Monday, June 5, 2017

Campfires talks: software testing

- what this presentation is about and why

- ~~why we need to test a software~~
- ~~why we automate testing process~~
- ~~what kinds of tests exist~~
  - ~~what should we test~~
  - ~~what should we NOT test~~
- ~~maslow pyramid of automated tests~~
  - ~~https://martinfowler.com/bliki/TestPyramid.html~~
  - ~~price~~
  - ~~effort~~
- ~~unit test~~
  - ~~FIRST~~
  - ~~what blocks us to write code~~
  - ~~hard to cover with tests code~~
  - ~~fast feedback loop~~
  - ~~different styles of unit tests~~
    - ~~AAA~~
    - ~~Record Reply~~
    - ~~Given When Then~~
  - ~~single assert rule~~
  - ~~test doubles~~
    - ~~different kinds~~
    - ~~mocks vs stubs~~
    - ~~seams~~

- ~~don't mock what you don't own~~
- ~~assert first rule~~
- tests are specifications
  - write to audience
- point to exact error
- spot check vs spy check
  - fragility
  - internal implementation coupling
- recording patterns -> 1s2s3s
- code coverage issues

- MOCHA specific
  - http://www.betterspecs.org/

Why we need software testing

- We owe it to our users and ourselves to deliver the best application we can.

  Ultimately, we need software testing because if we're putting a website or app out the1re, it's our responsibility to make sure it's something we can be proud of and have confidence in.

  A bug may make a simple task take a tiny bit more time or effort to complete. It may cause an image or button to break or render incorrectly. Or it may even compromise your users' privacy (Heartbleed bug, anyone?). In any case, if that bug goes unseen, it can have a real impact on real people.

- To ensure that what we create does what it's supposed to do.

  Software testing is needed to verify that your new scheduling functionality, documentation link, or live chat widget works as intended.

  A cool new feature may break a forgotten legacy feature – but hopefully regression testing catches the problem before it reaches users.

  Regardless of development methodology and whether or not your team refers to "requirements," the ultimate goal of testing is the same: to make sure that what is created does what it's supposed to do.

- Something that works when one person is using it may not work when hundreds of people are using it.

  The stress of hundreds of people hitting a website at the same time can be enough to bring it crashing down (or at the very least increase page load times annoyingly).

  You want to be sure that your website is always up and running, no matter how many people are trying to log in, run a search, purchase concert tickets, book a hotel room, register for a race… you get the picture.

  Something that works when one person is using it may not work when hundreds are, and software testing is the key to discovering those issues so you can fix them.

- There's always a chance that a user really will2 do that – no matter how silly it seems.

  Michael Bolton wrote an excellent piece on that most dreaded of bug report shut-downs: "No user would ever do that"

Too often we testers put a lot of effort into writing up a detailed, reproducible report on a bug we've discovered only to be told by management or a developer that the bug doesn't need to be addressed since, as they see it, it represents an edge case so remote that no user will ever encounter it.

Trust me: there's always a chance that a user will encounter that edge case over the course of using your application. Exploratory testing and general testing outside the specs can uncover pretty crazy bugs; the tricky part is figuring out where fixing them falls in the priority.

- There are lots of different devices, browsers, and operating systems out there.

  We're lucky to live in an age where we can choose from a technology buffet of phones, tablets, laptops, and desktops – not to mention the different browsers and operating systems in use on those devices.

  All this variety is great, but it also makes testing for compatibility essential. You never know how a user will try to view and use your site, and things like responsive design become more critical to conversions and customer satisfaction all the time.

  Testing on multiple devices, browsers, operating systems can help ensure your website works for as many of your users as possible.

Why we are automating testing process

- Automated Software Testing Saves Time and Money

  Software tests have to be repeated often during development cycles to ensure quality. Every time source code is modified software tests should be repeated. For each release of the software it may be tested on all supported operating systems and hardware configurations. Manually repeating these tests is costly and time consuming. Once created, automated tests can be run over and over again at no additional cost and they are much faster than manual tests. Automated software testing can reduce the time to run repetitive tests from days to hours. A time savings that translates directly into cost savings.

