1960s where Jerry Weinberg helped develop and test the code for the Mercury Space Program. Jerry went on to be a huge luminary in the world of software development writing a large number of amazingly good books covering many aspects of software development.

We then discuss the Formality Continuum and how that helps relate unit testing and manual testing. Manual testing is fundamentally different from the work of creating automated unit tests. To illustrate this, we provide a brief introduction to the distinction between “testing” and “checking”, and then briefly describe two well-developed frameworks for doing manual testing: Exploratory Testing, and Rapid Software Testing (which is a successor to Exploratory Testing).

With this background, we introduce the Agile Testing Quadrant, which illustrates some of the key ways in which unit testing is similar and different from other types of testing.

Next comes more concrete guidance on how to do unit testing, how it fits into Test-Driven Development (TDD), and a bit of discussion about whether “TDD is dead”. Finally, we touch on some of the ethics of unit testing, and provide links to additional resources.

Why Do Unit Testing?

Software Testing at Google

The text in the following boxes is from the 2012 book *How Google Tests Software: Help me test like Google* by James Whittaker, Jason Arbon, and Jeff Carollo. The book is a great read and discusses much more about testing at Google cerca 2012.

Some of the details may have changed since then, but many of the aspects are invariant, such as:

- the arc of how testing becomes more important as an organization grows in size,
- the centrality of test and reviews,
- the amount of infrastructure created to ensure high-levels of compliance to writing tests and doing reviews.

As you read this description, consider that early in its development **Google decided to store its entire codebase in a single code repository**. As noted in Why Google Stores Billions of Lines of Code in a Single Repository, by 2016 the Google codebase had grown to include approximately 1 billion files storing both 2 billion lines of code spread across 9 million source files, and the history of 35 million commits. (Source code repositories contain not just the source files, but files that define each commit.) The 25,000 Google software developers were committing 16,000 changes per day to this single shared repository. That is why Google needed to create the type of processes described below.

Roles

In order for the "you build it, you break it" motto to be real (and kept real over time), there are roles beyond the traditional feature developer that are necessary. Specifically, engineering roles that enable developers to do testing efficiently and effectively have to exist. At Google, we have created roles in which some engineers are responsible for making other engineers more productive and more quality-minded. These engineers often identify themselves as *testers,* but their actual mission is one of productivity. Testers are there to make developers more productive and a large part of that productivity is avoiding re-work because of sloppy development. Quality is thus a large part of that productivity. We are going to spend significant time talking about each of these roles in detail in subsequent chapters; therefore, a summary suffices for now.

The **software engineer (**SWE**)** is the traditional developer role. SWEs write functional code that ships to users. They create design documentation, choose data structures and overall architecture, and they spend the vast majority of their time writing and reviewing code. SWEs write a lot of test code, including test-driven design (TDD), unit tests, and, as we explain later in this chapter, participate in the construction of small, medium, and large tests. SWEs own quality for everything they touch whether they wrote it, fixed it, or modified it. That's right, if a SWE has to modify a function and that modification breaks an existing test or requires a new one, they must author that test. SWEs spend close to 100 percent of their time writing code.

The **software engineer in test (**SET**)** is also a developer role, except his focus is on testability and general test infrastructure. SETs review designs and look closely at code quality and risk. They refactor code to make it more testable and write unit testing frameworks and automation. They are a partner in the SWE codebase, but are more concerned with increasing quality and test coverage than adding new features or increasing performance. SETs also spend close to 100 percent of their time writing code, but they do so in service of quality rather than coding features a customer might use.

> SETs are partners in the SWE codebase, but are more concerned with increasing quality and test coverage than adding new features or increasing performance. SETs write code that allows SWEs to test their features.

The **test engineer** (TE) is related to the SET role, but it has a different focus. It is a role that puts testing on behalf of the user first and developers second. Some Google TEs spend a good deal of their time writing code in the form of automation scripts and code that drives usage scenarios and even mimics the user. They also organize the testing work of SWEs and SETs, interpret test results, and drive test execution, particularly in the late stages of a project as the push toward release intensifies. TEs are product experts, quality advisers, and analyzers of risk. Many of them write a lot of code; many of them write only a little.

> **Meta from instructors:** As an aside, around 2013 many large cloud-based software development organizations like Microsoft, Google, and Amazon started to eliminate (or nearly eliminate) software testing as a separate role. Instead, testing is the responsibility of the developer. This likely was largely motivated by the adoption of continuous integration / continuous deployment (CI/CD) and DevOps which allowed live software systems to be upgraded easily, quickly, safely, and frequently. Some runnings systems might be updated multiple times per day or hour.
>
> This is not necessarily good. The lived experience often shows that the quality of the testing often declines, because software developers do not understand how to think and act like a tester. If you know how to BOTH be a good developer AND be a good tester, you have a high chance of being quite successful. Brett Pettichord wrote a lovely article about how Testers and Developers Think Differently.
>
> Software systems that cannot be quickly and safely updated as needed (e.g., an airplane's flight controller) often have a higher need for dedicated software testers, or for software developers who embrace the mindsets and practices of excellent software testers.

and

> Google takes code reviews seriously, and, especially with common code, developers must have all their code reviewed by someone with a "readability" in the relevant programming language. A committee grants readabilities after a developer establishes a good track record for writing clean code which adheres to style guidelines. Readabilities exist for C++, Java, Python, and JavaScript: Google's four primary languages.

and

Testability

SWEs and SETs work closely on product development. SWEs write the production code and tests for that code. SETs support this work by writing test frameworks that enable SWEs to write tests. SETs also take a share of maintenance work. Quality is a shared responsibility between these roles.

A SET's first job is testability. They act as consultants recommending program structure and coding style that lends itself to better unit testing and in writing frameworks to enable developers to test for themselves. We discuss frameworks later; first, let's talk about the coding process at Google.

To make SETs true partners in the ownership of the source code, Google centers its development process around code reviews. There is far more fanfare about reviewing code than there is about writing it.

Reviewing code is a fundamental aspect of being a developer and it is an activity that has full tool support and an encompassing culture around it that has borrowed somewhat from the open source community's concept of "committer" where only people who have proven to be reliable developers can actually commit code to the source tree.

> Google centers its development process around code reviews. There is far more fanfare about reviewing code than there is about writing it.

At Google everyone is a committer, but we use a concept called *readability* to distinguish between proven committers and new developers. Here's how the whole process works:

Code is written and packaged as a unit that we call a change list, or CL for short. A CL is written and then submitted for review in a tool known internally as Mondrian (named after the Dutch painter whose work inspired abstract art). Mondrian then routes the code to qualified SWEs or SETs for review and eventual signoff.[8]

CLs can be blocks of new code, changes to existing code, bug fixes, and so on. They range from a few lines to a few hundred lines with larger CLs almost always broken into smaller CLs at the behest of reviewers. SWEs and SETs who are new to Google will eventually be awarded with a readability designation by their peers for writing consistently good CLs. Readabilities are language specific and given for C++, Java, Python, and JavaScript, the primary languages Google uses. They are credentials that designate an experienced, trustworthy developer and help ensure the entire codebase has the look and feel of having been written by a single developer.[9]

There are a number of automated checks that occur before a CL is routed to a reviewer. These *pre-submit rules* cover simple things such as adherence to the Google coding style guide and more involved things such as ensuring that every existing test associated with the

CL has been executed (the rule is that all tests must pass). The tests for a CL are almost always included in the CL itself—test code lives side by side with the functional code. After these checks are made, Mondrian notifies the reviewer via email with links to the appropriate CLs. The reviewer then completes the review and makes recommendations that are then handled by the SWE. The process is repeated until both the reviewers are happy and the pre-submit automation runs clean.

A submit queue's primary goal in life is to keep the build "green," meaning all tests pass. It is the last line of defense between a project's continuous build system and the version control system. By building code and running tests in a clean environment, the submit queue catches environmental failures that might not be caught by a developer running tests on an individual workstation, but that might ultimately break the continuous build or, worse, make its way into the version control system in a broken state.

A submit queue also enables members of large teams to collaborate on the main branch of the source tree. This further eliminates the need to have scheduled code freezes while branch integrations and test passes take place. In this way, a submit queue enables developers on large teams to work as efficiently and independently as developers on small teams. The only real downside is that it makes the SET's job harder because it increases the rate at which developers can write and submit code!

---

[8] An open-source version of Mondrian hosted on App Engine is publicly available at http://code.google.com/p/rietveld/.
[9] Google's C++ Style Guide is publicly available at http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.

and

How Submit Queues and Continuous Builds Came Into Being

by Jeff Carollo

In the beginning, Google was small. Having a policy of writing and running unit tests before checking in changes seemed good enough. Every now and then, a test would break and people would spend time figuring out why and then fix the problem.

