Scalable Web Systems

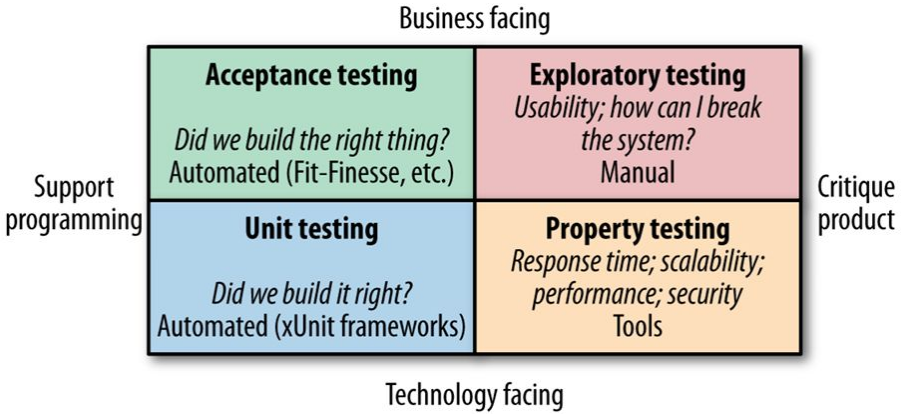
Testing

Overview

- Types of Tests
- Test Scope
- Implementing Service Tests
- Flaky and Brittle Tests
- Consumer-Driven Tests
- Testing After Production
- Cross-Functional Testing

Types of Tests

It is important to identify first what parts of a system a given test will target.

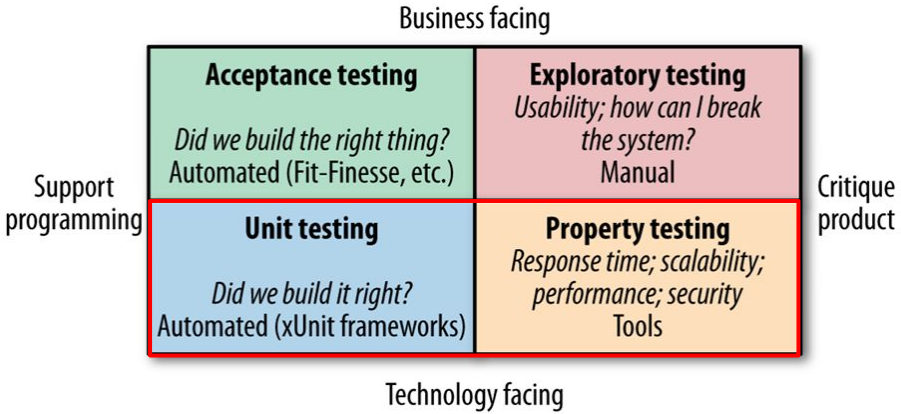


Types of Tests

It is important to identify first what parts of a system a given test will target.

There are tests that are **technology facing**.

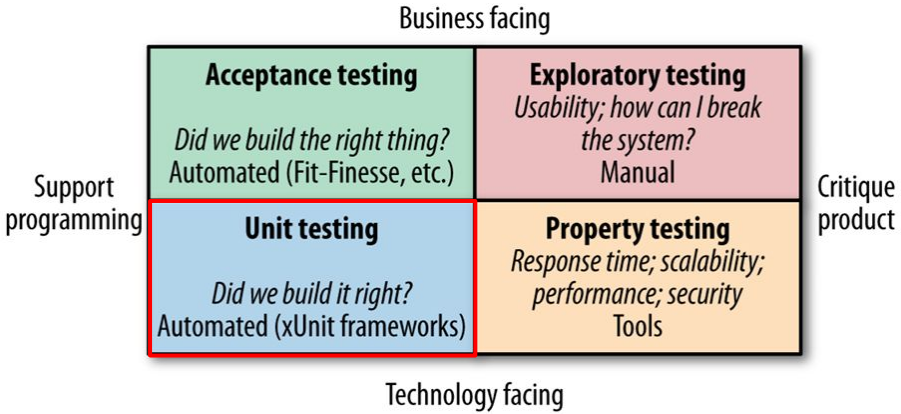
These types of tests aid the developers in creating the system in the first place.



Types of Tests

This includes **unit tests** - did you build it right?

These are primarily automated using a testing framework for a given technology.



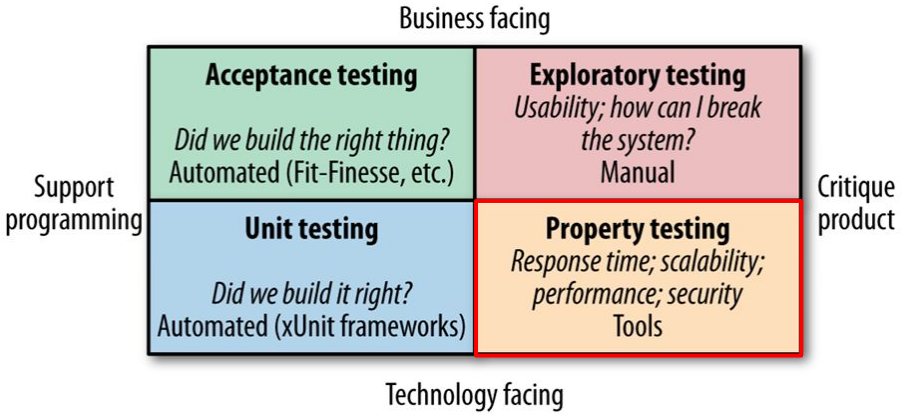
Types of Tests

This includes **unit tests** - did you build it right?

