Testing Essentials

What testing is, why it is important, what should be tested, and how should it be done.

Dylon Edwards

March 9, 2020

Software testing is an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test. [2] The following document delves deeper into what testing is, why it is important, what should be tested, and how it should be done.

Contents

List of Figures

1. What is testing?

Software testing, at a high level, is an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test. [2] At a low level, there are three core types of testing (unit testing, regression testing, and integration testing) and at least eight other types that should be considered: system testing, smoke testing, acceptance testing, performance testing, security testing, sanity testing, validation testing, and health checks.[1]

[1] Software testing definitions are debated and occasionally ambiguous; these are how I interpret them.

1.1. Unit testing

Unit testing ensures individual methods and functions (the smallest units of design [23]) of the classes, components, and modules used by your software work as expected [14]. Unit testing yields the most granular diagnostics of a system – should a sufficiently-well tested system break, the unit tests will describe exactly where the problems lie.

A well designed system will make the majority of unit tests trivial and simple. Some paradigms and rules of thumb to ease the testing process will be discussed in a later section.

1.2. Regression testing

Regression testing seeks to uncover software errors after changes to the program have been made (e.g. bug fixes, new functionality, and refactoring). [22] Once a problem has been identified, an important step in closing it is to ensure it never resurfaces. The best way to do this is to write a regression test, which may be either through unit testing or integration testing, depending on the scope of the problem.

1.3. Integration testing

Integration testing verifies that different modules or services used by your application work well together. [14] These typically involve testing the application through high-level APIs, including chaining API calls together to obtain expected results.

1.4. System testing

System testing Ensures that your application works as expected under different environments (e.g. operating systems or hardware). [31] Although your application may work well on a GPU-enabled system, it may be beneficial for it to also run on a CPU-only system.

1.5. Smoke testing

Smoke testing manually verifies that no defects exist in the system (preferably according to a test plan). [21] This is what engineers are doing when they run their system as part of the development process.

1.6. Acceptance testing

Acceptance testing verifies that the software meets the needs of its consumers; the verification is performed by the consumers. [21] Alpha testing, beta testing, and release candidates are forms of acceptance testing.

1.7. Performance testing

Performance testing determines how well the system operates under a heavy load and/or the maximum load it can handle. [21]

1.8. Security testing

Security testing verifies the system is safe from internal and external threats. [21] Examples of security testing include fuzz testing, penetration testing, and vulnerability testing.

1.9. Sanity testing

Sanity testing ensures a system or service is ready to execute. Examples of sanity testing include verifying that the application's dependencies exist, that the application has the necessary permissions to accomplish its tasks, that the application is configured appropriately, and that the application has access to the resources it needs.

1.10. Validation testing

Validation testing verifies that a system or service is running as expected. Such tests may include verifying that the process is running, is listening to the expected ports, and is responsive to requests.

1.11. Health checks

Health checks report the status of a running system (or the lack thereof). Health checks may include such things as verifying the application has enough free memory and disk space, that it may write to the disk (logging), that it is able to communicate with its database, and that the CPU load is sufficiently low for it to accept new requests.

A common usecase for health checks is that of a load balancer: the load balancer accepts requests by way of a VIP and delegates them among the hosts behind it. The load balancer needs to know which of the hosts are healthy and able to receive requests. The target service on each host has a health check handler that accepts special pings from the load balancer and returns a 200 status (Okay) if all is well. Should the service return anything else or fail to respond several times, consecutively, it is taken out of the availability pool until its health is restored.

2. Why is testing important?

Every untested statement adds technical debt. Technical debt is a metaphor that describes the gap between the current state of the system and the ideal state of the system, usually in situations where a compromise is made during development to meet demands in one dimension, such as lead time, by sacrificing work in another dimension, such as architecture or automated testing. [30]

The metaphor compares design choices to economics: by not doing something during development, one takes out a loan against the system and pays for it in future costs. [30] Like financial debt, technical debt can be necessary for moving the project forward, but also like financial debt it can be catastrophic in the long term if it is not repaid in a timely fashion.

> when a project is new, there is no debt and reacting to changing requirements is easy. As technical debt is accumulated, agility is lost. The catastrophic case is when it becomes (or is perceived to be) easier to write an entirely new system rather than modifying the existing system to meet new requirements
>
> ~ *Jason Walker, SDE III at Amazon*

Some of the tasks a well-tested system ease include:

- Boosting customer confidence
- Documenting systems
- Adding new features
- Refactoring existing code
- Updating dependencies
- Monitoring system health
- Migrating systems
- Changing maintainers

> I would argue that everything other than "boosting customer confidence" could be classified as "boosting developer confidence". Boosting developer confidence is important because it enables what is probably the #1 best tool to combat technical debt: continuous improvement (aka "The Boy Scout Rule"). After all, I don't pay off my mortgage in lump sums, but slowly over time, with the discipline of not missing my monthly bill. Likewise, instilling a culture and discipline of continuous improvement in a team is – in my experience – the best way to pay off technical debt. And the way to instill a discipline of CI is to make it easy to make small changes and refactoring to software. This leads to what I see as a virtuous cycle:
>
> - Quality software is easy to test.
> - Software that is well-tested improves developer confidence.
> - Confident developers are more likely to engage in the discipline of Continuous Improvement.
> - CI improves the quality of software.
> - Repeat.
>
> ~ *Jason Walker, SDE III at Amazon*

Test-driven versus test-deferred development times.

Test-driven development time scales linearly[a] with the number of features because you write their tests as you develop them. Once their tests have been written, they are run automatically and are, for all practical purposes, forgotten (unless they fail). This is particularly important because it means testing complex systems is feasible. Development time while continuously deferring tests (test-deferred development) scales quadratically because every feature must be manually tested each time a new one is added. The work required to test the system increases by one unit per feature, leading to a cumulative development time of, $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. [12] This quickly leads to impractical or infeasible development times, even for relatively simple systems.

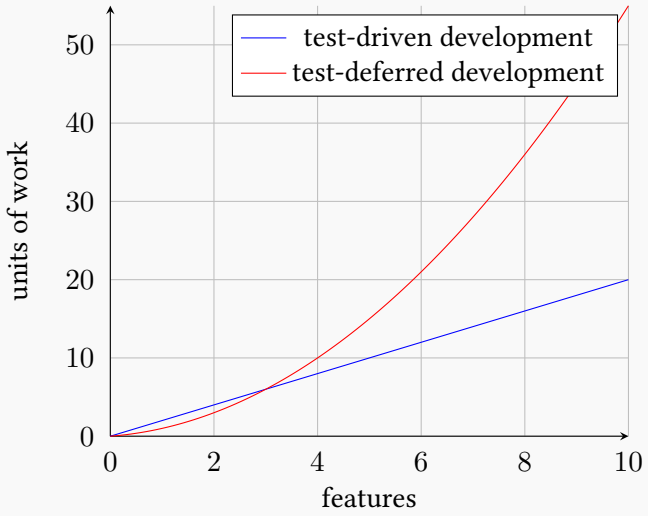

Figure 1: Test-driven versus test-deferred development times.

[a] Test-driven development time is $\mathcal{O}(2x)$ because it requires roughly twice the work to implement each feature but a constant amount of work to test the application. Test-deferred development time is $\mathcal{O}\left(\frac{n(n+1)}{2}\right)$ because it requires a constant amount of work to implement each feature but a quadratic amount of work to test the application.