The company grew. To realize the economies of scale, high-quality libraries and infrastructure were written, maintained, and shared by all developers. These core libraries grew in number, size, and complexity over time. Unit tests were not enough; integration testing was now required for code that had significant interactions with other libraries and infrastructure. At some point, Google realized that many test failures were introduced by dependencies on

other components. As tests were not being run until someone on a project wanted to check in a change to that project, these integration failures could exist for days before being noticed.

Along came the “Unit Test Dashboard.” This system treated every top-level directory in the companywide source tree as a “project.” This system also allowed anyone to define their own “project,” which associated a set of build and test targets with a set of maintainers. The system would run all tests for each project every day. The pass and fail rate for each test and project was recorded and reported through the dashboard. Failing tests would generate emails to the maintainers of a project every day, so tests did not stay broken for long. Still, things broke.

Teams wanted to be even more proactive about catching the breaking changes. Running every test every 24 hours was not enough. Individual teams began to write Continuous Build scripts, which would run on dedicated machines and continuously build and run the unit and integration tests of their team. Realizing that such a system could be made generic enough to support any team, Chris Lopez and Jay Corbett sat down together and wrote the “Chris/Jay Continuous Build,” which allowed any project to deploy its own Continuous Build system by simply registering a machine, filling in a configuration file, and running a script. This practice became popular quite rapidly, and soon most projects at Google had a Chris/Jay Continuous Build. Failing tests would generate emails to the person or persons most likely to be the cause of those tests failing within minutes of those changes being checked in! Additionally, the Chris/Jay Continuous Build would identify “Golden Change Lists,” checkpoints in the version control system at which all tests for a particular project built and passed. This enabled developers to sync their view of the source tree to a version not affected by recent check-ins and build breakages (quite useful for selecting a stable build for release purposes).

Teams still wanted to be more proactive about catching breaking changes. As the size and complexity of teams and projects grew, so too did the cost of having a broken build. Submit Queues were built out of necessity as protectors of Continuous Build systems. Early implementations required all CLs to wait in line to be tested and approved by the system serially before those changes could be submitted (hence, the “queue” suffix). When lots of long-running tests were necessary, a several-hour backlog between a CL being sent to the queue and that CL actually being submitted into version control was common. Subsequent implementations allowed all pending CLs to run in parallel with one another, but isolated from each other’s changes. While this “improvement” did introduce race conditions, those races were rare, and they would be caught by the Continuous Build eventually. The time saved in being able to submit within minutes of entering the submit queue greatly outweighed the cost of having to resolve the occasional Continuous Build failure. Most large projects at Google adopted the use of Submit Queues. Most of these large projects also rotated team members into a position of “build cop,” whose job it was to respond quickly to any issues uncovered in the project’s Submit Queue and Continuous Build.

This set of systems (the Unit Test Dashboard, the Chris/Jay Continuous Build, and Submit Queues) had a long life at Google (several years). They offered tremendous benefit to teams in exchange for a small amount of set-up time and varying amounts of ongoing maintenance. At some point, it became both feasible and practical to implement all of these systems in an integrated way as shared infrastructure for all teams. The Test Automation Program, or TAP, did just that. As of this writing, TAP has taken the place of each of these systems and is in use by nearly all projects at Google outside of Chromium and Android (which are open source and utilize separate source trees and build environments from the server side of Google).

The benefits of having most teams on the same set of tools and infrastructure cannot be overstated. With a single, simple command, an engineer is able to build and run all binaries and tests that might be affected by a CL in parallel in the cloud, have code coverage collected, have all results stored and analyzed in the cloud, and have those results visualized at a new permanent web location. The output of that command to the terminal is “PASS” or “FAIL” along with a hyperlink to the details. If a developer chooses to run his tests this way, the results of that test pass, including the code coverage information, are stored in the cloud and made visible to code reviewers through the internal Google code review tool.

What is software testing?

Cem Kaner, a prolific software testing thought leader (and has framed software testing as a social science), defines software testing as “an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.”

James Bach, another prolific software testing thought leader, defines software testings as “epistemology”. For more from James, see the vast amount of excellent material at satisfice.com.

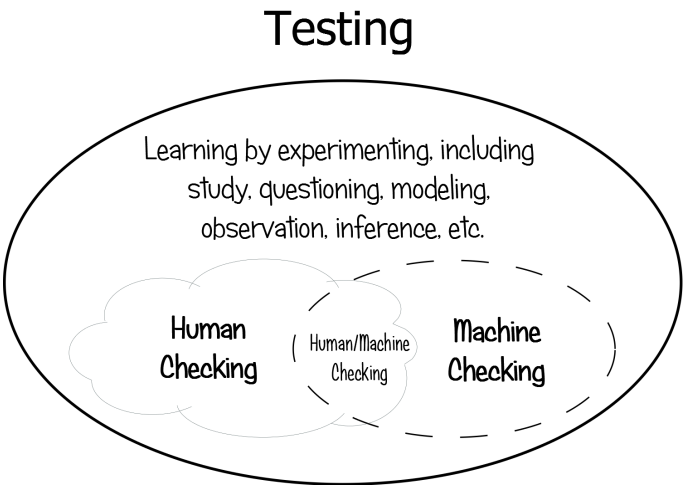
Testing ≠ Checking

James Bach and Michael Bolton’s article on Testing and Checking Redefined draws an important distinction between **testing** (which requires a human in the loop making decisions) and **checking** (which can be done by either a human or a machine):

> **Testing** *is the process of evaluating a product by learning about it through experiencing, exploring, and experimenting, which includes to some degree: questioning, study, modeling, observation, inference, etc.*
>
> (A **test** *is an instance of testing.*)

Checking *is the process of making evaluations by applying algorithmic decision rules to specific observations of a product.*

(A **check** *is an instance of checking.*)



(James Bach and Michael Bolton)

- **Testing** is a very creative process of finding an issue. It involves a **person** using their full cognitive abilities to creatively find and document issues. There is an extensive literature around doing effective testing (see
- **Checking** is following a scripted sequence of steps to see if an issue exists. The "automated tests" that people talk about really are "automated checks".
- Checking can be automated. Testing cannot be automated, though people are working on machine learning algorithms to do that.

Given this distinction, an "automated test" would be better called an "automated check". However, the use of "test" is too embedded in the culture to change.

What is "unit testing"?

There are many, many "types" of testing. For instance, the Guru99 site has a list of 100 types of software testing:

- acceptance
- accessibility
- active
- agile
- age
- ad-hoc
- alpha
- assertion
- API
- all-pairs
- automated
- basic path
- backward compatibility
- beta
- benchmark
- big bang integration
- binary portability
- boundary value
- bottom up
- branch
- breadth
- black box
- code-driven
- compatibility
- comparison
- component
- configuration
- condition coverage
- compliance
- concurrency
- conformance
- context driven
- conversion
- decision coverage
- destructive
- dependency
- endemic
- domain
- error-handling
- end-to-end
- endurance
- exploratory
- equivalence partitioning
- fault injection
- formal verification
- functional
- fuzz
- guerrilla
- gray box
- glass box
- GUI software
- globalization
- hybrid integration
- integration
- interface
- install/uninstall
- internationalization
- inter-systems
- keyword-driven
- load
- localization
- loop
- manual scripted
- manual-supported
- model-based
- mutation
- modularity-driven
- non-functional
- negative
- operational
- orthogonal array
- pair
- passive
- parallel
- past
- attrition
- performance
- qualification
- ramp
- regression
- recovery
- requirements
- security
- sanity
- scenario
- scalability
- statement
- static
- stability
- smoke
- storage
- stress
- structural
- system
- system integration
- top-down integration
- unit
- user interface
- usability
- volume
- vulnerability
- white box
- workflow

Unit testing is just one of those types.

This document focuses on the process of creating **automated unit tests**.

Martin Fowler, an author of much wisdom about software development, notes in his entry on UnitTest that the term “unit test” is very ill-defined. In general, “unit tests are low-level, focusing on a small part of the software system” but people disagree about what is “low-level” or “small”. In other words, “unit test” is a relative term. A unit tends to be smaller than a component, which tends to be smaller than a module, which tends to be smaller than a subsystem, which tends to be smaller than a system, and so on.

Guru 99 defines unit testing as a “software verification and validation method in which a programmer tests if individual units of source code are fit for use. It is usually conducted by the development team.”

Before describing how to do unit testing, we cover a set of models that help illustrate how unit testing is similar and different from other types of software testing.