These are primarily automated using a testing framework for a given technology.

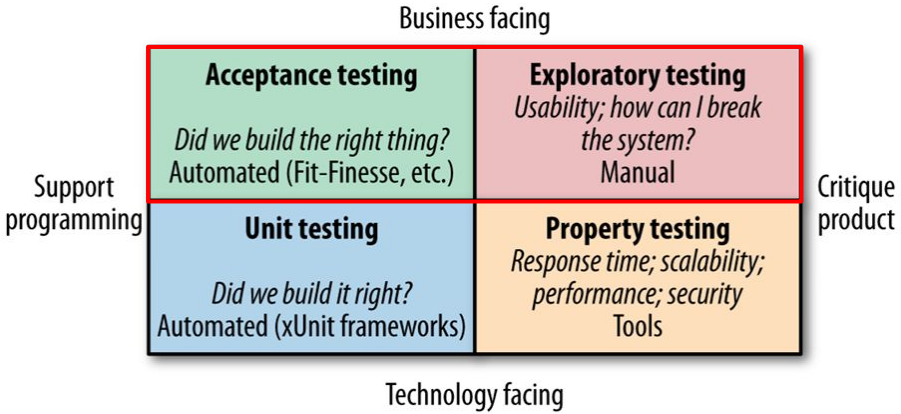
And **property testing** which focuses on:

1. Response time
2. Scalability
3. Performance
4. Security



Types of Tests

We also have **business facing** tests which focus on tests that help the non-technical stakeholders understand how your system works.



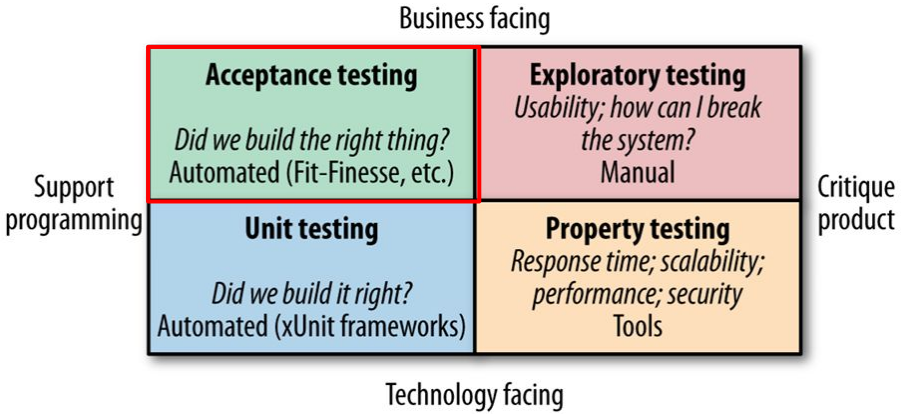
Types of Tests

We also have **business facing** tests which focus on tests that help the non-technical stakeholders understand how your system works.

This can be large-scoped end-to-end tests that automatically determine if your system is doing what is expected by an end user.

This is referred to as Acceptance Testing - a test conducted to determine if the requirements of a specification are met.

These are typically automated.



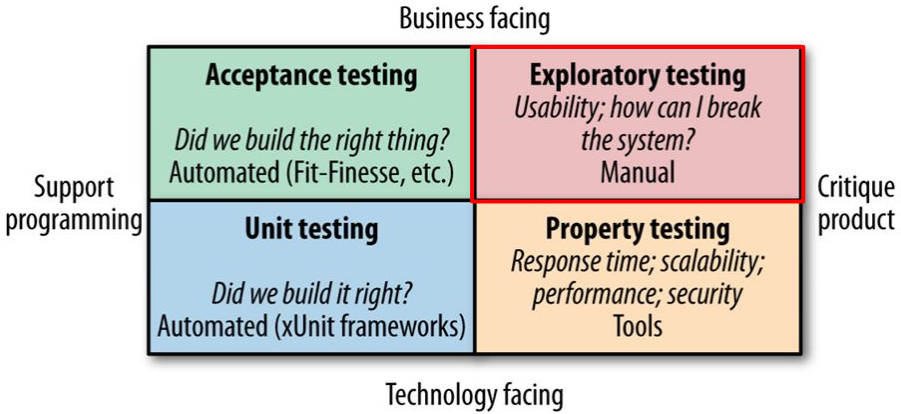
Types of Tests

We also have **business facing** tests which focus on tests that help the non-technical stakeholders understand how your system works.