- Vastly Increases Your Test Coverage

  Automated software testing can increase the depth and scope of tests to help improve software quality. Lengthy tests that are often avoided during manual testing can be run unattended. They can even be run on multiple computers with different configurations. Automated software testing can look inside an application and see memory contents, data tables, file contents, and internal program states to determine if the product is behaving as expected. Test automation can easily execute thousands of different complex test cases during every test run providing coverage that is impossible with manual tests.

- Testing Improves Accuracy

  Even the most conscientious tester will make mistakes during monotonous manual testing. Automated tests perform the same steps precisely every time they are executed and never forget to record detailed results. Testers freed from repetitive manual tests have more time to create new automated software tests and deal with complex features

- Automation Does What Manual Testing Cannot

  Even the largest software and QA departments cannot perform a controlled web application test with thousands of users. Automated testing can simulate tens, hundreds or thousands of virtual users interacting with a network, software and web applications.

- Automated QA Testing Helps Developers and Testers

  Shared automated tests can be used by developers to catch problems quickly before sending to QA. Tests can run automatically whenever source code

changes are checked in and notify the team or the developer if they fail. Features like these save developers time and increase their confidence.

- QA and Dev Team Morale Improves

  This is hard to measure but we've experienced it first hand. Executing repetitive tasks with automated software testing gives your team time to spend on more challenging and rewarding projects. Team members improve their skill sets and confidence and, in turn, pass those gains on to their organization.

Different kinds of testing

- by execution process
  - manual
  - automated - using some software
- by treating a system
  - black box

    > Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.

  - white box
- by persons who execute tests
  - QA
  - Customer
  - Developers
- by purpose
  - functionality testing
  - security
  - load
- by scope
  - whole system
  - feature
  - component
  - integration

Most popular kinds of testing

each kind of tests has it's own GOAL and we should not mix goals, otherwise it will be harder to achieve it and make right decisions

there is no single accepted terminology

- Functional Testing

  Functional testing is a formal type of testing performed by testers. Functional testing focuses on testing software against design document, Use cases, and requirements document. Functional testing is a black box type of testing and does not require internal working of the software, unlike white box testing.

- User Acceptance testing (UAT)

  User Acceptance testing is a must for any project; it is performed by clients/end users of the software. User Acceptance testing allows SMEs (Subject matter experts) from client to test the software with their actual business or real-world scenarios and to check if the software meets their business requirements.

- Integration Testing

  Integration testing is one of the most common and important types of software testing. Once the individual units or components are tested by developers as working then testing team will run tests that will test the connectivity among these units/component or multiple units/components. There are different approaches for Integration testing namely, Top-down integration testing, Bottom-up integration testing and a combination of these two known as Sand witch testing.

- End-to-end Testing

  End to end testing is performed by the testing team and the focus is to test end to end flows e.g. right from order creation till reporting or order creation till item return etc and checking. End to end testing is usually focused on mimicking real life scenarios and usage. End to end testing involves testing information flow across applications.

- Performance Testing

  Performance Testing measures the response time of an application with an expected number of users. The aim of this is to get a baseline and an indication of how an application behaves under normal conditions. Does it meet the required response time?

- Load Testing

  Load testing is a type of non-functional testing; load testing is done to check the behavior of the software under normal and over peak load conditions. Load testing is usually performed using automated testing tools. Load testing intends to find bottlenecks or issues that prevent software from performing as intended at its peak workloads.

- Stress Testing

  Stress Testing or Soak Testing is like load testing but we resume the load on the server for a long period, say 1 hour. The aim of stress testing is to ensure that under a constant load for a long duration, the servers don't crash, albeit responding slowly. Stress testing starts of the same as load testing, e.g. gradually increasing load on the servers, but once this load is reached, we resume the same load on the server for a given duration and then measure the response times.

- Break Point

  If we keep increasing the load on the server, there comes a point when the server cannot handle any more requests and it crashes, most probably starting to give a http error 500 response code. Once this happens, we get an indication of the capacity of the application, i.e. how many users can the application handle.

- Smoke testing

  is a type of testing that is carried out by software testers to check if the new build provided by the development team is stable enough i.e., major functionality is working as expected in order to carry out further or detailed testing. Smoke testing is intended to find "show stopper" defects that can prevent testers from testing the application in detail. Smoke testing carried out for a build is also known as build verification test.