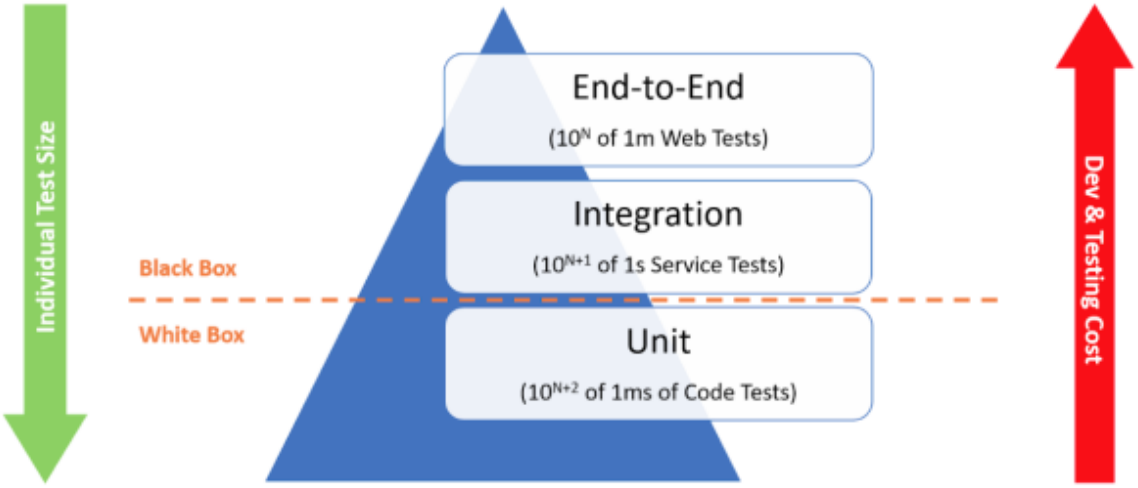
The Testing Pyramid

The Testing Pyramid is a layered model about how three categories of automated tests (unit, integration, end-to-end) are similar and different, and how together they help ensure that a software functions as desired.

Here are some terms to know for the following excerpt:

- "**White box**" tests are written with knowledge of the product code. The developer can read the code, and know how it operates.
- "**Black box**" tests are written without any knowledge of how the code works internally. These might be written by a tester or product manager, or by a developer for a code sub-system that they do not have the code for.

Excerpt from Automation Panda.



The "Testing Pyramid" is an industry-standard guideline for functional test case development. Love it or hate it, the Pyramid has endured since the mid-2000's because it continues to be practical. So, what is it, and how can it help us write better tests?

Layers

The Testing Pyramid has three classic layers:

- ***Unit tests* are at the bottom.** Unit tests directly interact with product code, meaning they are "white box." Typically, they exercise functions, methods, and classes. Unit tests should be short, sweet, and focused on one thing/variation. They should *not* have any external dependencies – mocks/monkey-patching should be used instead.
- ***Integration tests* are in the middle.** Integration tests cover the point where two different things meet. They should be "black box" in that they interact with live instances of the product under test, not

code. Service call tests (REST, SOAP, etc.) are examples of integration tests.

- ***End-to-end tests* are at the top.** End-to-end tests cover a path through a system. They could arguably be defined as a multi-step integration test, and they should also be "black box." Typically, they interact with the product like a real user. Web UI tests are examples of integration tests because they need the full stack beneath them.

All layers are *functional tests* because they verify that the product works correctly.

Proportions

The Testing Pyramid is triangular for a reason: **there should be more tests at the bottom and fewer tests at the top**. Why?

1. *Distance from code*. Ideally, tests should **catch bugs as close to the root cause as possible**. Unit tests are the first line of defense. Simple issues like formatting errors, calculation blunders, and null pointers are easy to identify with unit tests but much harder to identify with integration and end-to-end tests.
2. *Execution time.* Unit tests are very quick, but end-to-end tests are very slow. Consider the **Rule of 1's** for Web apps: a unit test takes ~1 millisecond, a service test takes ~1 second, and a Web UI test takes ~1 minute. If test suites have hundreds to thousands of tests at the upper layers of the Testing Pyramid, then they could take hours to run. An hours-long turnaround time is unacceptable for continuous integration.
3. *Development cost.* Tests near the top of the Testing Pyramid are **more challenging to write** than ones near the bottom because they cover more stuff. They're longer. They need more tools and packages (like Selenium WebDriver). They have more dependencies.
4. *Reliability*. Black box tests are susceptible to **race conditions and environmental failures**, making them inherently more fragile. Recovery mechanisms take extra engineering.

The total cost of ownership increases when climbing the Testing Pyramid. When deciding the level at which to automate a test (and if to automate it at all), taking a risk-based strategy to push tests down the Pyramid is better than writing all tests at the top. **Each proportionate layer mitigates risk at its optimal return-on-investment**.

Practice

The Testing Pyramid should be a guideline, not a hard rule. **Don't require hard proportions for test counts at each layer.** Why not? Arbitrary metrics cause bad practices: a team might skip valuable end-to-end tests or write needless unit tests just to hit numbers. W. Edwards Deming would shudder!

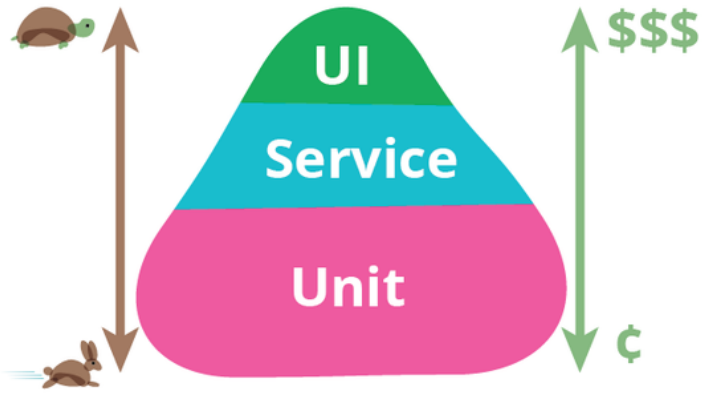
Instead, **use loose proportions to foster better retrospectives**. Are we covering too many input combos through the Web UI when they could be checked via service tests? Are there unit test coverage gaps? Do we have a pyramid, a diamond, a funnel, a cupcake, or some other wonky shape? Each layer's test count should be roughly an order of magnitude smaller than the layer beneath it. Large Web apps often have 10K unit tests, 1K service tests, and a few hundred Web UI tests.

Resources

Check out these other great articles on the Testing Pyramid:

- TestPyramid by Martin Fowler
- The Practical Test Pyramid by Ham Vocke
- Test Pyramid: the key to good automated test strategy by Tim Cochran
- Just Say No to More End-to-End Tests by Mike Wacker

Martin Fowler presents a slightly different framing for the testing pyramid:



(Martin Fowler, TestPyramid)

Martin Fowler is an excellent and influential writer about pragmatic and effective software engineering principles, processes, and practices. I highly recommend you spend time exploring and reading his website: MartinFowler.com.

In summary…

Compared with UI / end-to-end tests, unit tests:

- Test smaller pieces of code
- Are created by developers (instead of product owners, business analysts, testers)
- Cost less to create
- Cost less to maintain, because they are less brittle maintain
- Run faster
- Are more frequent in number
- Have a higher return on investment (ROI)
- Are about building the thing right (verification) instead of building the right thing (validation)
- Help inform the design of the software code, instead of informing the design of the user experience

Testing Methodologies - History and Framing

A bit of History

There has been a long running debate about how formal methodologies should be.

As described by James Bach and Michael Bolton in Exploratory Testing 3.0, software testing started in the 1960s as a very creative, problem-solving process by which people determined how well a system was or was not doing what it needed to do.

This included using test driven development, pair programming, and code and test reviews to ensure high quality and effectiveness. As described in this post on TDD on Punch Cards: Mercury Space Program (and elsewhere), all of those practices were used in the early 1960s to develop the code for the Mercury Space Program. As Jerry Weinberg, one of those programmers, wrote in 2018 shortly before he passed away:

> we tried to avoid many cycles of coding-testing-correcting. **We wrote out tests up front to the extent we could, had them punched into data cards, and when the code was ready, used them to run our tests**.
>
> Typically, our goal was to have no more than one cycle of corrections. This was made even more important when we were in Los Angeles, running our tests on a 704 in New York, with the punch cards being sent by (non-jet) air freight. A**t least 95% of the time,**

> **our programs ran the first time without error**, in large part **because of our pair programming practices and code and test reviews**.

Unfortunately, such practices almost completely disappeared from the mainstream as the formalizers tried to codify testing into a heavy-weight and plan-driven process (aka waterfall) using fixed forms with specific processes and artifacts, such as test scripts of the steps to check whether a part of a system worked, and test plans indicating what tests must be done when. Meanwhile, visionary practitioners pushed back, realizing that the core of effective testing was exploratory; this group pushed for "exploratory testing", which finally gained popularity in the 2000s.

Around the same time, in February 2001, s set of seventeen visionary practitioners and consultants came together at The Lodge at Snowbird ski resort in Utah to discuss the similarities and differences of the highly effective software processes and practices that they had independently created and tested at scale. Those processes and practices were effective in face of the particularities of software development. The result of that meeting was the Agile Manifesto.