This brings us to the very important notion of return on investment (ROI). Return on investment is a performance measure used to evaluate the efficiency of an investment or to compare the efficiency of a number of different investments. [16]. It is defined as:

Return on Investment

$$\text{ROI} \triangleq \frac{(\text{Investment Gain}) - (\text{Investment Cost})}{(\text{Investment Cost})} \tag{1}$$

The investment cost of test-deferred development is $\frac{x(x+1)}{2}$ (units of work); the investment cost of test-driven development is $2x$ (units of work). Therefore, the investment gain of switching to test-driven development from test-deferred develoment is:

$$\left[\frac{x\,(x+1)}{2}\right] - (2x) = \frac{x\,(x+1) - 4x}{2} \quad = \frac{x\,(x-3)}{2}$$

Plugging this back into our formula for determining the ROI in terms of units of work, we get the following:

$$\begin{aligned}
\text{ROI} &\triangleq \frac{(\text{Investment Gain}) - (\text{Investment Cost})}{(\text{Investment Cost})} \\
&= \frac{\left[\frac{x(x-3)}{2}\right] - 2x}{2x} = \frac{\left[\frac{x(x-3)-4x}{2}\right]}{2x} \\
&= \frac{\left[\frac{x(x-7)}{2}\right]}{2x} = \frac{x\,(x-7)}{4x} \\
&= \frac{x-7}{4}
\end{aligned}$$

Ultimately, we are frugal and want to save money; therefore we want a positive ROI. To determine how many units of work are required to generate a positive return on investment, we may do the following:

$$\text{ROI} > 0 \implies \frac{x-7}{4} > 0 \implies x - 7 > 0 \implies x > 7$$

ROI in terms of Units of Work

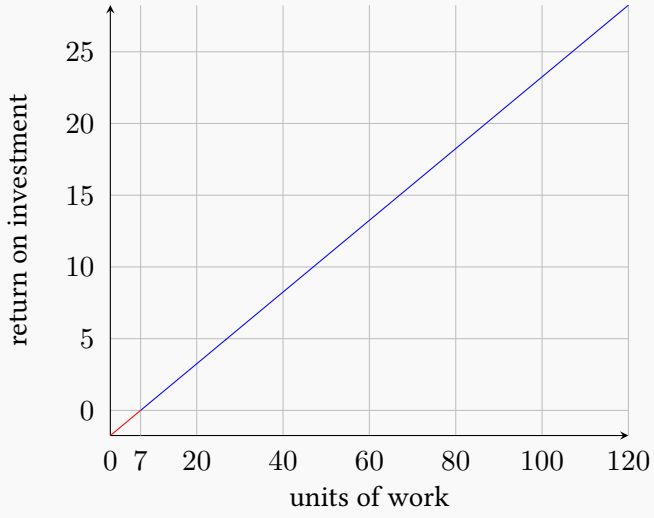

Figure 2: ROI in terms of Units of Work

Therefore, for any number of units of work greater than 7, we will save units of work (and therefore money) by utilizing test-driven development instead of test-deferred development. To estimate how much money will be saved, we may use the following formula:

Monetary Return on Investment

$$\text{ROI}_{\$}\ (x \text{ units of work}) = \underbrace{\left(\frac{x-7}{4}\right)}_{\text{units}} \mathbb{E}\big[\text{hours per unit}\big]\, \mathbb{E}\big[\$ \text{ per hour}\big] \tag{2}$$

To give this a concrete value, let's say an entry level engineer is tasked with implementing some system. This engineer makes $100,000 / year working 40 hours / week (on average). There are 52 weeks per year (on average), therefore the engineer costs approximately $48 / hour.

Let's say the system will take six months to complete. There are four weeks per month and (s)he works five days per week. Therefore, (s)he will spend a total of 120 days implementing the system.

Let's say (s)he does a good job estimating the number of units required to complete the system such that (s)he is able to complete two units per day. Then, the system would require 240 units of work at four hours of work per unit. Since $240 > 7$, the engineer ***would be saving the company approximately $11,184*** (conservatively, using test-driven development). If (s)he works on a team of 10 engineers who are all working on such projects, the company ***would be able to double its engineers*** in just over three years using the savings, alone:

Proof. Using the formula for continuously-compounded interest, it will be shown that the engineering team may double its size in three years. The formula is given by, $A = Pe^{rt}$, where $A \coloneqq$ Amount after compounding interest, $P \coloneqq$ Principal amount, $e \coloneqq$ Euler's constant, $r \coloneqq$ rate of interest, and $t \coloneqq$ time in years. Here, $P = 10$, $r = 2\frac{11{,}184}{100{,}000} \approx 0.22368$, and $A = 20$; we want to solve for t:

$$A = Pe^{rt} \iff \frac{A}{P} = e^{rt} \iff \ln\frac{A}{P} = rt \iff \frac{1}{r}\ln\frac{A}{P} = t$$

Now, we may solve for t directly, which will tell us the number of years required to double the team's size:

$$t = \frac{1}{\left(2\frac{11{,}184}{100{,}000}\right)}\ln\frac{20}{10} \approx 3.1$$

Therefore, in just over three years the team would be able to double its size using test-driven development. □

Engineering team size after three years using TDD.

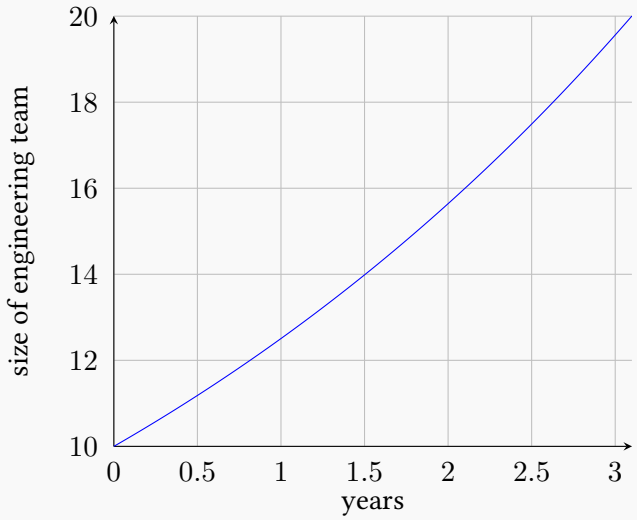

Figure 3: Engineering team size after three years using TDD.

Taking this further, let's say the company is not interested in doubling its engineers; rather, it is interested in doubling its throughput. Since the engineer is able to complete two units of work per day, the number of days required to develop x features using test-driven development is given by $2x/2 = x$ and the number of days required to develop the same number of features using test-deferred development is given by $\frac{x(x+1)}{2}/2 = \frac{x(x+1)}{4}$.