- Regression Testing

  is a type of software testing that is carried out by software testers as functional regression tests and developers as Unit regression tests The objective of regression tests is to find defects that got introduced to defect fix(es) or introduction of new feature(s). Regression tests are ideal candidates for automation.

These all tests have different goals, and idea to keep single goal and don't mix. Goal should be clear and achievable.

Every test has a price - money, effort of writing, effort to maintain, and we should carefully choose where we should put this effort.

When we are checking application execution possibilities there are also different kinds of tests (load, stress, performance) - and every kind has own goal

n my world, we use the terms as follows:

functional testing: This is a *verification* activity; did we build a correctly working product? Does the software meet the business requirements?

For this type of testing we have test cases that cover all the possible scenarios we can think of, even if that scenario is unlikely to exist "in the real world". When doing this type of testing, we aim for maximum code coverage. We use any test environment we can grab at the time, it doesn't have to be "production" caliber, so long as it's usable.

acceptance testing: This is a *validation* activity; did we build the right thing? Is this what the customer really needs?

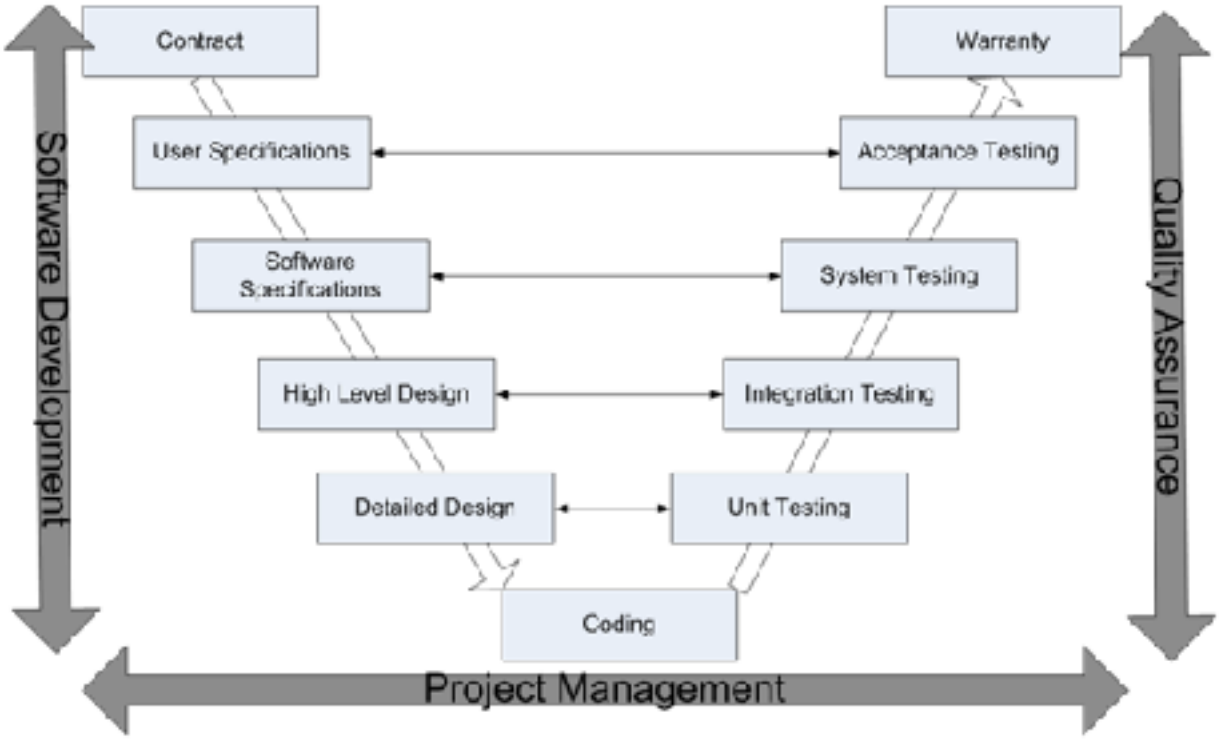
This is usually done in cooperation with the customer, or by an internal customer proxy (product owner). For this type of testing we use test cases that cover the typical scenarios under which we expect the software to be used. This test must be conducted in a "production-like" environment, on hardware that is the same as, or close to, what a customer will use. This is when we test our "ilities":

- **Reliability, Availability**: Validated via a stress test.
- **Scalability**: Validated via a load test.
- **Usability**: Validated via an inspection and demonstration to the customer. Is the UI configured to their liking? Did we put the customer branding in all the right places? Do we have all the fields/screens they asked for?
- **Security** *(aka, Securability, just to fit in)*: Validated via demonstration. Sometimes a customer will hire an outside firm to do a security audit and/or intrusion testing.
- **Maintainability**: Validated via demonstration of how we will deliver software updates/patches.