By the 2010s, James Bach and Michael Bolton, who have been very influential in developing exploratory testing, realized that the term "exploratory testing" was limiting and harmful. Here's what they wrote in Exploratory Testing 3.0 (bolding added):

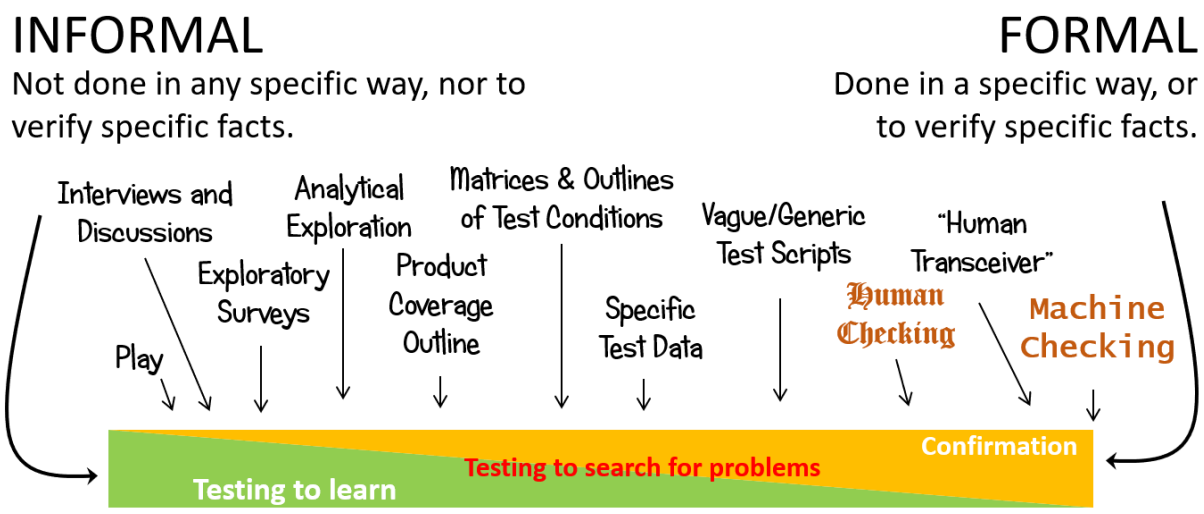
> ET 3.0 as a term is a bit paradoxical because what we are working toward, within the Rapid Software Testing methodology, is nothing less than **the deprecation of the term "exploratory testing."**
>
> Yes, we are retiring that term, after 22 years. Why?
>
> Because ***we now define all testing as exploratory***. Our definition of testing is now this:
>
> *"Testing is the process of evaluating a product by learning about it through exploration and experimentation, which includes: questioning, study, modeling, observation and inference, output checking, etc."*
>
> Where does scripted testing fit, then? By "script" we are speaking of any control system or factor that influences your testing and lies outside of your realm of choice (even temporarily). This does not refer only to specific instructions you are given and that you must follow. Your biases script you. Your ignorance scripts you. Your organization's culture scripts you. The choices you make and never revisit script you.

> By defining testing to be exploratory, scripting becomes a guest in the house of our craft; a potentially useful but foreign element to testing, one that is interesting to talk about and apply as a tactic in specific situations. An excellent tester should not be complacent or dismissive about scripting, any more than a lumberjack can be complacent or dismissive about heavy equipment. This stuff can help you or ruin you, but no serious professional can ignore it.
>
> Are you doing testing? *Then you are already doing exploratory testing*. Are you doing scripted testing? If you're doing it responsibly, you are doing *exploratory testing with scripting* (and perhaps with checking). If you're *only* doing "scripted testing," then you are just doing *unmotivated checking,* and we would say that you are not really testing. You are trying to behave like a machine, not a responsible tester.
>
> ET 3.0, in a sentence, is the demotion of scripting to a technique, and the promotion of exploratory testing to, simply, *testing*.

Of course, today we have many more tools available to help us test and check. And over the last several decades, people like Kem Caner, James Bach, Michael Bolton, and many others have substantially deepened our understanding of what testing is about.

The Formality Continuum

The formality continuum provides a useful tool for thinking about what types of methodology or practice is suitable in what contexts. The following diagram shows how testing methods vary from very informal on the left to very formal on the right. All of these methods are useful, but for different purposes and different contexts.



Loops of testing start with informal, exploratory work. If you want to do excellent formal testing (like automated checking), it must begin with excellent informal work.

(from https://www.developsense.com/blog/2014/07/sock-puppets-of-formal-testing/ which is linked to from Rapid Software Testing http://www.satisfice.com/rst.pdf)

What does the formality continuum show with respect to testing?

- The range of testing methods varies from very informal to very formal.
- Informal testing methods tend to be more about testing to learn about how a system works (the green triangle).
- Formal testing methods tend to be more about checking or confirming that a system works appropriately (the yellow triangle).
- Both formal and informal methods are used to search for problems.

Your task as a professional software engineer is to determine which method to use when. Any non-trivial project will require a broad mix of approaches, with the choices varying depending on the emerging needs. Be intentional and thoughtful. Avoid rigidity. Avoid fixating on just one place on the continuum.

Manual Testing

An important distinction that is not represented in the Testing Pyramid is manual testing, testing done by a **human**. While manual testing is more expensive than automated tests, **manual testing can be incredibly valuable … if done well**. There are lots of processes and methods to help one manual test effectively.

For several decades starting around the 1970s, there was a movement to formalize testing, to move testing toward the formal side of the formality continuum. Extensive test management tools were created to manage often large sets of bug reports, and manual testing scripts with very detailed steps to allow a person to manually check for an issue. Test scripts often was

linked to the features being tested. The test management system allowed an organization to triage, prioritize, track, and report on the number of open defects, test coverage, etc.

The test management system also allowed an organization to define a set of tests that must be run before major stages in the software development life cycle, such as before a release. Running all of those tests required lots of software testers. In the 1990s, some people recommended having 1 tester for every 2 developers.

Automating tests was a holy grail, but typically difficult to do. One reason was that the software was not designed with testability in mind.

This situation began to change more rapidly when the agile development community started to introduce and embrace more automated testing practices. Unit testing via xUnit test frameworks led to a rapid growth in the number of low-level automated unit tests. Behavior-driven testing allowed non-developers to define checks to run simply by filling in tables on a website with the return value to expect if function X was called with data Y. Automated build processes provide a platform for running tests daily and then on-demand whenever new code was integrated into the repository.

As TDD and other development practices were adopted, there was less need for humans whose jobs were devoted completely to testing. The number of “software testers” began to shrink rapidly in many organization, yet the overall quality of the software and ability to scale to huge numbers of users and amount of data continued to grow.

Excellent software engineers tend to:

- Automate repeated work that can be effectively and whose automation results in good return on investment (ROI). A rough guideline is to consider automating checks that are non-trivial to do manually and that need to be run more than a handful of times.
- Create and evolve effective principles, processes, and practices for doing the remaining work.

In some ways, the job of an excellent software engineer is to automate the repetitive tasks so that one can focus on even more interesting tasks.

Exploratory Testing

Exploratory testing is a set of principles, processes, and practices about using our amazing human abilities to do highly effective testing of software systems.

While the term “exploratory testing” was first coined in the 1980’s by Cem Kaner, the concepts were largely ignored until around 2010, when the practitioners began to publish several books on exploratory testing, including James Whittaker’s 2009 book *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*:, and Elisabeth Hendrickson’s 2013 book *Explore It! Reduce Risk and Increase Confidence with Exploratory Testing*.

James Bach, a highly influential thought leader around software testing, provides a very thoughtful overview of exploratory testing that covers:

- The exploratory nature of software testing
- Testing is not A science; testing IS science
- Formality vs. Informality
- Agency vs. Algorithm
- Deliberation vs. Spontaneity
- Structured vs. Unstructured
- Tacit vs. Explicit
- Exploration is unpredictable, yet responsible and controllable
- Testing is Not Proving; Verification is Not Testing

Here's how Bach's overview starts:

> *I didn't coin the term exploratory testing. Cem Kaner did that in the 80's . He was inspired by "exploratory data analysis," which is a term coined by John Tukey. In the late 80's and early 90's I was noticing that testers who did not follow any explicit script were better at finding bugs than those who did. I decided to find out why. That led to my first conference talk called* The Persistence of Ad Hoc Testing*, in 1993. Soon afterward I encountered Cem's terminology and began saying exploratory testing. At the time, Cem had left the industry to become a lawyer, so I was the literally the only testing guy speaking at conferences about exploratory testing. (If you were speaking or writing publicly about ET at that time, even by another name, please contact me so I can correct this claim.)*
>
> *In early 1995 I met Cem, who was coming back to testing. We joined forces. In the years that followed, we successfully popularized the idea of exploratory testing. In 1996, I designed the first class explicitly devoted to ET. This became the core of the Rapid Software Testing methodology. I also created the definition of exploratory testing that the ISTQB partially plagiarized and partially vandalized in their syllabus. In other words, this is my area of special expertise. Those of us really passionate about studying this have evolved our understanding over time. I will tell you where* my own education and analysis has taken me, over the last 30 years or so.

