Software Testing Fundamentals

- Testing is and always has been an important part of software development.

- Even if you think back to when you were first learning to program, before you probably knew anything about software testing as a formal concept, you probably still tested your code by inputting data into your programs and comparing their actual behavior against the behavior you expected.

- In fact, this practice of *comparing actual behavior against expected behavior* is the essence of software testing.

- Indeed, even the practice of ***manual testing***, where someone (maybe the developer, maybe a software tester, maybe even an end user) runs some code, manually enters some input into it, and then monitors the code's behavior, still plays a role in software development today.

- However, when we talk about "testing" today, we're more typically referring to ***automated testing***, where a suite of software tests is executed against a code base. Each test automatically runs some part (or the whole) of the code being tested and compares its actual behavior against the expected behavior.

- Automated testing is a developer-driven practice. Specifically, automated tests are typically designed and implemented by the same developers implementing the software being tested.

- For this reason, it is important to understand the fundamentals of automated software testing, including why we test our code, what kinds of automated software tests we can perform, and how automated software tests work at a high level. We will study those fundamentals here.

Why perform software testing?

- At a very high level, the goal of automated software testing is to prevent software bugs from infiltrating production software and affecting software users.

- This makes sense, since the cost of software bugs can range anywhere from confusion to frustration to discomfort to expensive disaster to fatality.

- Indeed, depending on whose report you read, inadequate software testing costs the US economy anywhere between tens of billions of dollars to more than two trillion dollars, and this does not even account for serious non-economic impacts, like lives lost.

- However, while factors related to the prevention of damage are very important, there are other compelling reasons why automated testing is a good idea. Perhaps foremost among these is the fact that automated testing can make *our own* lives better as software developers.

- In particular, well-designed automated tests can give us confidence that our code will work as intended for the people using it. This confidence is important because it allows us to *change* our code without worrying too much about breaking it. In other words, good tests can give us confidence to add new features, refactor our code, etc. and to rely on the tests to quickly catch mistakes we make, before they make it into the hands of our users.

- In addition, the act of implementing tests can actually help clarify what you want your software to *do* and can lead to better-designed systems by forcing you to consider possible use cases, boundary cases, architecture decisions, etc.

- In fact, investing the time to write good tests up front can save you time later in the development cycle by reducing the effort required to maintain your software.

- For all these reasons, we want to learn to be good software testers. This is what we'll begin to do here. As we learn, though, it's important to keep in mind that, while good tests will give us confidence that our code works as intended, they can't guarantee that it's bug-free. As Edsger Dijkstra (of Dijkstra's algorithm fame) said: *"Testing shows the presence, not the absence of bugs."*

What is a (good) software test?

- Let's start by digging into exactly what a good automated software test is.

- At its core, every good automated software test has the following characteristics:
    - It focuses on **a single, specific behavior** within a software project, e.g. calling a specific function, invoking a specific API method, or a specific element within the user interface (e.g. a specific form or even a button).

    - It is based on **a specific input**, e.g. a specific set of function parameters, a specific value passed to the API, or a specific type of user interaction (e.g. submitting a form).
    - The behavior being tested has **observable results**, e.g. a function's return value, the response body sent by an API method, or the change in the UI after the user submits a form.
    - **We know the expected results** of the behavior being tested when provided the specific input we're using in the test.
    - The test occurs within **a controlled environment**, e.g. within a single, isolated process on a dedicated machine.

- Given that these characteristics are satisfied, every software test's ultimate job is **to compare the actual/observed results of the software being tested against the expected results**. If the actual results match the expected results, the test passes. Otherwise, the test fails.

- Below are a few other things to pay attention to when trying to implement good software tests.

Tests should be unchanging

- Good tests are designed to capture the correct behavior of the software being tested. Thus, as long as the correct behavior of that software doesn't change, we shouldn't expect our tests to change, either. In other words, **once a test is written, it should not need to change**.