We know the engineer will be working for six months, or 120 business days. To determine how many features (s)he will be able to implement in the given time, we simply equate each expression to 120 and solve for the number of features:

Test-driven features in six months

$$x = 120 \qquad \triangleright \text{ trivial}$$

Test-deferred features in six months

$$\begin{aligned} \frac{x\,(x+1)}{4} &= 120 \\ x\,(x+1) &= 480 \\ x^2 + x - 480 &= 0 \\ x &\in \{-22.4, 21.4\} && \triangleright \text{ quadratic formula} \\ x &= 21.4 && \triangleright\ x > 0 \implies x \neq -22.4 \end{aligned}$$

$$x \approx 21 \qquad \triangleright\, x > 21 \in \mathbb{Z}^+ \implies \text{time} > 6 \text{ months}$$

Using test-deferred development, the engineer will only be able to implement 21 features. Using test-driven development, the engineer will be able to implement 120 features. Therefore, the engineer would be able to implement $5.7\times$ the number of features using test-driven development, which far surpasses the company's goal of $2\times$.

Test-driven versus test-deferred features in six months.

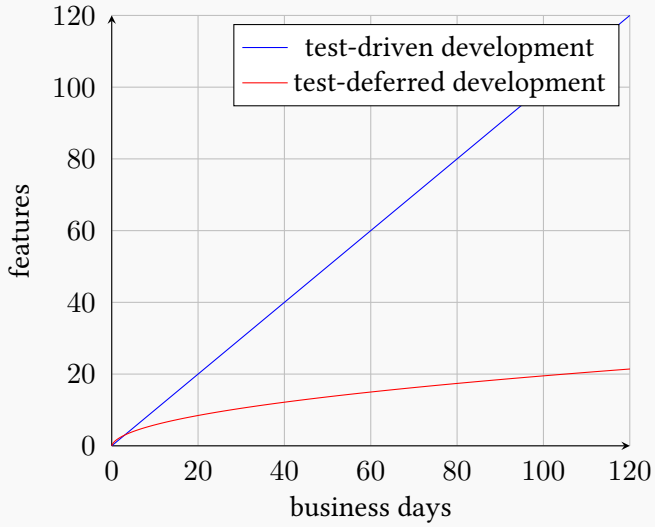

Figure 4: Test-driven versus test-deferred features in six months.

3. What should be tested?

Regarding unit testing, pretty much anything more than the most basic setter/getter methods should be tested. Debatably, even those should be tested in case more complex logic is assigned to them later, such as constraining values to the setter or inferring values from the getter.

Regarding integration testing, complex interactions among units should be tested, as well as every known use case for its customers. It is important to ensure your application works correctly from its customers' perspectives.

Regarding regression testing, every (confirmed) bug found outside automated testing should be fixed and have a regression test written for it to ensure it never resurfaces. Like stated before, these may be of either the unit testing or integration testing variety.

4. How should testing be done?

4.1. Test-driven development

Test-driven development is an iterative paradigm in which tests are written before the source code, from the inside-out perspective of the developer. [1] Test-driven development utilizes the following sequence of operations:

1. Add a test for the new unit, before writing its source code.
2. Run all the tests and ensure the new one fails (the source code hasn't been written yet).
3. Write the minimum code required to satisfy the new test.
4. Run all the tests again and verify the new test passes.
5. Refactor the code as necessary.
6. Repeat.

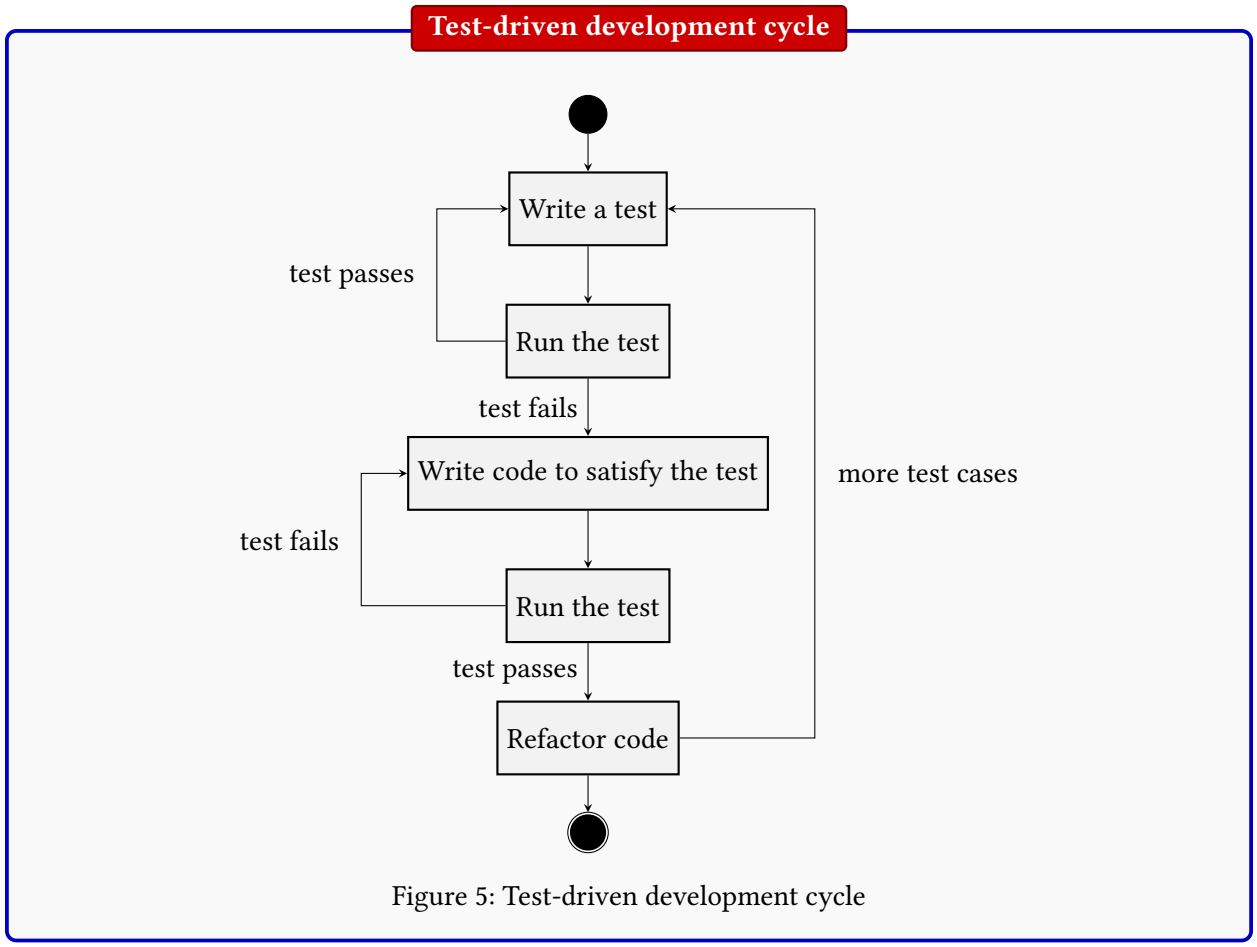

Figure 5: Test-driven development cycle

4.2. Behavior-driven development

Behavior-driven development (BDD) is an iterative paradigm derived from test-driven development in which tests are also written before the source code, but from the outside-in perspective of the consumer. [1] BDD is largely facilitated through the use of a simple domain-specific language (DSL) using natural language constructs (e.g. English-like sentences) that can express the behavior and the expected outcomes. [26]

Although, quite verbose for the majority of unit testing, behavior-driven development is a natural fit for integration testing, especially when a non-technical product manager is involved in the development process. When a BDD test fails, its nested context is printed in the failure message which gives a handy description of the problem, like, *"When I click the 'Add User' button and when I enter a name into the dialog then a new user with that name should be added to the 'Users' list."*

