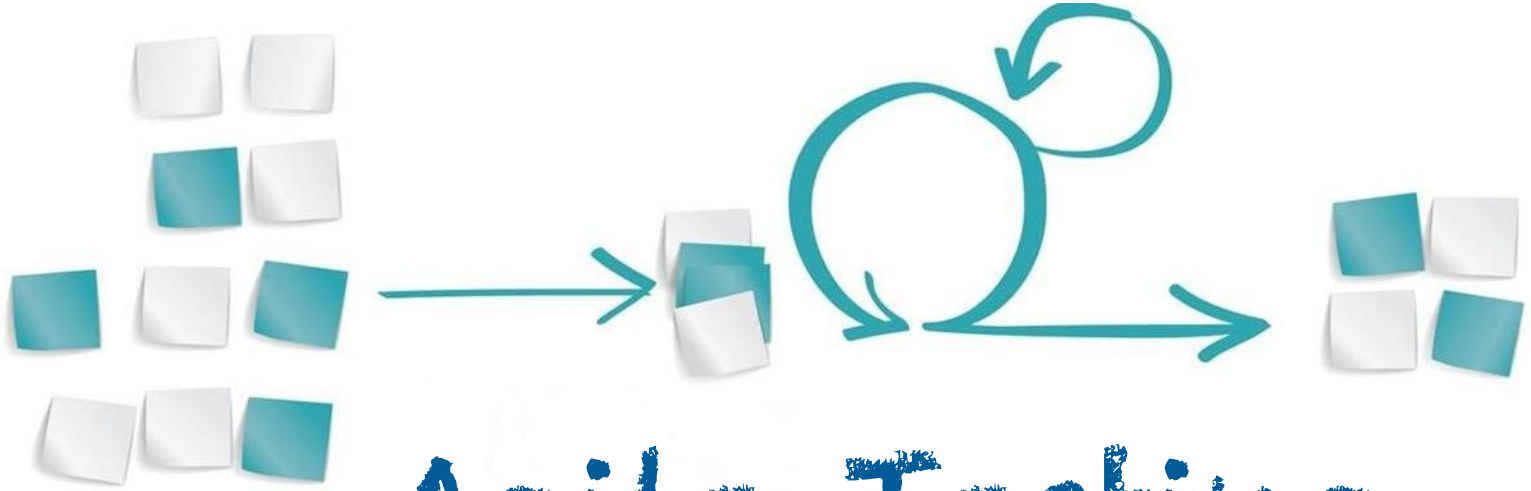
intelliware
software development

Agile



Agile Testing

BC Holmes
@bcholmesdotorg

What you'll learn in this presentation:

- Why do we use Agile testing?
- What Agile testing isn't
- What Agile testing is: unit testing and test-driven development (TDD)
- High-level properties of good tests
- Testing in different languages
- Test suites and code coverage
- Using mock objects to help isolate units
- Beyond unit testing

Why do we use Agile testing?

Design

- A test usually provides a concrete usage scenario that helps a developer think about the following when writing code:
  - What is the code expected to do?
  - How will the code be set up and executed?
  - What other pieces of code are needed as collaborators, and how will the code interact with them?
- Especially good at helping to write code that is loosely coupled.

Why do we use Agile testing?

Focus/Rhythm

- Unit tests provide an easy way for a developer to focus on a particular coding task.
  - Clear and fast feedback if the code is working or not.
- Always clear what to do next:
  - Either fix a failing test or write a new test as a prelude to adding functionality.

What Agile testing isn’t

Agile Testing is not a “phase”.

- In traditional methodologies, testing is a phase of the project that happens after coding.
- In Agile methodologies, coding, refactoring and testing are tightly inter-related tasks and can’t be separated.

A testing “phase” that’s months, weeks or even days in the future provides very slow feedback to developers.

- Faster feedback gets failure information to developers while the details of the code implementation is still fresh in their minds.

What Agile testing isn't (continued)

Agile testing is not a replacement for quality assurance (QA) testing.

- Agile testing is a practice to help developers create better code.
- Agile testing is intended to improve the quality (design, defects, etc.) of the final code deliverables.
  - That might mean that you have a lower need for QA, but not necessarily.

It's noteworthy that Lisa Crispen's book, *Agile Testing*, is principally about the role of dedicated QA-style testers on Agile projects, rather than the testing practices of Agile developers.

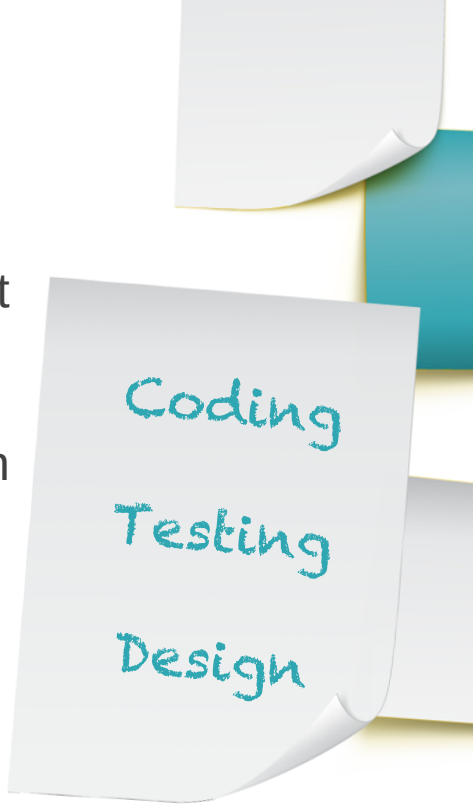
What Agile testing is:

unit testing and test-driven development

Test-driven development (TDD)

TDD is one of the earliest forms of Agile testing, and the most commonly implemented types of Agile testing. It's a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

- Write a "single" unit test describing an aspect of the program.
- Run the test, which should fail because the program lacks that feature.
- Write "just enough" code, the simplest possible, to make the test pass.
- "Refactor" the code until it conforms to the simplicity criteria.
- Repeat, "accumulating" unit tests over time.