Rapid Software Testing

Rapid Software Testing is a successor to exploratory testing, from which it evolved. As James Bach explains in Rapid Software Testing Methodology:

> *I call what I do* Rapid Software Testing.
>
> *Why do we test? We test to develop a comprehensive understanding of the product and the risks around it. We test to find problems that threaten the value of the product, or that threaten the on-time, successful completion of any kind of development work. We test to*

help the business, managers, and developers decide whether the product they've got is the product they want.

Above all, we test because it is the responsible thing to do. We have a duty of care toward our teams, our organizations, our customers, and society itself. Releasing poorly tested software would be a breach of that duty.

Rapid Software Testing (RST) is all about that. It is a responsible approach to software testing, centered around people who do testing and people who need it done. It is a methodology (in the sense of "a system of methods") that embraces tools (aka "automation") but emphasizes the role of skilled technical personnel who guide and drive the process.

The essence of this methodology lies in its ontology *(how we organize and define the various priorities, ideas, activities, and other elements of testing),* humanism *(we foster responsibility and resilience by putting the methodology under the control of each practitioner), and* heuristics *(fallible methods of solving a problem).*

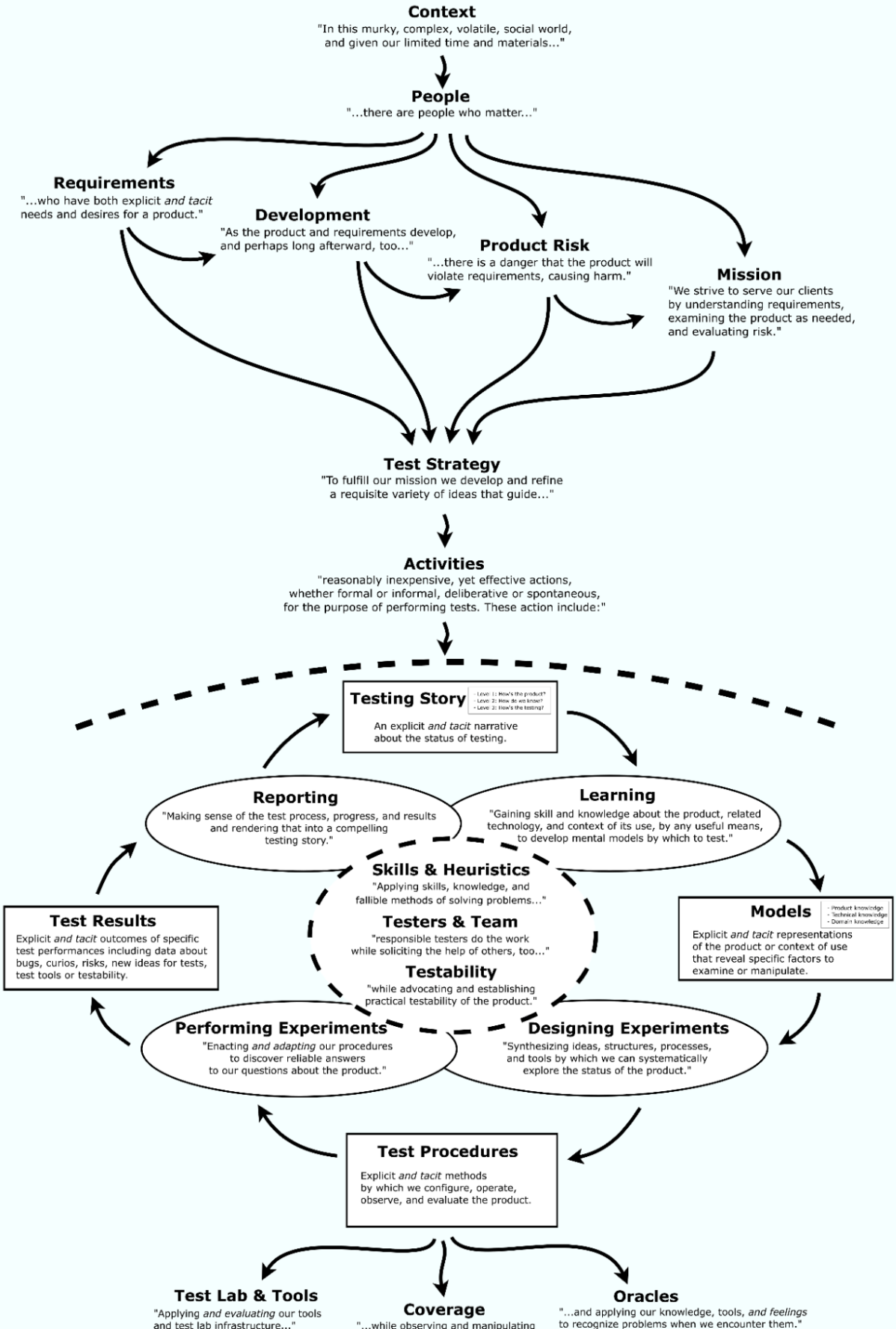
Rather than being a set of templates and rules, RST is a mindset and a skill set. It is a way to understand testing; and it is a set of things a tester knows how to do.

Being an amazing synthesizer, James Bach created the diagram, below, to illustrate rapid testing. Here is Bach's framing of the diagram:

There are many ways to diagram the testing process, but this is my current favorite.

This diagram is also an illustration of how test framing *works. Test framing is the process of relating everything and anything that you do while testing back up to the business. It is the process of putting a logical frame around your work to show its relevance. This is a story-telling process, and so this framework is expressed in the form of a narrative statement.*

A Rapid Testing Framework

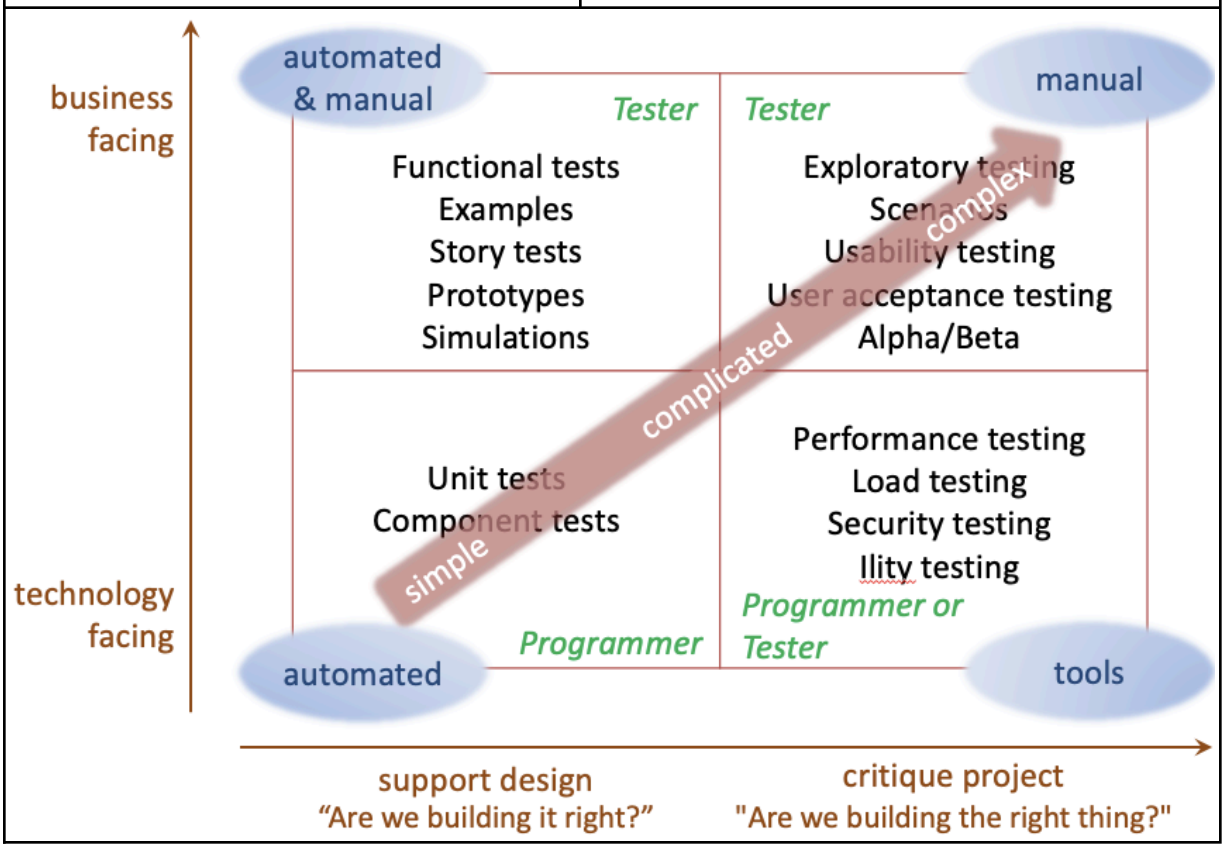


(from A Rapid Software Testing Framework, Satisfice.com)

Agile Testing Quadrant

The Agile Testing Quadrant is a 2x2 matrix that shows where unit tests fit within the larger scope of the many approaches to testing and running tests (checking). The complexity of what is being tested and of doing the testing increases as one moves from the lower-left toward the upper-right. This corresponds to the movement from the Cynefin framework's Clear (aka Simple) domain, to the Complicated domain, and into the Complex domain.