The product manager defines the use cases for the application. The engineers take those use cases and create tests to satisfy them. The tests are created using a BDD framework and test-driven development. Once the tests are run and verified to fail, the engineers implement the features, run the tests again, and verify they succeed.

bdd_example.js

```
// a fictitious BDD framework
bdd.run(bdd.when("I click the 'Add User' button", function() {
    return bdd.when("I enter a name into the dialog", function() {
        return bdd.then("a new user with that name should be added to"+
                        "the 'Users' list.", function(async_callback) {
            var user_name = "John Doe";
            var $users_list = $(".users.list");  // JQuery-like selector
            if ($users_list.find(":contains("+user_name+")")) {
                var message = "User, "+user_name+", already exists!";
                throw new BDDAssertionError(message);
            }
            var $user_name_dialog = $(".user-name.dialog");
            $user_name_dialog.open(function() {
                $user_name_dialog.find(":text.user_name").val(user_name);
                $user_name_dialog.find(":button.save").click()
            });
            $user_name_dialog.close(function() {
                if ($users_list.find(":contains("+user_name+")")) {
                    async_callback.succeed()
                } else {
                    async_callback.fail()
                }
            });
            $("button.add-user").click();
            return async_callback;
        });
    });
}));
```

4.3. Fluent assertions

Fluent assertions improve assertion readability, provide truly helpful error messages, and facilitate chaining assertions together. [4]

inarticulate_example.py

```
1  assert 1 % 2 == 0
```

When this script is executed, a mostly-useless error message is printed:

Terminal

```
$ python inarticulate_example.py
Traceback (most recent call last):
  File "inarticulate_example.py", line 1, in <module>
    assert 1 % 2 == 0
AssertionError
```

In this example, the failure is easy to debug, but assertion errors can be difficult to understand. Rewriting the example in a fluent sense, we get something like:

fluent_example.py

```
1  assert_that(1).is_even()
```

Executing this, we might get an exception message like:

Terminal

```
$ python fluent_example.py
Traceback (most recent call last):
  File "fluent_example.py", line 1, in <module>
    assert_that(1).is_even()
AssertionError: Expected "1" to be even.
```

A more realistic example might look like the following:

realistic_fluent_example.py

```
1  assert_that(john_smith).has_first_name("John")\
2      .has_last_name("Smith")\
3      .is_married_to(jane_smith)\
4      .has(4).children()
```

4.4. Example-based testing

Example-based testing verifies the behavior of components based on explicit data points provided by the developer. [15] This contrasts property-based testing, in which a random generator provides the data points.

example_based_testing_demo.py

```python
import pytest

from test_utils import assert_that

from math_functions import square

def x_generator():
    for x in range(10):
        yield x

@pytest.mark.parametrize("x", x_generator())
def test_square(x):
    assert_that(square(x)).is_equal_to(x * x)
```

4.5. Property-based testing

Property-based testing verifies the behavior of components based on randomly-generated data points provided by the property-testing framework; it is a special case of fuzz testing. [15] Some of the reasons property-based testing should be preferred over example-based testing are that it (1) greatly simplifies the data generation process, (2) reduces the testing verbosity, and (3) easily generates nontrivial edge cases.

property_based_testing_demo.py

```python
import hypothesis.strategies as st
from hypothesis import given

from test_utils import assert_that

from math_functions import square

@given(st.floats())
def test_square(x):
    assert_that(square(x)).is_equal_to(x * x)
```

When we run this example, we find there is actually a bug we did not catch in the corresponding example-based test.

4.6. Test generation

Test generation uses function specifications on parameter and return value constraints to automatically generate property-based tests. [3] As the definition implies, a specification must be provided for each of the functions. Tests are then generated from the specifications and fed randomly-generated data points based on their parameter constraints. Their output is finally validated using predicates provided by their specification. Please reference the appendix for an example using the state-of-the-art library, clojure.spec.

A few bonus features clojure.spec gives you beyond test generation include automatic function documentation and parameter validation. For example, given the function from the appendix, `drop-common-prefix`:

Terminal

```
$ clj
user=> (require '[clojure.spec.alpha :as s]
                '[clojure.spec.test.alpha :as stest]
                '[distance :as d])
nil
user=> (doc d/drop-common-prefix)
-------------------------
distance/drop-common-prefix
([v w])
  Strips the common prefix from two strings.
Spec
  args: (coll-of :distance/term :type vector? :count 2)
  ret: (tuple string? char? string? string? char? string?)
  fn: (fn [{[v w] :args, [v' a s w' b t] :ret}] (and (= v' (str a s))
                                                     (.endsWith v v')
                                                     (= w' (str b t))
                                                     (.endsWith w w')))
nil
user=> (stest/instrument `d/drop-common-prefix)
[distance/drop-common-prefix]
user=> (d/drop-common-prefix "foo" "foobar")
["o" \o "" "obar" \o "bar"]
user=> (d/drop-common-prefix "foo" nil)
ExceptionInfo Call to #'distance/drop-common-prefix did not conform to spec:
In: [1] val: nil fails spec: :distance/term at: [:args] predicate: string?
  clojure.core/ex-info (core.clj:4739)
user=> (d/drop-common-prefix "123" 12345)
ExceptionInfo Call to #'distance/drop-common-prefix did not conform to spec:
In: [1] val: 12345 fails spec: :distance/term at: [:args] predicate: string?
  clojure.core/ex-info (core.clj:4739)
```

4.7. Formal methods

Formal methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. [17] They use mathematical logic to rigorously deduce the correctness of a system's specifications. A downside to formal methods is that rigorously proving the correctness of complex systems is infeasible. Some heuristics employed to work around this shortcoming include:

- Apply formal methods to requirements and high-level designs where most of the details are abstracted away.
- Apply formal methods to only the most critical components.
- Analyze models of software and hardware where variables are discretized and ranges drastically reduced.

- Analyze system models in a hierarchical manner that enables "divide and conquer".
- Automate as much of the verification as possible.

One popular language for formal methods is TLA+. TLA+ is a formal methods specification language based on the idea that the best way to describe things formally is with simple mathematics, and that a specification language should contain as little as possible beyond what is needed to write simple mathematics precisely; it is especially well suited for writing high-level specifications of concurrent and distributed systems. [10] Please see the appendix for an example showing how to detect race conditions with TLA+. Please read, "How Amazon Web Services Uses Formal Methods", for a real-life success story about how AWS uses TLA+ to find bugs.

4.8. Dependency injection

Dependency injection is a technique whereby one object supplies the dependencies of another object; it is a special form of inversion of control (IoC). [27] Although, not specifically a testing technique, dependency injection is such an effective way to ease the testing process that it was worth mentioning here.

With dependency injection, one provides constructor parameters or utilizes setters to allow external processes to control the fields used by the class, thereby inverting the control over the fields from the class to whomever is constructing it. This is a tremendously important paradigm because it allows one to provide mocked objects in place of the production ones to ensure the class is utilizing them as intended. Please reference the appendix for an example.

5. How should testing be evaluated?

There are many metrics one might use to evaluate how well-tested one's application is. The following are a few that are particularly useful for automated testing.