- **Configurability**: Validated via demonstration of how the customer can modify the system to suit their needs.

This is by no means standard, and I don't think there is a "standard" definition, as the conflicting answers here demonstrate. The most important thing for your organization is that you define these terms precisely, and stick to them.

Test level is easy to explain using V-model, an example:



Each *test level* has its corresponding *development level*. It has a typical time characteristic, they're executed at certain phase in the development life cycle.

1. component/unit testing => verifying detailed design
2. component/unit integration testing => verifying global design
3. system testing => verifying system requirements
4. system intergration testing => verifying system requirements
5. acceptance testing => validating user requirements

test types

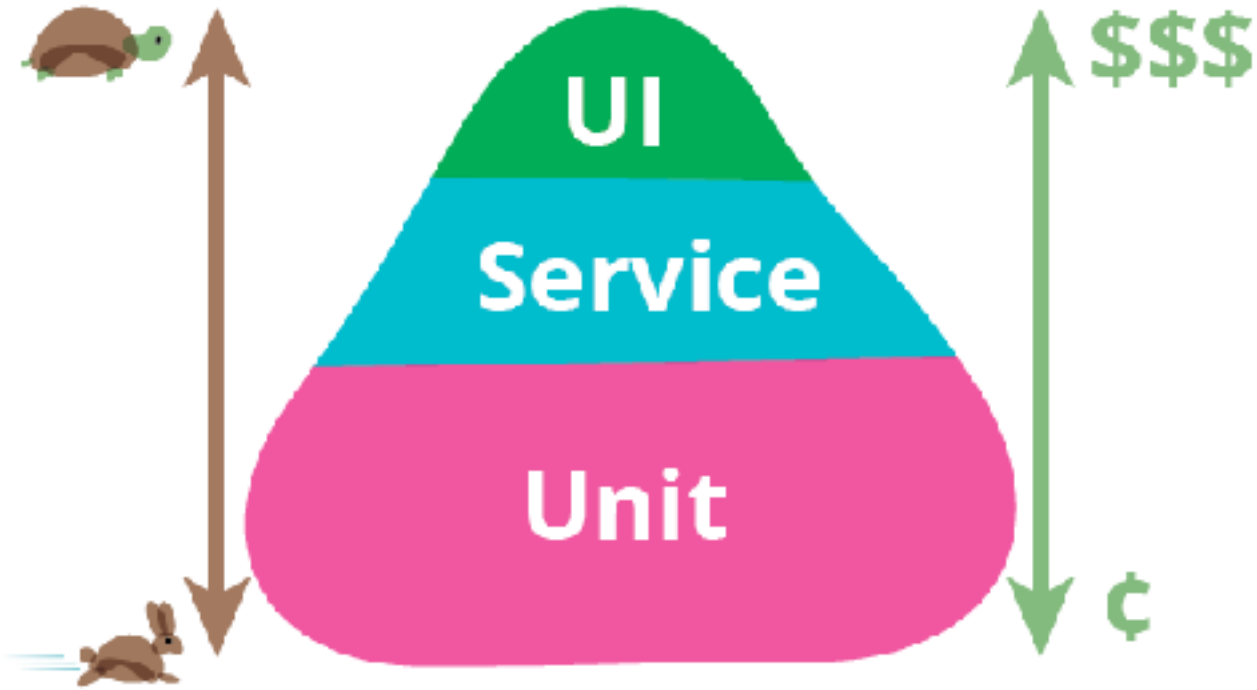
A *test type* is a characteristics, it focuses on a specific test objective. *Test types* emphasize your quality aspects, also known as technical or non-functional aspects. *Test types* **can** be executed at any *test level*. I like to use as *test types* the quality characteristics mentioned in ISO/IEC 25010:2011.