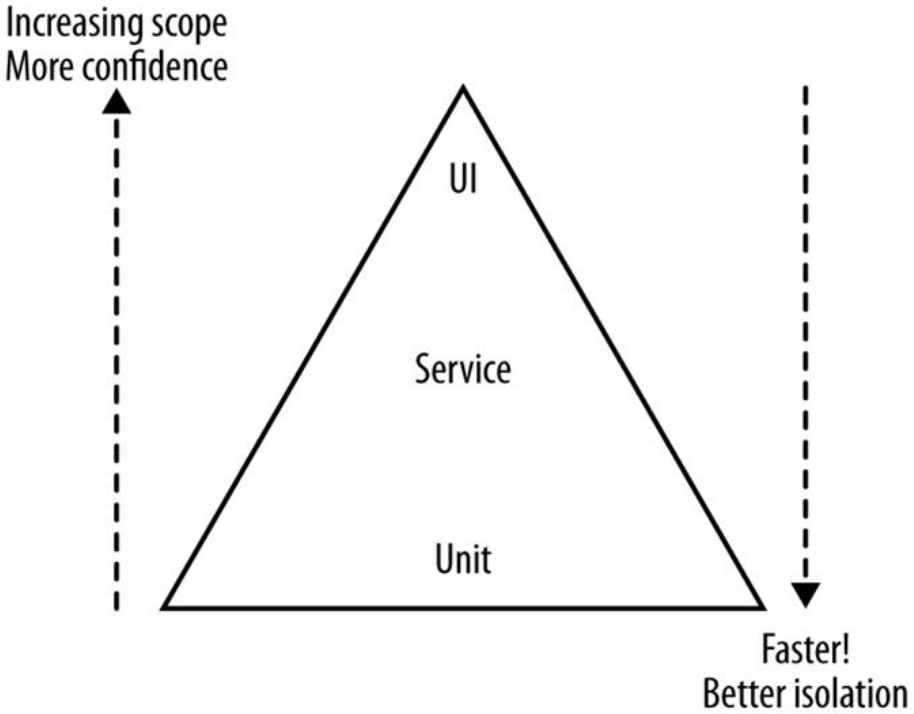
Lastly, there is Exploratory Testing that focuses on the usability of the system from a user's perspective.

These tests are largely manual with the goal of testing the system to determine how it *really* works.

This process can lead to new tests.



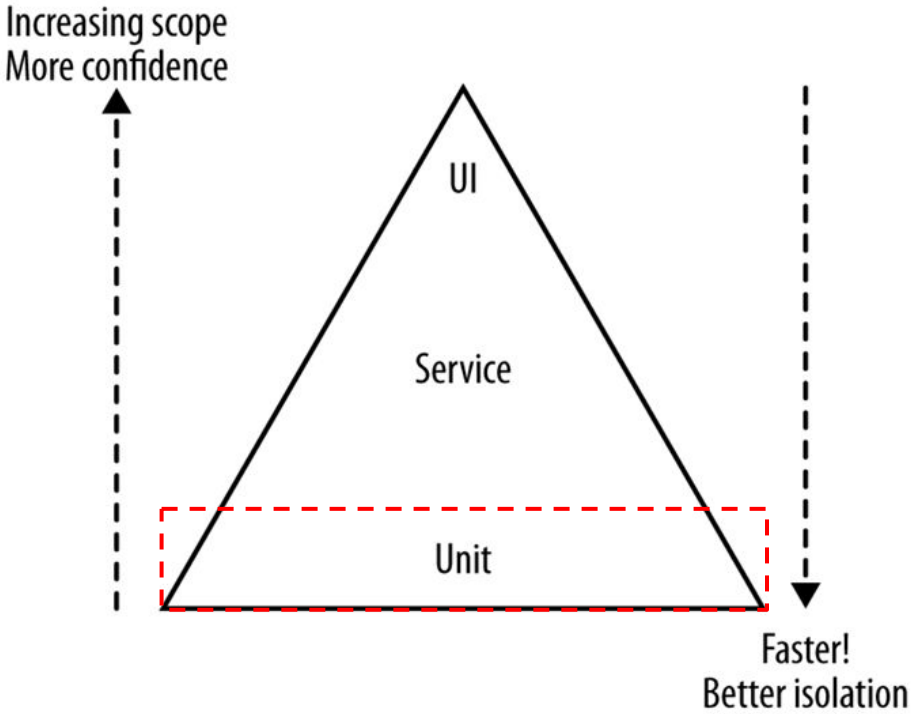
Test Scope



The figure on the left outlines a model called the Test Pyramid which was originally developed by Mike Cohn in *Succeeding with Agile*.

The pyramid helps us think about the scopes that tests should cover and the proportions of different types of tests should target.