- In particular, there are several different kinds of changes we might make to our software that shouldn't change its existing behaviors, including refactoring, adding new features, and fixing bugs.

- A good software test has a role in responding to each of these different kinds of changes, generally by ensuring that our software's existing behavior doesn't change as we change the code.

- Thus, if we find ourselves needing to modify existing tests when refactoring, adding new features, or fixing bugs, it may be an indication that our changes to the code are unintentionally impacting the existing behavior of our software.

- Of course, poorly written tests may actually need to be changed. We'll look at some characteristics of poorly written tests below.

Testing implementation details

- When writing tests, we may be tempted to rely on internal implementation details within the tests themselves.

- For example, we might want to implement a test by reading the value of a specific variable, calling a private function, or otherwise accessing parts of the code that only we, as its developers, are aware of or have access to.

- For example, imagine we are testing a specific feature in an application in which a dialog opens when the user clicks a specific button. Imagine that the mechanism that controls the display of the dialog is a boolean value named `showDialog` and that a click of the button in question sets the value of `showDialog` to `true`.

- We might be tempted to implement our test by ensuring that the value of `showDialog` is `true` after the button in question is clicked. This would be an example of relying on implementation details within a test.

- In general, we want to try to avoid using implementation details in our tests like this because it makes our tests brittle. For example, in our scenario above, what if we refactored our code and renamed the value `showDialog`? Or, what if we made a bigger refactoring that eliminated `showDialog` altogether?

- These changes would both break a test that relied on reading the value of `showDialog`, requiring us to change the test.

- Thus, in general, it is best practice to **implement tests that interact with our software the way a user would**, e.g. by using only public APIs (not private APIs and internal values) and by directly observing changes to the state of the UI.

- In our example scenario above, a better test that interacted with the software the same way as the user would examine whether the dialog was actually displayed in the UI when the button was clicked (e.g. by checking the UI for the presence of the text displayed by the dialog).

- As the course goes on, we'll explore tools and techniques that make it easier for us to write tests that interact with our software the way the user would instead of testing implementation details.

- Note, though, that there are cases in which you *will* want to test implementation details, especially when an important code behavior is not apparent from the public interface of that code.

- For example, you might want to test that a data access function correctly reads from a cache instead of from a database when appropriate. This behavior will likely not be apparent in the return value of this function (i.e. the function's public interface), so testing it may require peeking into the internals of the function.

The developer user vs. the end user

- When we say you should "implement tests that interact with our software the way a user would", it's important to remember that that there can be two different kinds of users who may use our software:
    - **The end user** – a person who is running the code you're writing, e.g. clicking buttons and typing text into it.
    - **The developer user** – another developer who is invoking the code you're writing from their own code, e.g. someone who's incorporating a class you wrote into their own application.

- Regardless of which of these users our code will have (and it could have both, e.g. if we were implementing a reusable UI component, like a dialog), we should always implement our tests using the interface that user would use, i.e. clicks, typing, etc. when our code will have an end user and public method calls, etc. when our code will have a developer user.

Determinism and flaky tests

- Another important characteristic to strive for in an automated software test—though it isn't always possible—is **determinism**. In other words a good test will always generate the same results (i.e. pass or fail) given the same testing conditions (i.e. testing the same code with the same inputs).

- Tests that are non-deterministic are known as ***flaky tests***. These are tests that sometimes pass and sometimes fail, even though the code being tested and the test itself haven't changed.

- Flaky tests erode developers' confidence in their tests and can become a huge time sink, as developers try to track down the cause of failures from flaky tests. For this reason, we want to try to avoid implementing flaky tests. We'll talk later

in the course about strategies for doing this.

- Again, though, note that sometimes test flakiness is unavoidable. For example, a particular test may simply require a call to a network service, and uncontrollable factors like network issues, service unavailability, etc. may cause this test to be flaky.

- In situations like this, we can take steps to try to mitigate the effects of flakiness, e.g. by automatically rerunning a failing test.