Upper left quadrant	Upper right quadrant
Testers use a combination of manual and automated tests that focus more exploring how well the design of a part of the system (e.g., feature, user story) will work for users and customers. Many of these tests are created and run by testers, though developers sometimes also do this work.	Testers do the creative work of seeing how well the entire system actually meets important business and user needs and constraints. These tests often are created and run by testers or other people closer to the user than developers.



Lower left quadrant	Lower right quadrant
Programmer's automate fairly simple technology-facing tests that support their design process. This helps ensure that the developer is designing the system well with respect to its architecture and the way the code works. This is where unit tests reside.	Programmers or testers use specialized tools to create complicated processes for testing non-functional requirements, such as how well a system handles a high load of usage, how secure it is, how well it scales, etc. These tests are about ensuring that the system works as needed. These tests often require expertise and a deep understanding of the nature of the related non-functional requirement being tested.

Self-Testing Code

Read Martin Fowler's article on self-testing code to get a glimpse into some of the reasons that writing automated unit tests is such an important part of being a professional software developer these days.

In James W. Grenning's Test-Driven Development for Embedded C (which he wrote because people kept telling him that it was impossible to do TDD for embedded systems) he talks about the "physics" of TDD. Without sufficient automated unit tests, one will almost certainly end up practicing "debug-later programming" in which there is a long time between when the bug was injected into the code and when it is fixed:

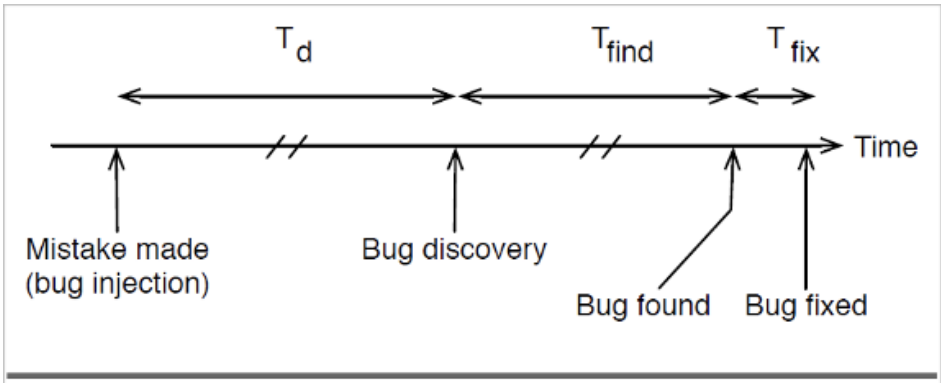


Figure 1.1: Physics of Debug-Later Programming

The goal of TDD is to reduce this time by discovering the mistake really, really quickly:

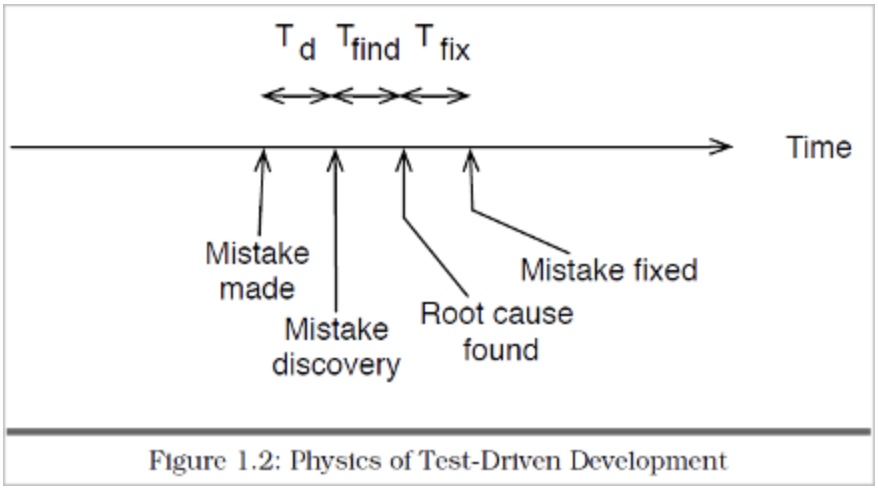


Figure 1.2: Physics of Test-Driven Development

Jeff McKenna, who was the coach for the team that invented Scrum, once told me (David Socha) a story about a time when he was working as an agile coach for a company that was developing software related to protein folding. The company was creating software that simulated molecular dynamics, which was very difficult to validate with external reference models, since so little was known about how proteins folded. To make sure that they were not making mistakes they adopted and evolved an extensive and excellent set of agile software engineering practices for management and development. This included using TDD for all of the code they wrote. They did such a good job that the average time between when the bug was injected (mistake was made by the programmer) and the bug was found was less than 10 minutes. Given that their defects were found and fixed so quickly, they didn't even need a defect database since there was never more than a couple of known defects.

How does one do unit testing?

Some of the key aspects of doing unit testing include:

- What to test?
- When to write the tests? Before you write the code? As you write the code? After you write the code? Each approach provides different advantages. Good development typically involves all three approaches: writing tests before, during, and after writing the code.
- How to write, and automate, the tests? There are many frameworks, with different frameworks working for different types of code. For example, testing core domain objects often uses an xUnit testing framework specific to the programming language (there are hundreds of xUnit frameworks), while testing the browser user interface used by a human may involve tools like Selenium.
- How many tests to write? What is a necessary and sufficient number and variety of tests?

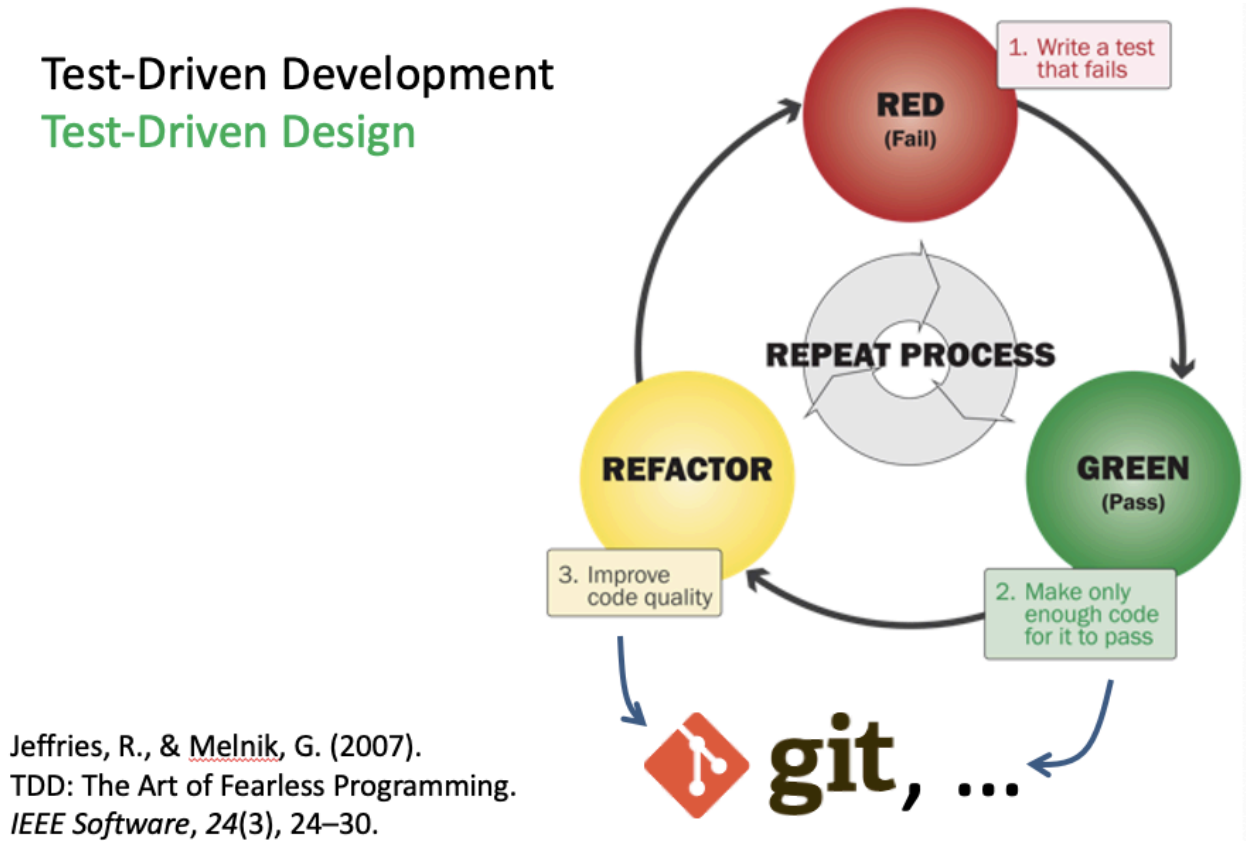
The core practice is to write automated tests to ensure that the code that you create works correctly. In general, which you write first (unit test or the code to be tested) is less important

than that you write both. In general, include unit tests with any code changes being committed so that you can achieve self-testing code.