1. functional testing
2. reliability testing
3. performance testing
4. operability testing
5. security testing
6. compatibility testing
7. maintainability testing
8. transferability testing

To make it complete. There's also something called *regression testing*. This an extra classification next to *test level* and *test type*. A *regression test* is a test you want to repeat because it touches something critical in your product. It's in fact a subset of tests you defined for each *test level*. If a there's a small bug fix in your product, one doesn't always have the time to repeat all tests. *Regression testing* is an answer to that.

Maslow's pyramid of testing

- We owe it to our users and ourselves to deliver the best application we can.



proportion of performance to cost. Unit tests are most effective ones. That's why we SHOULD have much more of them, compare to UI

tests are fast, cheap and feedback cycle is short

Integration tests

contracts fulfilling -> unit test we stub that we have an 0 in case on wrong input, in real life we get an error

---

- FIRST acronym

Building the Right Feedback Loop

Tests create a feedback loop that informs the developer whether the product is working or not. The ideal feedback loop has several properties:

- It's fast. No developer wants to wait hours or days to find out if their change works. Sometimes the change does not work - nobody is perfect - and the feedback loop needs to run multiple times. A faster feedback loop leads to faster fixes. If the loop is fast enough, developers may even run tests before checking in a change.
- It's reliable. No developer wants to spend hours debugging a test, only to find out it was a flaky test. Flaky tests reduce the developer's trust in the test, and as a result flaky tests are often ignored, even when they find real product issues.
- It isolates failures. To fix a bug, developers need to find the specific lines of code causing the bug. When a product contains millions of lines of codes, and the bug could be anywhere, it's like trying to find a needle in a haystack.

Think Smaller, Not Larger

So how do we create that ideal feedback loop? By thinking smaller, not larger.

Unit Tests

Unit tests take a small piece of the product and test that piece in isolation. They tend to create that ideal feedback loop:

- Unit tests are fast. We only need to build a small unit to test it, and the tests also tend to be rather small. In fact, one tenth of a second is considered slow for unit tests.

- Unit tests are reliable. Simple systems and small units in general tend to suffer much less from flakiness. Furthermore, best practices for unit testing - in particular practices related to hermetic tests - will remove flakiness entirely.
- Unit tests isolate failures. Even if a product contains millions of lines of code, if a unit test fails, you only need to search that small unit under test to find the bug.

Writing effective unit tests requires skills in areas such as dependency management, mocking, and hermetic testing. I won't cover these skills here, but as a start, the typical example offered to new Googlers (or Nooglers) is how Google builds and tests a stopwatch.

Unit Tests vs. End-to-End Tests

With end-to-end tests, you have to wait: first for the entire product to be built, then for it to be deployed, and finally for all end-to-end tests to run. When the tests do run, flaky tests tend to be a fact of life. And even if a test finds a bug, that bug could be anywhere in the product.

Although end-to-end tests do a better job of simulating real user scenarios, this advantage quickly becomes outweighed by all the disadvantages of the end-to-end feedback loop:

Integration Tests

Unit tests do have one major disadvantage: even if the units work well in isolation, you do not know if they work well together. But even then, you do not necessarily need end-to-end tests. For that, you can use an integration test. An integration test takes a small group of units, often two units, and tests their behavior as a whole, verifying that they coherently work together.

If two units do not integrate properly, why write an end-to-end test when you can write a much smaller, more focused integration test that will detect the same bug? While you do need to think larger, you only need to think a little larger to verify that units work together.

Testing Pyramid

Even with both unit tests and integration tests, you probably still will want a small number of end-to-end tests to verify the system as a whole. To find the right balance between all three test types, the best visual aid to use is the testing pyramid. Here is a simplified version of the testing pyramid from the opening keynote of the 2014 Google Test Automation Conference:
The bulk of your tests are unit tests at the bottom of the pyramid. As you move up the pyramid, your tests gets larger, but at the same time the number of tests (the width of your pyramid) gets smaller.