5.1. Line Coverage

Line coverage is a variation of test coverage that measures which lines of source code were tested. Many code coverage tools allow one to enable/disable coverage reporting for various functions, which is useful for getters/setters that consist of one-liner operations; testing such functions has little benefit and decreases the readability of one's tests.

Line Coverage

$$\text{line coverage} \triangleq \frac{\text{\# lines tested}}{\text{\# lines in application}} \tag{3}$$

5.2. Branch Coverage

Branch coverage is a variation of test coverage that measures which branches of execution were tested for each line. This is similar to line coverage, but differs in that it ensures every condition in every conditional expression is evaluated to true at least once.

Branch Coverage

$$\text{branch coverage} \triangleq \frac{\text{\# of control structure branches tested}}{\text{\# of total control structure branches}} \tag{4}$$

For example, let's say you have a function with condition that may evaluate to true in two different ways. You could write a unit test that caused its condition to evaluate to true once, and get 100% line coverage, but it would only have 50% branch coverage:

predicates.py

```python
def is_one_or_two(x):
    if x == 1 or x == 2:
        return True
    return False
```

test_predicates.py

```python
from predicates import is_one_or_two

def test_is_one_or_two():
    assert is_one_or_two(1)     # 33% branch coverage; 67% line coverage
    assert is_one_or_two(2)     # 67% branch coverage; 67% line coverage
    assert not is_one_or_two(3) # 100% branch coverage; 100% line coverage
```

5.3. Automation Progress

Automation progress describes how well one is progressing toward one's goal of automating the testing of one's application. Something to note is that the goal is to automate every test that is automatable – there may still be some tests that must or should be verified manually. [5]

Percent Automatable

$$\text{Percent Automatable} \triangleq \frac{\text{\# test cases implemented}}{\text{\# test cases automatable}} \tag{5}$$

5.4. Percent of Automated Testing Test Coverage

This metric describes the ratio of automated tests over all tests performed on the system (both automated and manual). [5] A well-tested application should optimally have a PTC (Percent of Automated Testing Test Coverage) near 1:

Percent Automated Testing Test Coverage

$$\text{PTC} \triangleq \frac{\text{automation coverage}}{\text{total coverage}} \tag{6}$$

5.5. Defect Trend Analysis

DTA (Defect Trend Analysis) determines the trend of defects found: is it improving as the testing phase is winding down or is the trend worsening? [5]

Defect Trend Analysis

$$\text{DTA} \triangleq \frac{\text{\# known defects}}{\text{\# test procedures executed}} \tag{7}$$

5.6. Defect Removal Efficiency

DRE (Defect Removal Efficience) specifies the ratio of defects found and fixed during testing over the total number of defects found. [5] It indicates how successful the automated tests were at identifying the defects of the application. In another sense, it is an indirect measurement of the quality of the product. [5]

Defect Removal Efficiency

$$\text{DRE} \triangleq \frac{\text{\# defects found during testing}}{(\text{\# defects found during testing}) + (\text{\# defects found after delivery})} \tag{8}$$

6. Conclusion

Integration testing is paramount to ensuring a good customer experience. When integration tests break it means the system is not behaving as the customers expect, which gives you the what. Unit testing will give you the where, why, and extent of breakage. The when, how, and who would be determined by examining development logs. These form the who, what, when, where, why, how, and to what extent of software testing. Regression testing will ensure the application never breaks in the same manner again, and utilizes unit testing and integration testing.

Properly testing a system has the short-term tradeoff that it increases development time, but that tradeoff is offset in the long-term by faster future development times and greatly increased reliability. When adding new features to or refactoring a well-tested system, it is quick and easy to check whether the changes have broken it and where the faults are. That's not to say you'll catch every edge case with automated testing – automated tests are only as good as their authors make them – but libraries and tools make extensively testing your system easy and reproducible.

This is in comparison to making changes to a poorly-tested system where you have to smoke test it for prolonged periods of time (hours; days; weeks; longer) to make sure you haven't broken anything. If you

have then you try to fix it and restart your smoke testing session. By the time you've finished you will still likely have missed several test cases because it is difficult to remember and exhaustively test every one in a complex system.

It may take half the time to reach the first milestone without writing automated tests, but there rapidly comes a point at which the time required to develop a test-deferred system greatly surpasses that required to develop a test-driven one. If you are developing a short-lived prototype that you are confident will never go into production, then you may complete the project in half the time by not writing tests. However, from my experience, 10/10 prototypes developed for production systems go into production. If you skimp out on testing and design choices during the prototyping phase then the production system will be handicapped until its technical debt is payed. For production systems, it's best to push back deadlines to enforce proper testing standards – doing so will save the company substantial amounts of money in the long-term.

Glossary

Notation	Description	Page List
acceptance testing	verifies that the software meets the needs of its consumers; the verification is performed by the consumers. [21]	1–3
AWS	Amazon Web Services	1, 16
BDD	behavior-driven development	1, 12
behavior-driven development	an iterative paradigm derived from test-driven development in which tests are also written before the source code, but from the outside-in perspective of the consumer. [1]	1, 12
branch coverage	a variation of test coverage that measures which branches of execution were tested for each line.	1, 17
bug	a consequence/outcome of a coding fault. [25]	1
defect	a variation or deviation from the original business requirements. [25]	1
dependency injection	a technique whereby one object supplies the dependencies of another object; it is a special form of inversion of control (IoC). [27]	1, 2, 16, 24, 25
DI	dependency injection	1
example-based testing	verifies the behavior of components based on explicit data points provided by the developer. [15]	1, 13, 14
fluent assertions	improve assertion readability, provide truly helpful error messages, and facilitate chaining assertions together. [4]	1, 13
formal methods	mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. [17]	1, 15, 16
fuzz testing	verifies the behavior of a system with unexpected or random inputs. [20]	1, 4
health checks	report the status of a running system (or the lack thereof).	1, 2, 4
integration testing	verifies that different modules or services used by your application work well together. [14]	1–3, 10, 12, 18

Notation	Description	Page List
inversion of control	a design principle in which custom-written portions of a computer program receive the flow of control for a generic framework. [28]	1
IoC	inversion of control	1
line coverage	a variation of test coverage that measures which lines of source code were tested.	1, 16, 17
mocking	creates an object which mirrors the API of another. [11]	1
penetration testing	analyzes how secure software and its environments (hardware, operating system, newtork, etc.) are when subject to attack by an external or internal intruder. [20]	1, 4
performance testing	determines how well the system operates under a heavy load and/or the maximum load it can handle. [21]	1–3
PlusCal	an algorithm language that …is translated into a TLA+ specification, to which the TLA+ tools can be applied. [10]	1, 29
property-based testing	verifies the behavior of components based on randomly-generated data points provided by the property-testing framework; it is a special case of fuzz testing. [15]	1, 2, 13, 14, 27
regression testing	seeks to uncover software errors after changes to the program have been made (e.g. bug fixes, new functionality, and refactoring). [22]	1–3, 10, 18
return on investment	a performance measure used to evaluate the efficiency of an investment or to compare the efficiency of a number of different investments. [16]	1, 6
ROI	return on investment	1, 6, 7
sanity testing	ensures a system or service is ready to execute.	1, 2, 4
security testing	verifies the system is safe from internal and external threats. [21]	1, 2, 4
smoke testing	manually verifies that no defects exist in the system (preferably according to a test plan). [21]	1–3