Test Scope

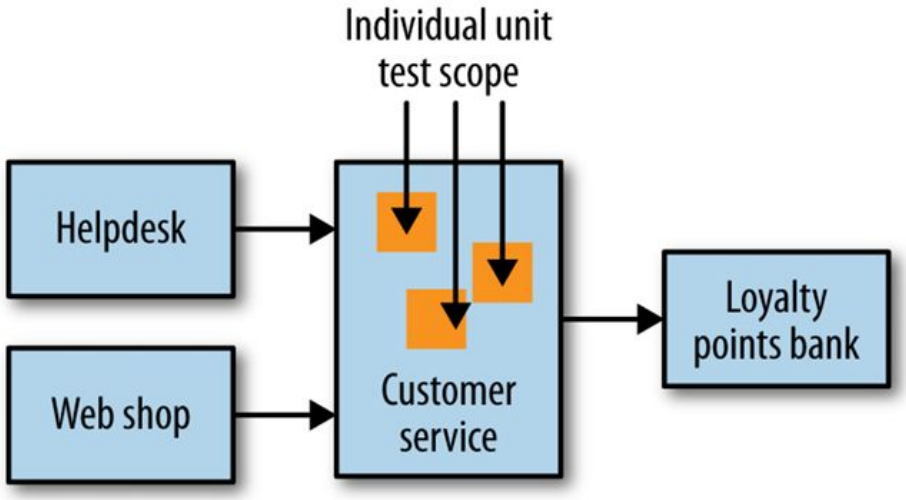


Unit tests are fine-grained tests the focus on individual lines of code, functions, or modules.

They execute quickly and there are typically many of them (often thousands).

They also catch most of the bugs in your system.

Test Scope

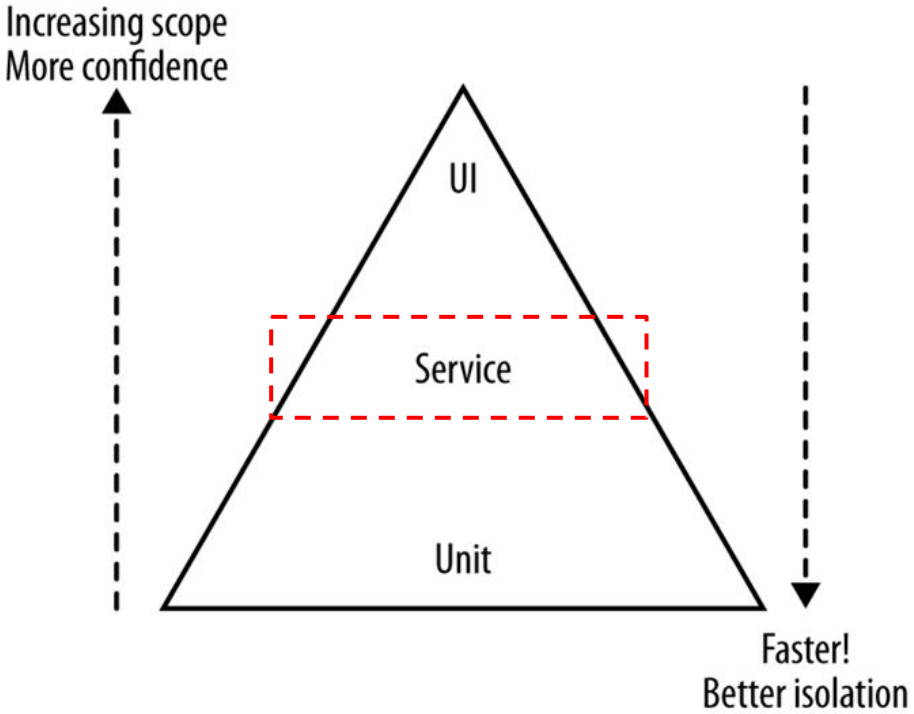


Unit tests are fine-grained tests the focus on individual lines of code, functions, or modules.

They execute quickly and there are typically many of them (often thousands).

They also catch most of the bugs in your system.

Test Scope



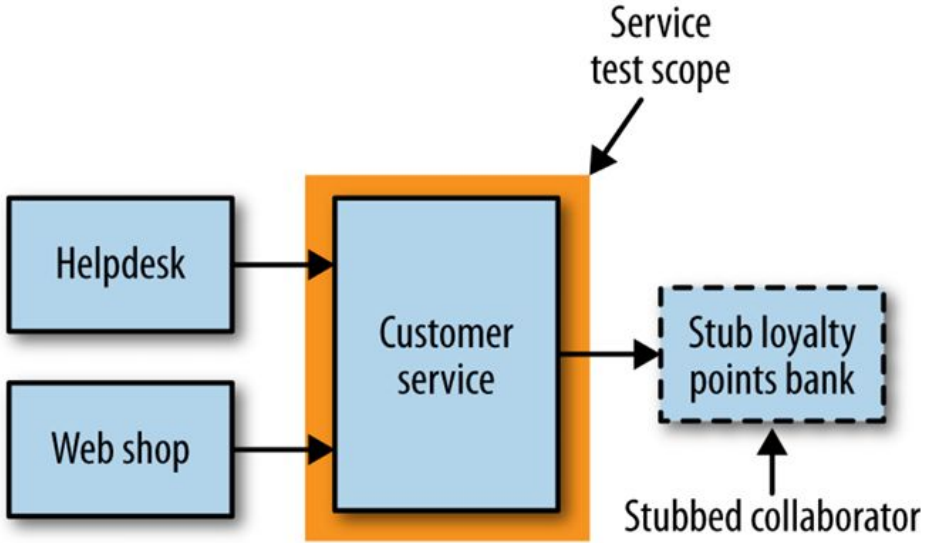
Service tests are designed to bypass the user interface and test services directly.