Different types of automated software tests

- There are three main types of automated software tests that we'll focus on in this course: ***unit tests***, ***integration tests***, and ***end-to-end tests***. The distinguishing characteristic that differentiates these three types of test is scope, i.e. how much code is being tested.

Unit tests

- A ***unit test*** is designed to validate a small, focused piece of code, such as a single function or an individual class (the name "unit test" comes from the fact that these tests validate "units" of code).

- For example, a unit test might ensure that a given function returns the correct value given known inputs as arguments or that the submit handler for a form correctly validates user input.

- In general, because unit tests focus on small units of code, they themselves are small and generally easy to implement. For this reason, we will often implement multiple unit tests to validate a given unit of code under a variety of different conditions, or ***test cases***.

- Let's say, for example, that we are implementing unit tests for a form in a web application that collects the user's shipping address. We might implement one test (or more!) that ensures that the form accepts a valid address when it is input. We might also implement tests that ensure that the form rejects various kinds of invalid input (e.g. missing ZIP code, missing city name, etc.) or doesn't break for unusual input (e.g. a city name that's 257 characters long).
    - Test cases of the latter type, that involve unusual, often rare inputs, are known as ***boundary cases***. It is important to include these kinds of test

cases in our test suite.

- Taken together, these multiple tests help give us confidence that our unit of code behaves correctly under many different circumstances.

- The main limitation of unit tests (which many clever gifs have been created to demonstrate) is that they don't ensure that units of code work correctly *together*. This is where integration tests come into play.

Integration tests

- An ***integration test*** is larger in scope than a unit test. Integration tests are specifically designed to validate *interactions* between multiple software components.

- For example, whereas a unit test might simply verify that our app's shipping address form correctly validates its submitted input, an integration test might verify that when a valid address is submitted into the form, the form correctly passes that address to a shipping cost calculator, which in turn calculates the correct shipping cost. This integration test would specifically be verifying the interaction between the form and the shipping cost calculator.

- Another integration test might make sure the user's shipping address was correctly stored in the database after it is submitted. This test would be validating the interaction between the address form and the database.

- Because integration tests validate that different parts of our code interact correctly together, they can give us more confidence than unit tests that only validate units of our code in isolation.

- However, at the same time, because integration tests are wider in scope, they can be more complex to implement and maintain.

- Thus, unit tests and integration tests represent a tradeoff that must be balanced. We will discuss approaches to achieving this balance in a minute.

End-to-end tests

- ***End-to-end tests*** are tests at the largest scope and validate large connected portions of the codebase running together, such as an entire page or user flow. These tests may involve verifying the behavior of many different application

processes running on different machines (e.g. web client, web/API server, database server, etc.).

- End-to-end tests directly validate an application's behavior through its user interface and are typically implemented using some kind of "robot" that behaves like a user, clicking around the application, typing things, etc.

- For example, we might implement an end-to-end test to validate the entire checkout flow of an e-commerce application, where a "robot" moves through the process of clicking the "checkout" button, entering their shipping address and payment information, clicking the "complete checkout" button, and receiving a transaction confirmation.

- The end-to-end test implementation would encode the expected behavior of the application at each step of this process, and the testing "robot" would verify that the actual behavior of the application matches that expected behavior.

- As you might expect, end-to-end tests are the most expensive (in terms of time and resources) to implement, run, and maintain of the three types of automated tests we've looked at here.

- However, because end-to-end tests use our entire application in just the way a real user uses it, they give us the most confidence that our application will work for real users the way we intend it to work.

Other types of automated software tests

- There are a few other types of automated software testing that we won't focus on much in this course but that still bear a mention here.

- ***Performance tests*** are tests that verify that the resource usage (e.g. runtime or memory usage) of a piece of software is within an acceptable limit.

- A performance test might, for example, run a piece of code and time how long it takes to finish running. If execution takes too long, the test fails.