As a good first guess, Google often suggests a 70/20/10 split: 70% unit tests, 20% integration tests, and 10% end-to-end tests. The exact mix will be different for each team, but in general, it should retain that pyramid shape. Try to avoid these anti-patterns:

- Inverted pyramid/ice cream cone. The team relies primarily on end-to-end tests, using few integration tests and even fewer unit tests.
- Hourglass. The team starts with a lot of unit tests, then uses end-to-end tests where integration tests should be used. The hourglass has many unit tests at the bottom and many end-to-end tests at the top, but few integration tests in the middle.

Just like a regular pyramid tends to be the most stable structure in real life, the testing pyramid also tends to be the most stable testing strategy.

Hard to test code

javascript is really powerful, so many problems you can solve, but they increase effort to write a test and make tests more complicated more

---

- singlethons / static context

in c sharp it is static construcors, real singletones

in js it is different, Dynamic imports ???

destructuring imports - if you get directly a meyhod you unable to mock it

inline initialization, when you define some internal variables in shared place, we cannot adjust them for testing

same constants could be an issu, like const waittimeout and in test we cannot decrease it

- object construction

solution is deprndency injection

- different levels of abstraction

in business level you should mock and work with streams or filesystem interface

- **too many collaborations - split by responsibility - like react smart and dum**

F.I.R.S.T.

Fast - they should be fast to run in order to run them frequently. Slow tests are executer rarely, in most cases only on CI. Or developers execute only subset of tests - but that can hide a problem and increase feedback cycle

I/O make tests slow in most cases. Autogenerated tests also could be slow and in most cases are useless. We should test logic / algorithms, not configuration.

Isolated / Independent - every test should have single reason to fail. On failure they should point to exact problem. In ideal case from test name we should immediately understand what is not working, and don't go deeper to check setup part, or act part, or go to SUT and check implementation. Bad thing when one test create a shared state (like object in DB, or setup global object) and other tests expect that system is already in this state. All tests should executed in clean ENV. They also should not depend on order of execution and don't interferes with other tests.

Repeatable - You should obtain the same results every time you run a test. Also the result should be independent from ENV or time. Common reason to fail here:

- shared objects in memory

- external dependency (file system, db, web service) which maintain state

- Non-deterministic behavior - using of DateTime, Random or threads

- over-specification - comparing entire screen images or HTML when only a small part of the result is interesting

Self-Verifying - A good unit test fails or passes unambiguously. We should don't have cases when after tests suit we should go to DB or logs file and manually check if everything was OK.

Timely - Write tests before production code. Write it for every feature.

As we can see some kind of tests are not follow this guides, like integration tests or end-to-end tests - they usually rely on some preset or previous state of tests.

there are some other rules

- It is written gray-box, i.e. it reads as if it were black-box, but sometimes takes advantage of white-box knowledge. (Typically a critical factor in avoiding combinatoric issues.)

- It is coded to the same standard as shipping code, i.e. the team's best current understanding of coding excellence.
- It provides precise feedback on any errors that it encounters
- It is written before the code-change it is meant to test.

Arrange Act Assert

record and reply

some time ago was popular syntax for tests based on mocks which called record and reply

```
MockRepository mocks = new MockRepository();
using (mocks.Record()) {
   ILoanRepository loanRepo = mocks.StrictMock<ILoanRepository>();

SetupResult.For(loanRepo.GetLoanExtended("sdfsdf")).Return(list.AsEnumerable<Loan
>());
}
using (mocks.Playback()) {
   //test execution
}
```

syntax is not really readable, so later it was changed to match AAA -> verify

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExist()
{
   // Arrange
   var guid = Guid.NewGuid();
   _viewMock.Setup(x => x.Respondents).Returns(new[] { guid.ToString() });
   _repositoryMock.Setup(x => x.GetById(guid)).Returns(() => null);
   // Act
   _viewMock.Raise(x => x.ExecuteOperation += null, EventArgs.Empty);
   // Assert
   _viewMock.VerifySet(x => x.OperationErrors =
      It.Is<IList<string>>(l => l.Contains("Non-existant respondent: "+guid)));
```

}

Four Phase Test

Arrange - Act - Assert - Teardown

Given - When - Then - Teardown

The essential idea is to break down writing a scenario (or test) into three sections:

The given part describes the state of the world before you begin the behavior you're specifying in this scenario. You can think of it as the pre-conditions to the test.

The when section is that behavior that you're specifying.

Finally the then section describes the changes you expect due to the specified behavior.

BDD style ->

Gherkin is the language that Cucumber understands. It is a Business Readable, Domain Specific Language that lets you describe software's behaviour without detailing how that behaviour is implemented.
Gherkin serves two purposes — documentation and automated tests. The third is a bonus feature — when it yells in red it's talking to you, telling you what code you should write.