For a system comprising a number of services, a service test would test an individual service's capabilities.

Testing a single service in isolation will enable finding and fixing problems as quickly as possible.

To achieve this isolation, we need to stub out all external collaborators so only the service itself is in scope

Test Scope



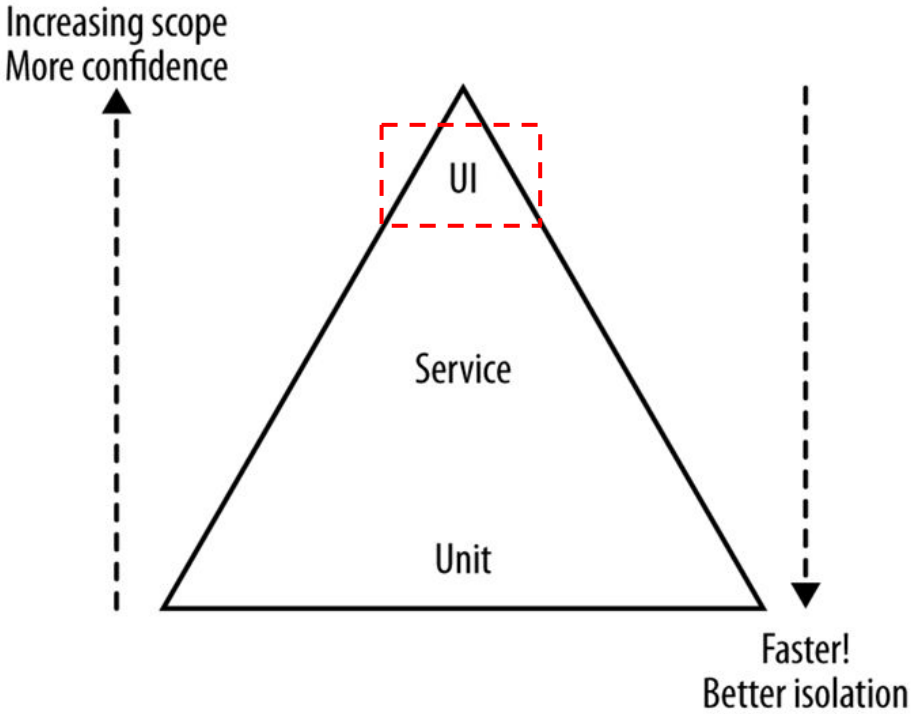
Service tests are designed to bypass the user interface and test services directly.

For a system comprising a number of services, a service test would test an individual service's capabilities.

Testing a single service in isolation will enable finding and fixing problems as quickly as possible.

To achieve this isolation, we need to stub out all external collaborators so only the service itself is in scope

Test Scope

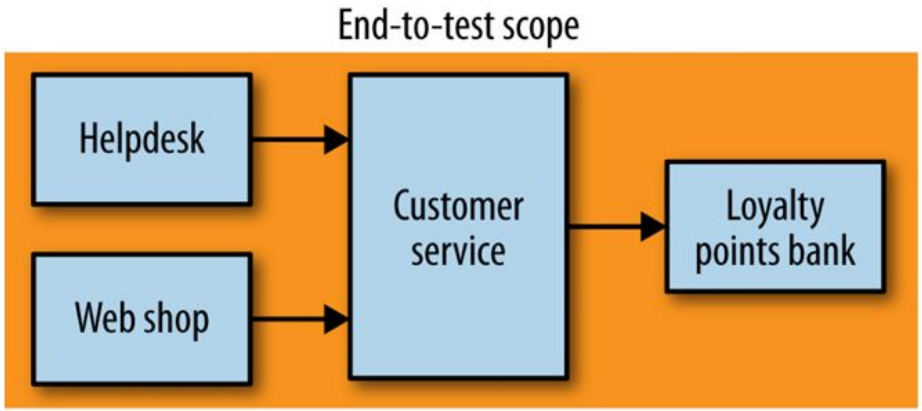


UI tests, or more clearly articulated as end-to-end tests are tests run against your entire system.

Often they will be driving a GUI through a browser, but could easily be mimicking other sorts of user interaction, like uploading a file.

They focus on the system in its entirety.