> "You have self-testing code when you can run a series of automated tests against the code base and be confident that, should the tests pass, your code is free of any substantial defects." - Martin Fowler

Test-Driven Development - the red-green-refactor method

One way to do this is follow Test-Driven Development (TDD)'s **red-green-refactor pattern**, as Martin Fowler briefly discusses in TestDrivenDevelopment:



The steps to TDD are:

1. Write a test that fails
    a. Write ONE (1) test for the next small bit of functionality you want to add to the production code.
    b. Run that ONE (1) test and make sure it fails. This ensures that you are actually testing what you are trying to achieve. (I've heard several experienced developers talk about how sometime a new test passes … even though the code it was testing did not have worked that way! Ether the test was incorrect, or the code did not do what the developer thought it did.)

2. Make only enough code for that new test to pass
    a. Write JUST ENOUGH production code for that ONE (1) test to pass.
    b. Run all of the other unit tests to make sure you haven't broken anything else in the system.
    c. Commit the new improvements to your development branch.
3. Improve code quality
    a. If needed, refactor the new and/or old code to make it well structured.
    b. Run all of the other unit tests to make sure you haven't broken anything else in the system.
    c. Commit the new improvements to your development branch.
4. Repeat from step #1

Committing the code and tests before the next step allows you to easily abandon any changes that go poorly.

Separating adding changing functionality (step #2) from refactoring (step #3) reduces complexity, since you are only doing one type of work. Changing new functionality can be difficult. Refactoring can be difficult. Doing both at the same time is even more difficult. So, only do one at a time.

To see this in action, watch a few episodes of James Shore's Let's Play TDD screencasts in which he creates a real project via TDD (scroll down to the bottom to start with episode #1).

Doing TDD often seems counter-intuitive, especially to people who already have established their programming practices. While working in Microsoft, Arlo Belshee found getting experienced software developers to appreciate the benefits of TDD took about six weeks. The problem is that changing one's behavior takes time, and during the time of changing behavior performance often goes down.

And remember that TDD, like pair programming, were originally developed way back in the early 1960s, and then lost until being reinvented in the 1990s.

Refactoring

Keeping code clean and easy to modify for future needs is as important as writing good code in the first place. This is because not only is code read far more often than it is written, the contexts and desires of those using a piece of code keep evolving for the entire life of a system.

Here is how Martin Fowler frames refactoring. Also see his slidedeck.

> **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
>
> Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce

a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Refactoring lowers the cost of enhancements

When a software system is successful, there is always a need to keep enhancing it, to fix problems and add new features. After all, it's called software for a reason! But the nature of a code-base makes a big difference on how easy it is to make these changes. Often enhancements are applied on top of each other in a manner that makes it increasingly harder to make changes. Over time new work slows to a crawl. To combat this change, it's important to refactor code so that added enhancements don't lead to unnecessary complexity.

Refactoring is a part of day-to-day programming

Refactoring isn't a special task that would show up in a project plan. Done well, it's a regular part of programming activity. When I need to add a new feature to a codebase, I look at the existing code and consider whether it's structured in such a way to make the new change straightforward. If it isn't, then I refactor the existing code to make this new addition easy. By refactoring first in this way, I usually find it's faster than if I hadn't carried out the refactoring first.

Once I've done that change, I then add the new feature. Once I've added a feature and got it working, I often notice that the resulting code, while it works, isn't as clear as it could be. I then refactor it into a better shape so that when I (or someone else) return to this code in a few weeks time, I won't have to spend time puzzling out how this code works.

When modifying a program, I'm often looking elsewhere in the code, because much of what I need to do may already be encoded in the program. This code may be functions I can easily call, or hidden inside larger functions. If I struggle to understand this code, I refactor it so I won't have to struggle again next time I look at it. If there's some functionality buried in there that I need, I refactor so I can easily use it.

Automated tools are helpful, but not essential

When I wrote the first edition of Refactoring, back in 2000, there were few automated tools that supported Refactoring. Now many languages have IDEs which automate many common refactorings. These are a really valuable part of my toolkit allowing me to carry out refactoring faster. But such tools aren't essential - I often work in programming languages without tool support, in which case I rely on taking small steps, and using frequent testing to detect mistakes.

Today's Integrated Development Environments (IDEs) have great refactoring support either built in or available via plugins / extensions. You likely will be impressed by your IDE refactoring support. Take advantage of it! For instance, Visual Studio Code has built in support for a few refactoring methods, such as:

- Extract method - extracts selected code into a new method
- Extract variable - converts selected expression into a variable
- Rename symbol - changes all instances of this symbol name in the codebase

Some Visual Studio Code extensions support dozens of refactoring methods (see 5 VS Code Extensions That Make Refactoring Easy).

What tests to write

This section touches upon a few of the many ways to reason about which tests to write, how many tests are sufficient, etc.

A test case is a specific set of inputs for a system / module / method that leads to an expected set of output values. Testing involves the process of coming up with useful test cases.

Equivalence Partitioning and Boundary Testing

Given that there is an infinite variety of tests that one could perform on any given system, testers have developed a range of methods for coming up with sets of test cases that have a higher chance of finding issues. Two related blackbox testing techniques for doing this are *equivalence partitioning* and *boundary testing*.

Consider the following example problem. You want to test that an online bus ticket purchasing system correctly computes the bus fare based upon the following requirement:

> *Children under two ride the bus for free. Youth under 18 years old pay $10, adults pay $15, and senior citizens pay $5.*

That requirement divides the inputs into four categories, also known as partitions or domains:

A. age < 2 → free
B. 2 ≤ age < 18 → $10
C. 18 ≤ age < 65 → $15
D. 65 ≤ age → $5

This example illustrates two interesting aspects about partitions:

- Any age in each of those partitions should produce the same result, so there is little value for testing every number in each partition.
- Errors often occur at the boundaries between partitions, e.g., did the programmer incorrectly "<" instead of "≤"?

Thus, to test the bus calculation code, we can select one representative value from each partition, plus values immediately on the edges between partitions. For the above example, we could select the following values:

Table 1: Initial set of equivalence partitions for bus tickets, showing which unit tests would be useful to write.

Partition	Partition values	Values to test	Expected result	Reason for this test
Children under 2	$0 \leq x < 2$	0, 1	Free	
Youth	$2 \leq x < 18$	2, 10, 17	$10	
Adult	$18 \leq x < 65$	18, 30, 64	$15	
Senior citizen	$65 \leq x$	65, 70	$5	

The above table tests for legitimate values.

It's also important to consider some of the illegitimate values, and on what can go wrong. What can go wrong in the above situation includes more than just boundary conditions and values within a partition not acting equivalently.

As Bret Pettichord notes in his illuminating and very readable article "Testers and Developers Think Differently: Understanding and utilizing the diverse traits of key players on your team", testers are good at focusing on what can go wrong.

A tester would also want to test that the **software gracefully handles illegitimate inputs**. In the above example, illegitimate inputs include a partition of negative numbers, and a partition of ages that are beyond the age that a person could be expected to live to. One should test inputs in those partitions to ensure that invalid inputs are handled gracefully. A graceful handling might include presenting an error message explaining why the input is invalid.