```
1: Feature: Some terse yet descriptive text of what is desired
2:   Textual description of the business value of this feature
3:   Business rules that govern the scope of the feature
4:   Any additional information that will make the feature easier to understand
5:
6:   Scenario: Some determinable business situation
7:     Given some precondition
8:       And some other precondition
9:     When some action by the actor
10:      And some other action
11:      And yet another action
12:    Then some testable outcome is achieved
13:      And something else we can check happens too
14:
15:  Scenario: A different situation
```

16: ...

Arrange: Set up the object to be tested. We may need to surround the object with collaborators. For testing purposes, those collaborators might be test objects (mocks, fakes, etc.) or the real thing.

Act: Act on the object (through some mutator). You may need to give it parameters (again, possibly test objects).

Assert: Make claims about the object, its collaborators, its parameters, and possibly (rarely!!) global state.

Occasionally a sequence is needed, but the 3A pattern is partly a reaction to large tests that look like this:

Arrange

Act

Assert

Act

Assert

Arrange more

Act

Assert

…

To understand a test like that, you have to track state over a series of activities. It's hard to see what object is the focus of the test, and it's hard to see that you've covered each interesting case. Such multi-step unit tests are usually better off being split into several tests.

tests doubles

test double -> dump (nulls or do nothing) -> stub (test specific values or behavior like exception throw) -> spy (also track calls, which parameters and how many times) -> mocks (smart enough to verify expected behavior)

fakes - simulate real system, used mostly for integration or acceptance testing

Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

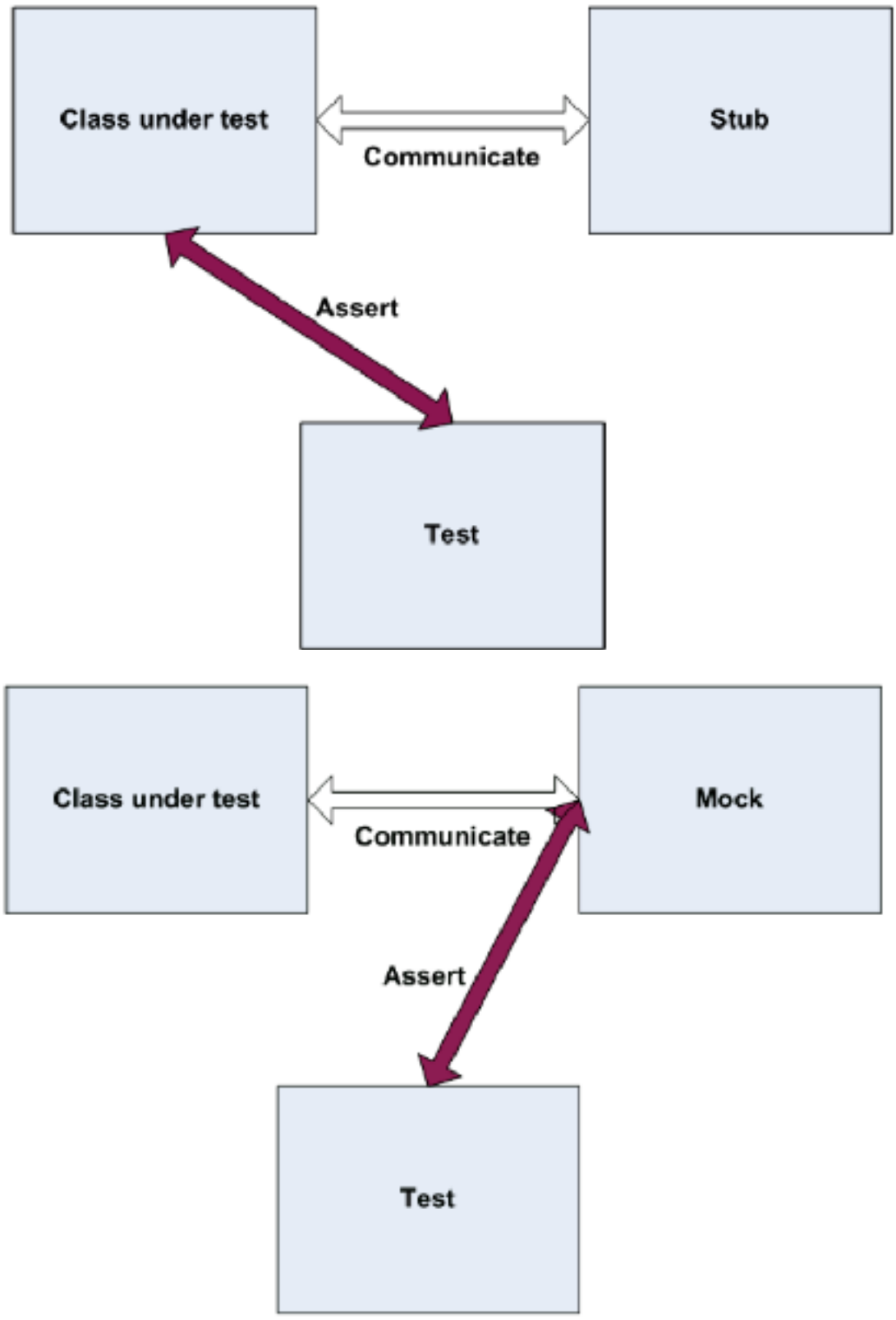
Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.

Mocks are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

mocks vs stubs



mocks can fall tests, stubs should not, only logic in SUT can do this.

so that's mean 1 mock per tests (could be many stubs)

if there is a mock -> verify only it without other things

seam - A seam is a place where you can alter behavior in your program without editing in that place. the place where we can inject test double, it's very from language to language.

some language allow you do link seam, compile code to different output for tests.

Dependency injection -> on construction level, setters level, invocation level (parameter)