Test Scope

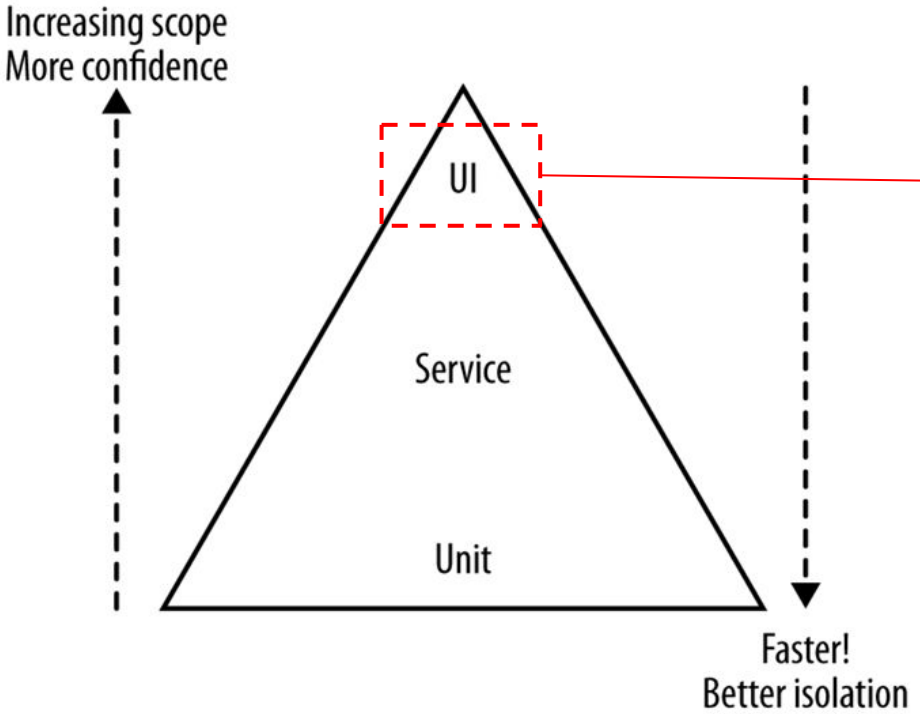


UI tests, or more clearly articulated as end-to-end tests are tests run against your entire system.

Often they will be driving a GUI through a browser, but could easily be mimicking other sorts of user interaction, like uploading a file.

They focus on the system in its entirety.

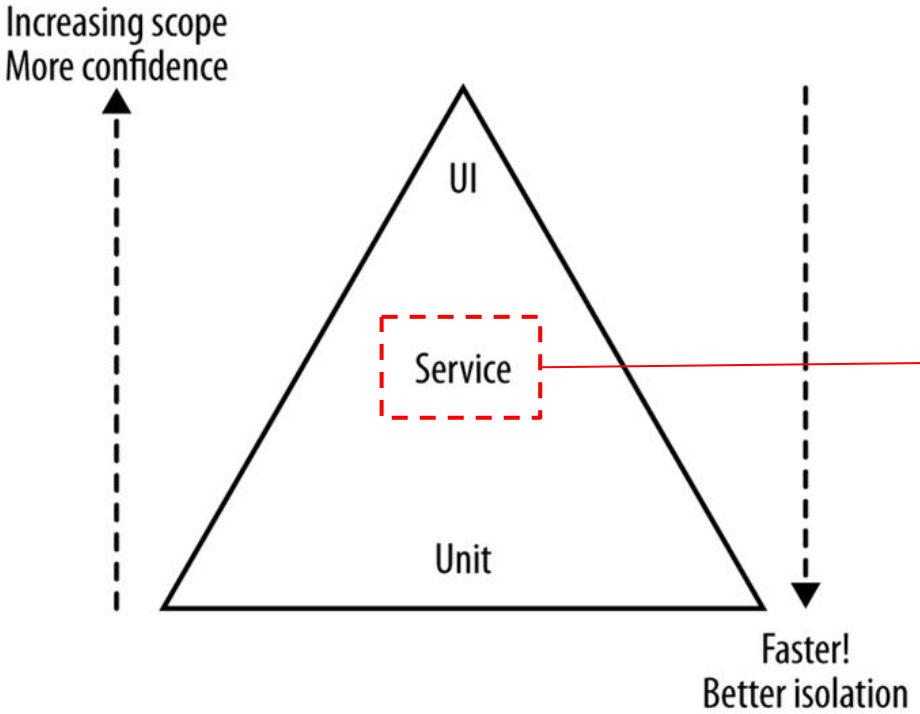
Test Scope - Trade-offs



Higher confidence that the functionality being tested works, however, these tests typically take longer to run increasing the feedback cycle time.

Generally, you have fewer of these tests and do not run them with every build.