- ***Load tests*** are tests that verify whether an application can maintain its performance at scale.

- Load tests often simulate a large number of users all using an application at once. If the application slows down significantly or begins to exhibit other errors

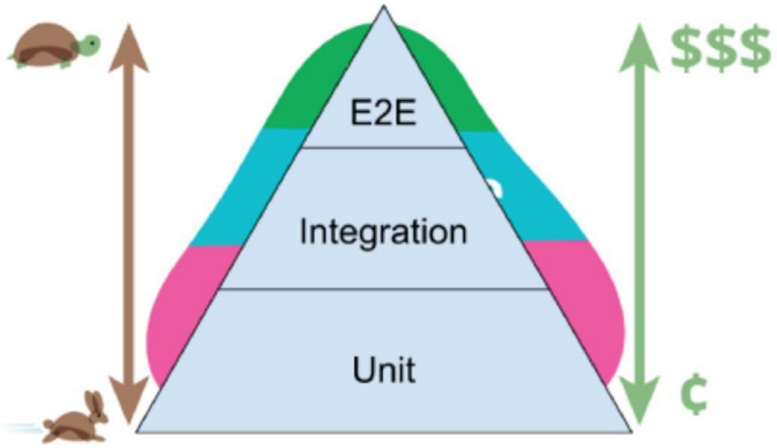
during a load test, this can be a sign that the code must be streamlined (e.g. reducing the number of API calls the application makes) or that larger architectural changes (e.g. more servers) may be needed.

Balancing different types of automated tests

- In general, most production software projects will benefit from a mix of unit, integration, and end-to-end tests.

- What proportion of each different kind of test we should implement to best balance their respective advantages and disadvantages is a question that has been debated.

- One model that has been traditionally popular for thinking about how to balance the proportion of unit tests vs. integration tests vs. end-to-end tests is the "testing pyramid":



Courtesy of Kent C. Dodds (h/t to Martin Fowler and the Google Testing Blog)

- The idea behind the testing pyramid is that unit tests are the fastest and least expensive to write, run, and maintain, so these should form the bulk of our tests. As we move up the pyramid to integration and end-to-end tests, which get increasingly slower and more expensive to write, run, and maintain, we should have fewer and fewer of these kinds of tests.

- Thus, according to the testing pyramid, most of our tests should be unit tests, with integration tests comprising a smaller proportion of our tests and end-to-end

tests comprising a smaller proportion still.

- Another model that was more recently introduced is the "testing trophy":

Courtesy of Kent C. Dodds

- The "testing trophy" model suggests that most of our tests should be integration tests, with unit tests comprising the next-largest proportion of tests and end-to-end tests comprising the smallest proportion.
    - Note that the "testing trophy" image above also suggests the importance of ***static tests***, which are tests designed to catch typos, type errors, etc. Static tests are often executed *as we code*, e.g. by a ***linter*** integrated into our code editor.

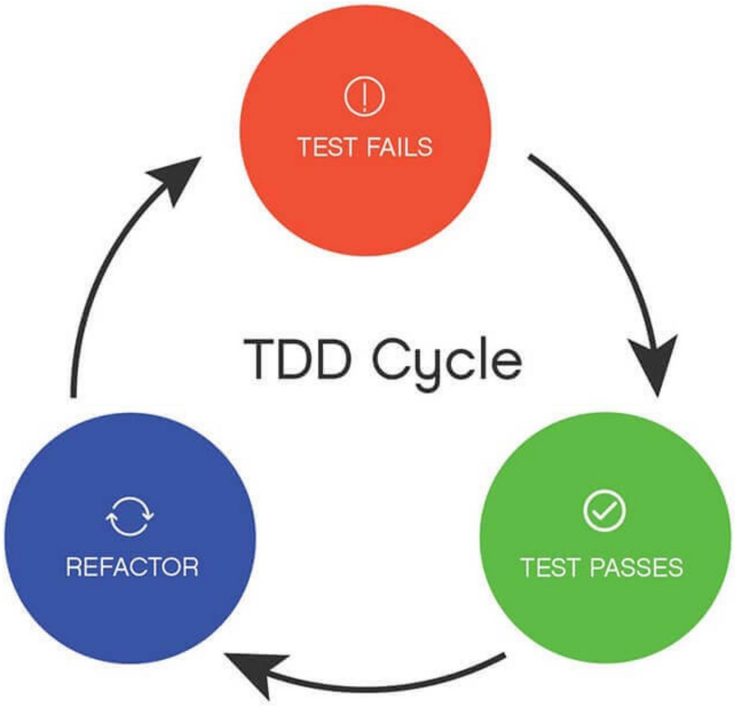
- Proponents of the "testing trophy" argue that the testing pyramid doesn't reflect the amount of confidence we get from each different kind of test. Generally, confidence gained from a single test increases as we move up the pyramid.