sorting method

protected methods, factories.

Test practices

- single "logical" assert

  every test should contain single logical assert. we dont want to have act / assert / act / assert. in this cases the tests willbe harder to read and understand.

  sometimes we can use "gard asserts" like act / assert no null / act / assert

  in most cases every guard assertion could be represented as a separate test, but its up to developer, if it really make sense to keep it is not a big issue if used carefully.

  logical assert does not mean single expect statement, or single verify mock, sometimes several checks represent single logical operation and we should check it. the main goal - when test is broken we easily understand where the problem is.

- assert first style

  sometimes it is easy to write tests from the end. like - I expect to have 100 usd on my account after putting 50 usd on it ad having previously 50 there. thats also nice because sometimes you can write assertions in a wrong way - they will be always green and like TDD you can spot this kind of errors

- don't mock what you don't own

  the idea here to make some wrapper / abstraction over some frameworks or native methods - where you can hide low level details and later easily fix migration issues. like after updating from frameworkX from version 4 to version 5

it start throw exceptions rather then return empty value. also its good in case of switching frameworks. you should mock behavior you which you control

- write to audience

betterspecs

- use contexts
- keep your descriptions short

new technics

- snapshot testing
    - https://medium.com/powtoon-engineering/a-complete-guide-to-testing-javascript-in-2017-a217b4cd5a2a
    -
- crazy monkey
- feather data generation testing
- karma vs mocha vs chai vs selenium vs test cafe
-

Resources:

https://github.com/ghsukumar/SFDC_Best_Practices/wiki/F.I.R.S.T-Principles-of-Unit-Testing

https://pragprog.com/magazines/2012-01/unit-tests-are-first

https://dzone.com/articles/writing-your-first-unit-tests

http://agileinaflash.blogspot.de/2009/02/first.html

http://howtodoinjava.com/best-practices/first-principles-for-good-tests/

http://agileinaflash.blogspot.de/2009/03/arrange-act-assert.html

http://wiki.c2.com/?ArrangeActAssert

vs Given When Then

https://softwareengineering.stackexchange.com/questions/308160/differences-between-given-when-then-gwt-and-arrange-act-assert-aaa

https://martinfowler.com/bliki/GivenWhenThen.html

http://xunitpatterns.com/Four%20Phase%20Test.html

http://xp123.com/articles/3a-arrange-act-assert/