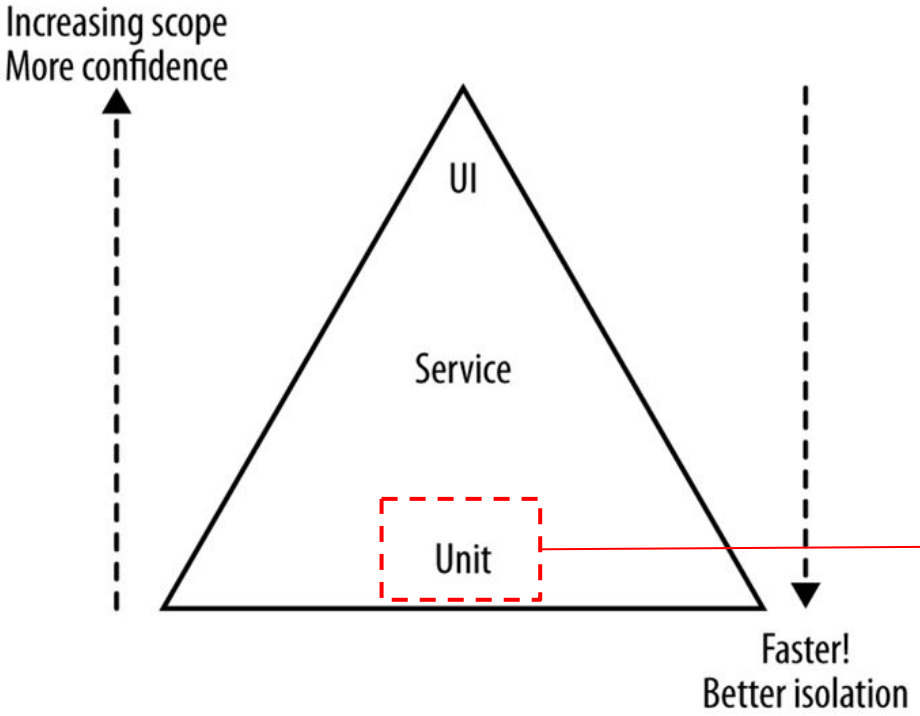
Test Scope - Trade-offs



High confidence that the functionality of an individual service being tested works. They take time to run, but not as long as an end-to-end test.

You will have more tests in this category than UI tests as you are testing each service in isolation.

Test Scope - Trade-offs



Generally, as you descend the pyramid you will have more tests that can be executed more frequently.

Often, bugs that are found in upper portions of the pyramid will dictate writing additional unit tests to catch the problem in the future.

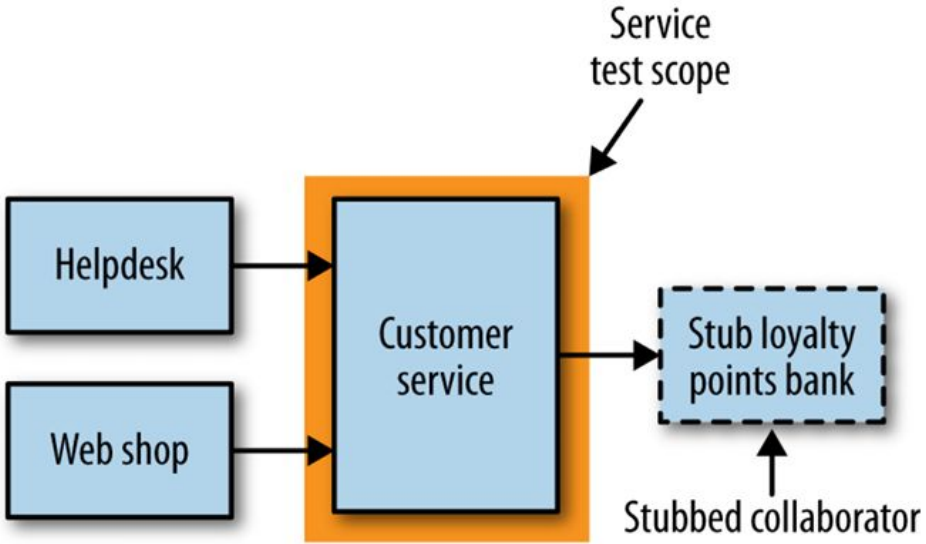
Implementing Service Tests

Implementing unit tests is relatively straight-forward.

You write a test for a specific function and determine if it is working correctly.

Service tests are a little more tricky.

Our service tests want to test a slice of functionality across the whole service, but to isolate ourselves from other services we need to find some way to stub out all of our collaborators.



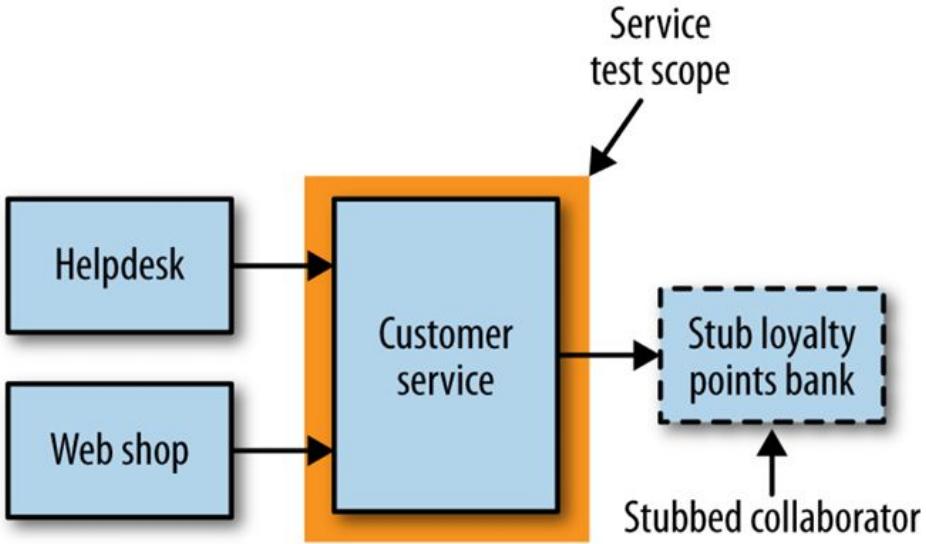
Implementing Service Tests

If we wanted to write a service test for the customer service we would:

1. Run the customer service
2. Stub out any downstream services

The primary goal is to predictably test the customer service in isolation.