Notation	Description	Page List
software testing	an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test. [2]	1, 2
spying	observes the invocations, input parameters, and responses of object methods. [11]	1
stubbing	overrides a method on an object to return a canned response. [11]	1
system testing	Ensures that your application works as expected under different environments (e.g. operating systems or hardware). [31]	1–3
TDD	test-driven development	1, 2, 9
technical debt	a metaphor that describes the gap between the current state of the system and the ideal state of the system, usually in situations where a compromise is made during development to meet demands in one dimension, such as lead time, by sacrificing work in another dimension, such as architecture or automated testing. [30]	1, 4, 19
test coverage	measures the amount of testing performed by a suite of tests. [19]	1
test estimation	a management activity which approximates how long a task would take to complete. [18]	1
test generation	uses function specifications on parameter and return value constraints to automatically generate property-based tests. [3]	1, 2, 14, 15, 28
test plan	a detailed document that outlines the test strategy, testing objectives, resources (manpower, software, hardware) required for testing, test schedule, test estimation and test deliverables. [6]	1
test strategy	a plan for defining the testing approach that answers questions like, “What do you want to get done?”, and, “How are you going to accomplish it?”, which further guide the team to defining software testing and test coverage requirements. [7]	1
test-driven development	an iterative paradigm in which tests are written before the source code, from the inside-out perspective of the developer. [1]	1, 6, 8–12

Notation	Description	Page List
TLA Toolbox	an IDE (integrated development environment) for the TLA+ tools. [9]	1, 30
TLA+	a formal methods specification language based on the idea that the best way to describe things formally is with simple mathematics, and that a specification language should contain as little as possible beyond what is needed to write simple mathematics precisely; it is especially well suited for writing high-level specifications of concurrent and distributed systems. [10]	1, 2, 16, 29–31
unit testing	ensures individual methods and functions (the smallest units of design [23]) of the classes, components, and modules used by your software work as expected [14].	1–3, 10, 12, 18
validation testing	verifies that a system or service is running as expected.	1, 2, 4
VIP	Virtual IP	1, 4
vulnerability testing	involves identifying or exposing the software, hardware and network to vulnerabilities that can be exploited by attackers. [20]	1, 4

Appendices

A. dependency injection

A.1. Without dependency injection

accessors_without_di.py

```
from db import Connection  # fictitious database connection class

class UserAccessor:
    def __init__(self):
        self.db = new Connection()
    def find_by_name(self, first_name, last_name):
        return self.db.select_from("users").where(
            first_name=first_name,
            last_name=last_name)

class RoleAccessor:
    def __init__(self):
        self.user_accessor = UserAccessor()
        self.db = new Connection()  # duplicated connection
    def find_by_user_name(self, first_name, last_name):
        # not very efficient -- this is for demonstration only
        user = self.user_accessor.find_by_name(first_name, last_name)
        return self.db.select_from("roles")\
            .where(user_id=user.id)\
            .order_by("id")
```

test_accessors_without_di.py

```
from accessors_without_di import UserAccessor


def test_user_accessor_find_by_name():
    accessor = UserAccessor()
    with accessor.db.transactioon():
        try:
            query = """
                INSERT INTO users (id, first_name, last_name)
                VALUES ({}, {}, {})
"""
            accessor.db.execute(query, 1, "John", "Smith")
            user = accessor.find_by_id(1)
            assert user is not None
            assert user.id == 1
            assert user.first_name == "John"
            assert user.last_name == "Smith"
        finally:
            accessor.db.rollback()

```

```
def test_role_accessor_find_by_user_name():
    accessor = RoleAccessor()
    with accessor.db.transactioon():
        try:
            query = """
                INSERT INTO users (id, first_name, last_name)
                VALUES ({}, {}, {})
"""
            accessor.db.execute(query, 1, "John", "Smith")
            query = """
                INSERT INTO roles (id, name, user_id)
                VALUES ({}, {}, {})
"""
            accessor.db.execute(query, 1, "admin", 1)
            accessor.db.execute(query, 2, "user", 1)
            roles = accessor.find_by_user_name("John", "Smith")
            assert roles is not None
            assert len(roles) == 2
            assert roles[0].id == 1
            assert roles[0].name == "admin"
            assert roles[0].user_id == 1
            assert roles[1].id == 2
            assert roles[1].name == "user"
            assert roles[1].user_id == 1
        finally:
            accessor.db.rollback()
```

A.2. With dependency injection

accessors_with_di.py

```
from db import Connection  # fictitious database connection class

class UserAccessor:
    def __init__(self, db):
        self.db = db
    def find_by_name(self, first_name, last_name):
        return self.db.select_from("users").where(
            first_name=first_name,
            last_name=last_name)

class RoleAccessor:
    def __init__(self, db, user_accessor):
        self.db = db
        self.user_accessor = user_accessor
    def find_by_user_name(self, first_name, last_name):
        # not very efficient -- this is for demonstration only
        user = self.user_accessor.find_by_name(first_name, last_name)
        return self.db.select_from("roles")\
            .where(user_id=user.id)\
            .order_by("id")
```

test_accessors_with_di.py

```
from db import Connection, ConnectionTimeoutError

from accessors_with_di import UserAccessor, RoleAccessor

from testing_tools import mock, when

def build_user(id, first_name, last_name):
    user = mock(object)
    user.id = id
    user.first_name = first_name
    user.last_name = last_name
    return user

def build_role(id, name, user):
    role = mock(object)
    role.id = id
    role.name = name
    role.user_id = user.id
    return role

def mock_user_in_db(db, user):
    when(db).select_from("users")\
        .where(first_name=user.first_name,
               last_name=user.last_name)\
        .then_return(user)

def mock_user_roles_in_db(db, user, roles):
    when(db).select_from("roles")\
        .where(user_id=user.id)\
        .then_return(roles)

def test_user_accessor_find_by_name():
    expected_user = build_user(1, "John", "Smith")
    db = mock(Connection)
    mock_user_in_db(db, expected_user)
    accessor = UserAccessor(db)
    actual_user = accessor.find_by_name(
        user.first_name,
        user.last_name)
    assert expected_user == actual_user

def test_role_accessor_find_by_user_name():
    user = build_user(1, "John", "Smith")
    admin_role = build_role(1, "admin", user)
    user_role = build_role(1, "user", user)
    expected_roles = [admin_role, user_role]
    db = mock(Connection)
    mock_user_in_db(db, user)
    mock_user_roles_in_db(db, user, roles)
    user_accessor = UserAccessor(db)
    role_accessor = RoleAccessor(db, user_accessor)
    actual_roles = role_accessor.find_by_user_name(
```

```
        user.first_name,
        user.last_name)
    assert expected_roles == actual_roles

# Not so easy to do without mocking
def test_user_accessor_handles_connection_timeout_errors_gracefully():
    expected_user = build_user(1, "John", "Smith")
    db = mock(Connection)
    when(db).select_from("users")\
        .where(first_name=expected_user.first_name,
               last_name=expected_user.last_name)\
        .then_raise(ConnectionTimeoutError)\  # timeout once ...
        .then_return(expected_user)           # ... then succeed
    accessor = UserAccessor(db)
    actual_user = accessor.find_by_name(
        user.first_name,
        user.last_name)
    assert expected_user == actual_user
```