- In addition, "testing trophy" proponents argue that testing tools have improved both in speed and ease of use to the point where it is now more practically feasible to maintain and run a larger suite of integration tests.

Test-driven development

- ***Test-driven development*** (or just ***TDD***) is a development strategy that places automated testing at the core of software development.

- TDD consists of a cycle of testing, coding, and refactoring that's often called the "red, green, refactor" cycle:



Courtesy of Kent C. Dodds

- The steps of this cycle, in more detail are:
    - ❌ **Red** – Start by writing a test that contains just enough code to test the behavior of the next feature you want to implement. Since the feature isn't implemented yet, this test will fail (giving you a red error message).

    - ✅ **Green** – Next, write just enough code to make your test pass (giving you a green success message) without worrying too much about design or elegance.

    - 🔄 **Refactor** – Once your test is passing, refactor your code to make sure it's clean and well-written. Your test should continue to pass as you refactor, and if it fails, that's a signal that your refactoring broke something.

- This cycle is repeated for each new feature/behavior you want to add to your code.

- One of the major benefits of TDD is that writing the test first forces you to think first about your code's interface and behavior. Implicit in the "red" phase of the red, green, refactor cycle is an exercise of design.

- In other words, in order to implement a test for code you haven't written yet, you need to make decisions about how you will invoke that code, what its behaviors will be, and how it will communicate its results back to you.

- Making these decisions within the context of a test forces you to think about your code from the perspective of its user (in this case, the test). Doing this up front can help yield code with an interface that's clear and simple, i.e. code that's easy to use and understand.

- One of the keys to successfully employing TDD is working in very small increments. We start by implementing a single test that's very limited in scope, then we write just enough code to make that test pass. We repeat this process until an entire feature is implemented. Oftentimes, this will require breaking a problem that itself seems quite simple into even smaller steps.

- Importantly, TDD does *not* require that we implement *all* of our tests up front. In fact, this is an anti-pattern in TDD. Instead, with TDD, we're always working on just one test at a time.

- To understand what it looks like to work in very small increments using TDD, I'd strongly recommend reading through the "A TDD Example" section from the "Test-Driven Development" chapter of The Art of Agile Development.

Limits of automated testing and important types of manual testing

- Automated software testing is an important part of modern software development, but it's important to recognize that automated testing is not suitable for all testing tasks, and there are situations where manual testing is still necessary and important.

- Situations where manual testing can still be useful are often ones that require human creativity or judgment.

- Security testing, for example, often involves trying to identify complex and subtle vulnerabilities in a system. This is something that humans can do much better through a creative process known as ***exploratory testing*** than an automated system.
    - Of course, once a human has identified a security vulnerability in a system, an automated test can be created to ensure that the vulnerability remains fixed.

- Similarly, testing the quality of things like search results, audio, video, etc. is hard to automate, since this kind of quality is often a matter of human judgment. These kinds of judgments are usually best accomplished through systematic manual testing.