Any downstream services the customer service interacts with would be stubs returning exactly what is expected for the tests.



Tricky End-to-End Tests

To implement an end-to-end test involves deploying multiple services (10, 100, possibly thousands!).

Being able to test a system end-to-end certainly elevates our confidence that the system is working properly.

However:

1. Complicated
2. Slow
3. Hard to verify the test is correct
   “flaky and brittle tests”

Flaky and Brittle Tests

Who writes these tests?

How do you coordinate multiple service teams to write tests across all services?

How long does the test take?

End-to-end tests can take a long time to execute. Do we want to wait that long to deploy?

What do we do in the meantime?

Does development stop while we run an end-to-end test that may not even be correct?

Flaky and Brittle Tests

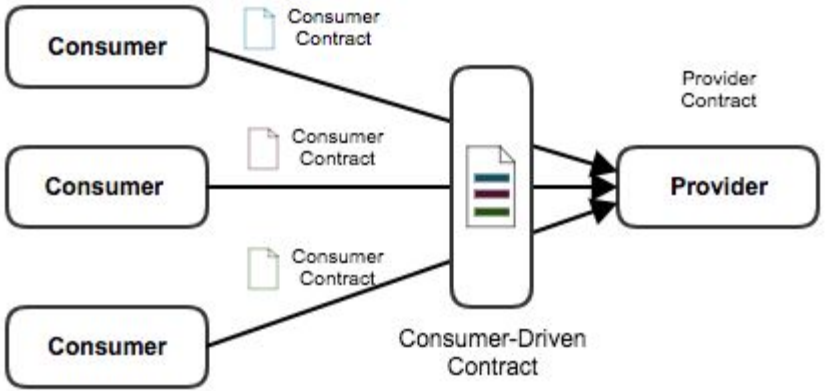
So, perhaps testing the entire system doesn't make sense…

However, it might make sense to subset your services into something more manageable and test a part of the system in isolation.

If you can do that, perhaps you can gain confidence that at least part of the system is working.

It is also easier to coordinate amongst a smaller number of teams.

Consumer Driven Tests



To be more specific, it may help to isolate the producer/provider and consumer into its own isolated testing environment.

The development team for the consumer services/UI along with the service development team (producer/provider) construct tests in collaboration.

This provides isolation in how consumers and producers interact through a contract that both parties participate in.

Testing After Production

More testing is done before a system is in production.

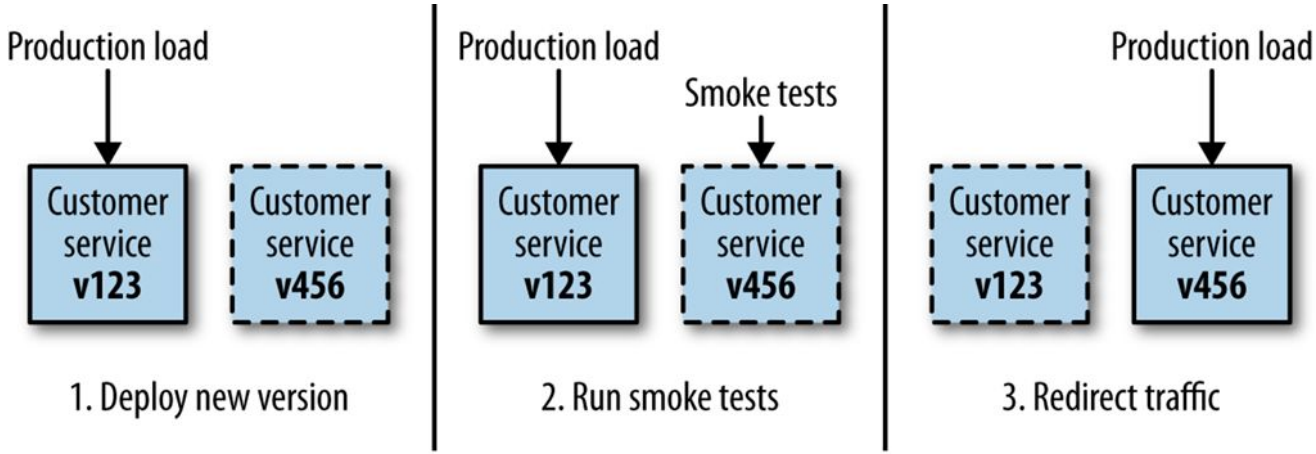
However, it may be helpful to test a system after it has been deployed to ensure that it is working correctly in the specific environment.

Users use the system in ways that are unexpected, so there are often points of failure that the development teams would have never thought of!

Separating Deployment from Release

One way of handling testing in the wild is to deploy a service and “smoke test” before redirecting production load to the new service.

A smoke test is simply testing the vital functionality of the system



Cross Functional Testing

- Acceptable latency of a web page
- The number of users a system should support
- How accessible your user interface should be to people with disabilities
- How secure your customer data should be

Performance Tests

When decomposing systems into smaller microservices, we increase the number of calls that will be made across network boundaries.

Where previously an operation might have involved one database call, it may now involve three or four calls across network boundaries to other services, with a matching number of database calls.

All of this can decrease the speed at which our systems operate.