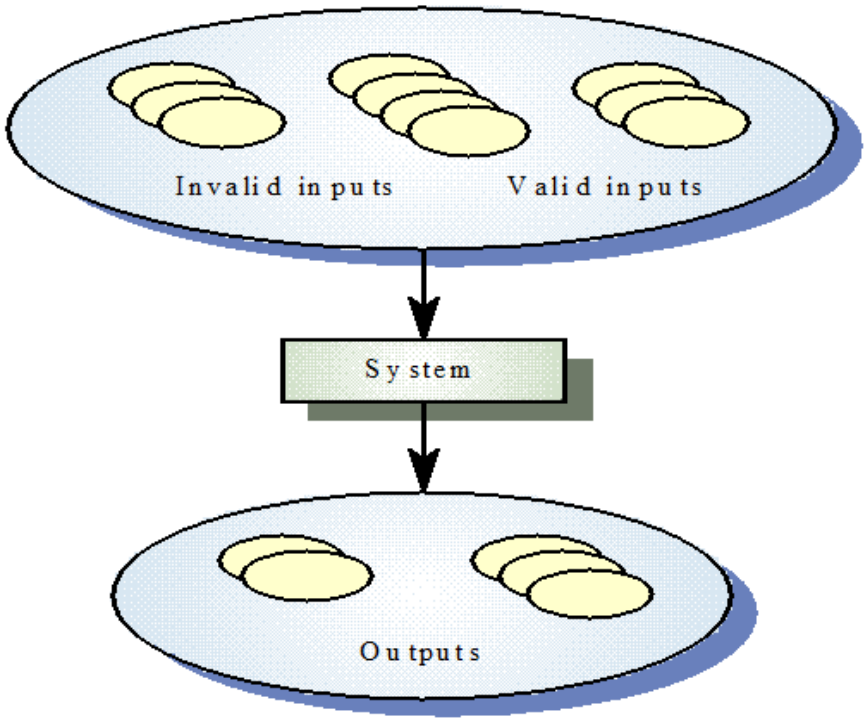
Below is a more extensive set of partitions that might be useful to test for the above example. It adds three partitions: one for negative numbers, one to check that three-digit ages work, and one to test for unreasonably large ages.

Table 2: Fuller set of equivalence partitions for bus tickets.

Partition	Partition values	Values to test	Expected result	Reason for this test
Invalid	$X < 0$	-1	Invalid age	Invalid age
Children under 2	$0 \leq x < 2$	0, 1	Free	
Young people	$2 \leq x < 18$	2, 10, 17	$10	

Adult	18 ≤ x < 65	18, 30, 64	$15	
Senior citizen	65 ≤ x < 100	65, 70	$5	
Senior citizen	100 ≤ x < 110	100, 105	$5	3 digits works?
Invalid	110 ≤ x	110, 120	Invalid age	Nobody lives that long

Equivalence partitioning is analysis of module with respect to understanding what set of inputs lead to what set of outputs and vice-versa. It is used to help identify boundary limits / edge conditions for test cases and, hence, test data. Ideally, you want a test case for each unique partition, as well just inside each edge of each partition. The set of partitions identified will depend upon whether it is blackbox or whitebox testing.



(Sommerville, I. *Software Engineering*)

For more on equivalence partitioning (also known as domain analysis), and boundary value analysis:

- Michael Robinson's Guideline: Equivalence Class Analysis
- Guru99: Boundary Value Analysis & Equivalence Partitioning with Examples (pdf in the References)

How to name unit tests?

The name of a unit test should describe the design decision / concern that is being tested. For instance, for the partitions listed in Table 2, above, consider the following names for the test of age = -1:

Test method name	**Comments on the test method name**
NegativeOne	NOT a good name. It indicates value to be tested, but not WHY it is useful to test “-1”.
NegativeAgesAreInvalid	A good name. It states an invariant of how the method should work. An assertion.

Both the second and third names reveal information about how the design should work, or quality concerns the developer had about the method. They help define the “contract” about how this method should work.

Here is a possible set of unit test names for the partitions listed in Table 2, above:

Partition	Partition values	Values to test	Example names for unit tests
Invalid	X < 0	-1	isNegativeAgeInvalid
Children under 2	0 ≤ x < 2	0 1	isZeroAChildUnderTwo isOneAChildUnderTwo
Young people	2 ≤ x < 18	2 10 17	checkMinimumAgeForYoungPerson checkIntermediateAgeForYoungPerson checkMaximumAgeForYoungPerson
Adult	18 ≤ x < 65	18 30 64	checkMinimumAgeForAdult checkIntermediateAgeAdult checkMaximumAgeForAdult
Senior citizen	65 ≤ x < 100	65 70 99	checkMinimumAgeForSenior checkIntermediateAgeSenior checkMaximumTwoDigitAgeForSenior
Senior citizen	100 ≤ x < 110	100 105	doesSmallestThreeDigitAgeWork doesAnotherThreeDigitAgeWork
Invalid	110 ≤ x	110 120	isInvalidIfTooOld

How many tests to write? Using code coverage

Code coverage refers to how well a set of tests “cover” the code. As Atlassian’s article An introduction to code coverage notes, There are many different ways to measure coverage, such as:

- **Function coverage**: how many of the functions defined have been called.
- **Statement coverage**: how many of the statements in the program have been executed.
- **Branches coverage**: how many of the branches of the control structures (if statements for instance) have been executed.
- **Condition coverage**: how many of the boolean sub-expressions have been tested for a true and a false value.
- **Line coverage**: how many of lines of source code have been tested.

It's best to work with your team to agree upon the type and amount of coverage that you are aiming for. Note that achieving 100% coverage does not guarantee against defects.

This video does a good job of describing the basics of writing a few unit tests for some sample code, and using a tool to determine the code coverage: 100% CODE COVERAGE - think you're done? Think AGAIN. The video is about Python, but there are similar tools for most modern programming languages.

What not to test?

Learning what to test and what not to test takes practice. Most people write too few tests. Thus, a best practice is to start by writing more tests than you think are reasonable. This will help you figure out which ones are not needed.

Common difficulties with doing unit testing

Following are some situations which make it very difficult to do automation unit testing:

- When there are timing issues, such as multi-threaded code or concurrency. Race conditions make it very difficult to debug and nearly impossible to create an automated test to reliably reveal the race condition. Instead, developers might use formal modeling techniques to try to proof that their code is free of timing issues.
- Remember that unit tests are appropriate for the lower-left quadrant of the Agile Testing Quadrant, and the bottom tier of the Testing Pyramid.

Is TDD Dead?

On April 23, 2014 David Heinemeier Hansoon released a blog post titled TDD is dead. Long live testing. This led to quite a furor on the web. Which led to Martin Fowler, Kent Beck, and David Heinemeier Hansson recording a series of conversations that shows the debate is much more nuanced and that TDD is not dead. Instead, all three agreed that TDD is a useful method for some people in some contexts, and doesn't work well in other contexts.

This is true for most software development processes: they are context-sensitive. These processes keep evolving as we better understand what approaches work well in what contexts, as build tools to aid our processes, and as we come to understand the types of mindsets that help us be effective.

While TDD may be highly effective and a “best practice” in many contexts, TDD does not work in every situation. As Fred Brooks noted in his 1987 IEEE Software article No Silver Bullet Essence and Accidents of Software Engineering, there are no silver bullets.

The Ethics of Unit Testing

> *“Above all, we test because it is the responsible thing to do. We have a duty of care toward our teams, our organizations, our customers, and society itself. Releasing poorly tested software would be a breach of that duty.”* - James Bach, Rapid Software Testing Methodology
>
> *Confirmation bias: “The tendency to selectively search for or interpret information in a way that confirms one's preconceptions or hypotheses” - Cognitive Bias, A. Wilke, R. Mata, in Encyclopedia of Human Behavior (Second Edition), 2012 - see Science Direct*

We who develop and maintain software hold much power and much privilege. Software is part of almost every aspect of our lives, personal and professional. Software engineers **have a large responsibility for doing our jobs well**. That includes demonstrating that the code that we write *actually works*. That it does what is needed in a way that is secure, aligns with privacy standards, is sufficiently performant, is accessible, etc.

Don’t just *think* that your code works; *demonstrate* that it works. Provide yourselves and your colleagues with evidence showing that your code works. Avoid confirmation bias. Part of being a *professional* software engineer is being appropriately suspicious about whether your code works. Doing that helps ensure that we are aligning with the ethical mandates of the ACM Code of Ethics and Professional Conduct and IEEE Code of Ethics regarding being ethical practitioners.

Good scientists work work to disprove their conjectures before claiming they are true. Science only claims a conjecture is correct once the community has agreed that sufficient effort has been done to disprove the conjecture.

For any non-trivial system, that means writing and frequently running tests that ensure not only that the code did the right thing when you or someone else first wrote it, but also that the code keeps doing the right thing regardless of how other parts of the system around it change.

> **Meta**: Don’t believe your code works correctly until you have tests that show that it does.

And don’t trust that your inputs are correct. Learn from electrical engineers who have are trained to *expect* that the inputs to their circuits, like electrical power, will sometimes be outside the specified limits. To manage that, electrical engineers design their circuits to gracefully handle unexpected extremes. Thus, instead of melting down a system, they trip a circuit breaker.

> **Meta**: Assume that the other parts of the system around your code will change, and will at times provide unreasonable values to your code. Write your code to fail gracefully if the other parts of the system don't work as expected / needed. And use a mechanism such as throwing an Exception to inform the caller that your code has failed.

We need to be confident that our code works. Not confidence born of ignorance or untested assumptions or hubris, but confidence that comes from empirically testing the code we create.

If we are not confident about our code, we shouldn't deploy it.

Resources

- Martin Fowler's Software Testing Guide