B. Property-based testing

property_based_testing_demo.py

```
import hypothesis.strategies as st
from hypothesis import given

from test_utils import assert_that

from math_functions import square

@given(st.floats())
def test_square(x):
    assert_that(square(x)).is_equal_to(x * x)
```

Terminal

```
$ py.test --rootdir . property_based_testing_demo.py::test_square
=============================================================================
test session starts
=============================================================================
platform darwin -- Python 3.7.0, pytest-3.9.3, py-1.7.0, pluggy-0.7.1
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/path/to/.hypothesis/examples')
rootdir: /path/to, inifile:
plugins: hypothesis-3.82.1
collected 1 item

property_based_testing_demo.py F
                                                                    [100%]
```

```
============================== FAILURES ==============================
_____________________________ test_square _____________________________

    @given(st.floats())
>   def test_square(x):

property_based_testing_demo.py:9:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

x = nan

    @given(st.floats())
    def test_square(x):
>       assert_that(square(x)).is_equal_to(x * x)
E       assert_that(square(nan)).is_equal_to(nan * nan)

property_based_testing_demo.py:10:
    AssertionError: Expected 'nan' to be equal to 'nan'
----------------------------------------------------------------------
Hypothesis
----------------------------------------------------------------------
Falsifying example: test_square(x=nan)
====================== 1 failed in 0.31 seconds ======================
```

C. Test generation

distance.clj

```clojure
(ns distance
  (:require [clojure.spec.alpha :as spec]
            [clojure.spec.gen.alpha :as gen]))

(defn drop-common-prefix
  "Strips the common prefix from two strings."
  ([^String v, ^String w]
   (loop [^String v v, a (.charAt v 0), ^String s (.substring v 1),
          ^String w w, b (.charAt w 0), ^String t (.substring w 1)]
     (if (and (= a b)
              (not (.isEmpty s))
              (not (.isEmpty t)))
       (recur s (.charAt s 0) (.substring s 1)
              t (.charAt t 0) (.substring t 1))
       [v a s, w b t]))))

;;; Specs

(spec/def ::term string?)

(def prefix-pair-gen
  (gen/fmap
```

```
    (fn [[u v w]]
      [(str u v) (str u w)])
    (spec/gen
     (spec/and (spec/coll-of ::term :type vector? :count 3)
               (fn [[u v w]]
                 (and (not-empty v)
                      (not-empty w)))))))

(spec/def ::prefix-pair
  (spec/with-gen
    (spec/coll-of ::term :type vector? :count 2)
    (constantly prefix-pair-gen)))

(spec/fdef drop-common-prefix
           :args ::prefix-pair
           :ret (spec/tuple string? char? string?, string? char? string?)
           :fn (fn [{[v w] :args, [v' a s, w' b t] :ret}]
                 (and (= v' (str a s))
                      (.endsWith v v')
                      (= w' (str b t))
                      (.endsWith w w'))))
```

distance_test.clj

```
(ns distance-test
  (:require [distance :as sut]
            [clojure.spec.test.alpha :as stest]
            [clojure.test :as t]))

(defmacro test-conformity
  "Tests the conformity of all available specs at the point of invocation."
  ([]
   `(test-conformity (stest/checkable-syms)))
  ([sym-expr]
   (cons `do
         (for [sym (eval sym-expr)]
           `(t/deftest ~(symbol (str "test-conformity-of-" (name sym)))
              (let [spec# (symbol ~(str sym))
                    results# (stest/check spec#)
                    summary# (stest/summarize-results results#)
                    {total# :total, check-passed# :check-passed} summary#]
                (t/is (= total# check-passed#))))))))

(test-conformity)
```

D. Race condition detection with TLA+

We'll begin by defining a simple spec in the PlusCal language that manipulates account balances in parallel: [24]

Transfer.tla

```
---- MODULE Transfer ----
EXTENDS Naturals, TLC

(* --algorithm transfer
variables alice_account = 10, bob_account = 10,
          account_total = alice_account + bob_account;

process Transfer \in 1..2
  variable money \in 1..20;
begin
Transfer:
  if alice_account >= money then
    A: alice_account := alice_account - money;
       bob_account := bob_account + money;
end if;
C: assert alice_account >= 0;
end process

end algorithm *)

MoneyNotNegative == money >= 0
MoneyInvariant == alice_account + bob_account = account_total
====
```

Next, we need to translate it to a TLA+ specification. Using the TLA Toolbox, this is as simple as opening the "File" menu and selecting, "Translate PlusCal Algorithm". Once translated, we will have the following specification:

Transfer.tla

```
\* BEGIN TRANSLATION
\* Label Transfer of process Transfer at line 12 col 3 changed to Transfer_
VARIABLES alice_account, bob_account, account_total, pc, money

vars == << alice_account, bob_account, account_total, pc, money >>

ProcSet == (1..2)

Init == (* Global variables *)
        /\ alice_account = 10
        /\ bob_account = 10
        /\ account_total = alice_account + bob_account
        (* Process Transfer *)
        /\ money \in [1..2 -> 1..20]
        /\ pc = [self \in ProcSet |-> "Transfer_"]

Transfer_(self) == /\ pc[self] = "Transfer_"
                   /\ IF alice_account >= money[self]
                         THEN /\ pc' = [pc EXCEPT ![self] = "A"]
                         ELSE /\ pc' = [pc EXCEPT ![self] = "C"]
                   /\ UNCHANGED << alice_account, bob_account, account_total,
```

```
                                    money >>

A(self) == /\ pc[self] = "A"
           /\ alice_account' = alice_account - money[self]
           /\ bob_account' = bob_account + money[self]
           /\ pc' = [pc EXCEPT ![self] = "C"]
           /\ UNCHANGED << account_total, money >>

C(self) == /\ pc[self] = "C"
           /\ Assert(alice_account >= 0,
                     "Failure of assertion at line 16, column 4.")
           /\ pc' = [pc EXCEPT ![self] = "Done"]
           /\ UNCHANGED << alice_account, bob_account, account_total, money >>

Transfer(self) == Transfer_(self) \/ A(self) \/ C(self)

Next == (\E self \in 1..2: Transfer(self))
           \/ (* Disjunct to prevent deadlock on termination *)
              ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION
```

Running this through the TLA+ checker, we see that there exists a race condition in which Alice's account has two quantities deducted from it simultaneously – whose sum is greater than Alice's balance – resulting in a negative balance. There is an invariant that Alice's balance must always be non-negative, therefore the test fails:

Terminal

```
@!@!@STARTMSG 2132:0 @!@!@
The first argument of Assert evaluated to FALSE; the second argument was:
"Failure of assertion at line 16, column 4."
@!@!@ENDMSG 2132 @!@!@
@!@!@ENDMSG 2154 @!@!@
@!@!@ENDMSG 1000 @!@!@
@!@!@STARTMSG 2121:1 @!@!@
The behavior up to this point is:
@!@!@ENDMSG 2121 @!@!@
@!@!@STARTMSG 2217:4 @!@!@
1: <Initial predicate>
/\ bob_account = 10
/\ money = <<3, 10>>
/\ alice_account = 10
/\ pc = <<"Transfer_", "Transfer_">>
/\ account_total = 20

@!@!@ENDMSG 2217 @!@!@
```

```
@!@!@STARTMSG 2217:4 @!@!@
2: <Transfer_ line 36, col 20 to line 41, col 43 of module Transfer>
/\ bob_account = 10
/\ money = <<3, 10>>
/\ alice_account = 10
/\ pc = <<"A", "Transfer_">>
/\ account_total = 20

@!@!@ENDMSG 2217 @!@!@
@!@!@STARTMSG 2217:4 @!@!@
3: <Transfer_ line 36, col 20 to line 41, col 43 of module Transfer>
/\ bob_account = 10
/\ money = <<3, 10>>
/\ alice_account = 10
/\ pc = <<"A", "A">>
/\ account_total = 20

@!@!@ENDMSG 2217 @!@!@
@!@!@STARTMSG 2217:4 @!@!@
4: <A line 43, col 12 to line 47, col 50 of module Transfer>
/\ bob_account = 20
/\ money = <<3, 10>>
/\ alice_account = 0
/\ pc = <<"A", "C">>
/\ account_total = 20

@!@!@ENDMSG 2217 @!@!@
@!@!@STARTMSG 2217:4 @!@!@
5: <A line 43, col 12 to line 47, col 50 of module Transfer>
/\ bob_account = 23
/\ money = <<3, 10>>
/\ alice_account = -3
/\ pc = <<"C", "C">>
/\ account_total = 20

@!@!@ENDMSG 2217 @!@!@
@!@!@STARTMSG 2103:1 @!@!@
The error occurred when TLC was evaluating the nested
expressions at the following positions:
0. Line 49, column 12 to line 53, column 78 in Transfer
1. Line 49, column 15 to line 49, column 28 in Transfer
2. Line 50, column 15 to line 51, column 66 in Transfer
```

References

[1] Vinai Amble. *Quick Guide to Test-Driven Development (TDD) vs. BDD vs. ATDD*. [Online; accessed 19-October-2018]. 2018. URL: https://www.ca.com/en/blog-agile-requirements-designer/guide-to-test-driven-development-tdd-vs-bdd-vs-atdd.html.

[2] Cem Kaner, J.D., Ph.D. "Exploratory Testing". Quality Assurance Institute Worldwide Annual Software Testing Conference. 2006. URL: http://www.kaner.com/pdfs/ETatQAI.pdf.

[3] *clojure.spec – Rationale and Overview*. [Online; accessed 22-October-2018]. 2018. URL: https://clojure.org/about/spec.

[4] *Fluent assertions for java*. [Online; accessed 24-October-2018]. URL: http://joel-costigliola.github.io/assertj/.

[5] Thom Garrett. "Useful Automated Software Testing Metrics". In: ().

[6] *How to Create a Test Plan*. [Online; accessed 25-October-2018]. URL: https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html.

[7] *How to create Test Strategy Document*. [Online; accessed 26-October-2018]. URL: https://www.guru99.com/how-to-create-test-strategy-document.html.

[8] Shivprasad Koirala. *Dependency Injection (DI) vs. Inversion of Control (IOC)*. [Online; accessed 22-October-2018]. 2013. URL: https://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO.

[9] Lamport, Leslie B. *THE TLA TOOLBOX*. [Online; accessed 6-November-2018]. URL: https://lamport.azurewebsites.net/tla/toolbox.html.

[10] Lamport, Leslie B. *What is TLA?* [Online; accessed 6-November-2018]. URL: https://lamport.azurewebsites.net/tla/tla-intro.html.

[11] Hugh Mamill. *Mocks, Spies, Partial Mocks and Stubbing*. [Online; accessed 22-October-2018]. 2015. URL: https://www.javacodegeeks.com/2015/11/mocks-spies-partial-mocks-and-stubbing.html.

[12] Natasha and Paul and Penny. *Question: what is the sum of the first 100 whole numbers?? how am i supposed to work this out efficiently? thanks*. [Online; accessed 5-November-2018]. URL: http://mathcentral.uregina.ca/qq/database/qq.02.06/jo1.html.

[13] Chris Newcombe et al. "How Amazon Web Services Uses Formal Methods". In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: http://doi.acm.org/10.1145/2699417.

[14] Sten Pittet. *The different types of software testing*. [Online; accessed 19-October-2018]. 2018. URL: https://www.atlassian.com/continuous-delivery/different-types-of-software-testing.

[15] *Property-based testing*. [Online; accessed 24-October-2018]. URL: http://www.scalatest.org/user_guide/property_based_testing.

[16] *Return on Investment (ROI)*. [Online; accessed 31-October-2018]. URL: https://www.investopedia.com/terms/r/returnoninvestment.asp.

[17] Ricky Butler. *WHAT IS FORMAL METHODS?* [Online; accessed 6-November-2018]. URL: https://shemesh.larc.nasa.gov/fm/fm-what.html.

[18] *Software Test Estimation Techniques: Step By Step Guide.* [Online; accessed 26-October-2018]. URL: https://www.guru99.com/an-expert-view-on-test-estimation.html.

[19] *Test Coverage in Software Testing.* [Online; accessed 26-October-2018]. URL: https://www.guru99.com/test-coverage-in-software-testing.html.

[20] *Types of Software Testing – Complete List.* [Online; accessed 25-October-2018]. URL: https://www.testingexcellence.com/types-of-software-testing-complete-list/.

[21] *Types of Software Testing: Different Testing Types with Details.* [Online; accessed 19-October-2018]. 2018. URL: https://www.softwaretestinghelp.com/types-of-software-testing/.

[22] *Types of Software Testing: List of 100 Different Testing Types.* [Online; accessed 19-October-2018]. 2018. URL: https://www.guru99.com/types-of-software-testing.html.

[23] *Types of Software Testing.* [Online; accessed 19-October-2018]. 2018. URL: https://www.geeksforgeeks.org/types-software-testing/.

[24] Wayne, Hillel. *Learn TLA+.* [Online; accessed 6-November-2018]. URL: https://learntla.com/introduction/.

[25] *What is Software Bug? Learn Defect Management Process.* [Online; accessed 26-October-2018]. URL: https://www.guru99.com/the-unconventional-guide-to-defect-management.html.

[26] Wikipedia contributors. *Behavior-driven development.* [Online; accessed 2-November-2018]. 2018. URL: https://en.wikipedia.org/wiki/Behavior-driven_development.

[27] Wikipedia contributors. *Dependency injection.* [Online; accessed 30-October-2018]. 2018. URL: https://en.wikipedia.org/wiki/Dependency_injection.

[28] Wikipedia contributors. *Inversion of Control.* [Online; accessed 30-October-2018]. 2018. URL: https://en.wikipedia.org/wiki/Inversion_of_control.

[29] Wikipedia contributors. *Software testing.* [Online; accessed 19-October-2018]. 2018. URL: https://en.wikipedia.org/wiki/Software_testing.

[30] Kristian Wiklund et al. "Technical Debt in Test Automation". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* IEEE, 2012. DOI: 10.1109/ICST.2012.192.

[31] Rehman Zafar. *What is software testing? What are the different types of testing?* [Online; accessed 19-October-2018]. 2012. URL: https://www.codeproject.com/Tips/351122/What-is-software-testing-What-are-the-different-ty.