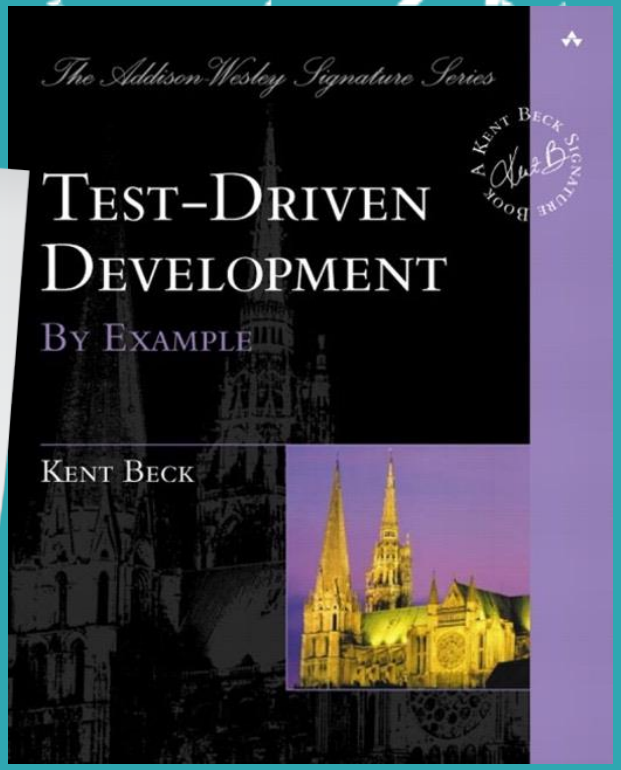


Source: Agile Alliance

Origins

Developed by Kent Beck in 2002

Described in the book ***Test-Driven Development: By Example***

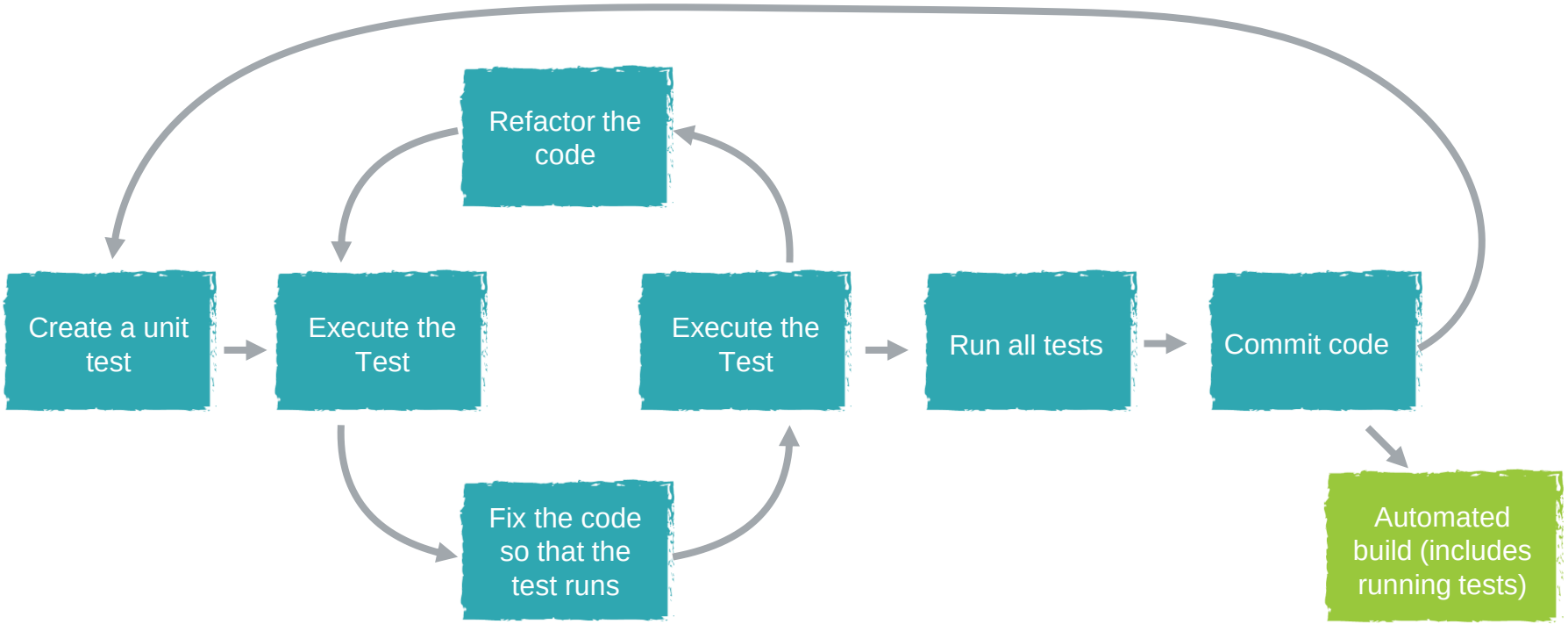


What is a unit test?

"In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use. Intuitively, **one can view a unit as the smallest testable part of an application**. […] In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. **Unit tests are short code fragments created by programmers… during the development process.**"

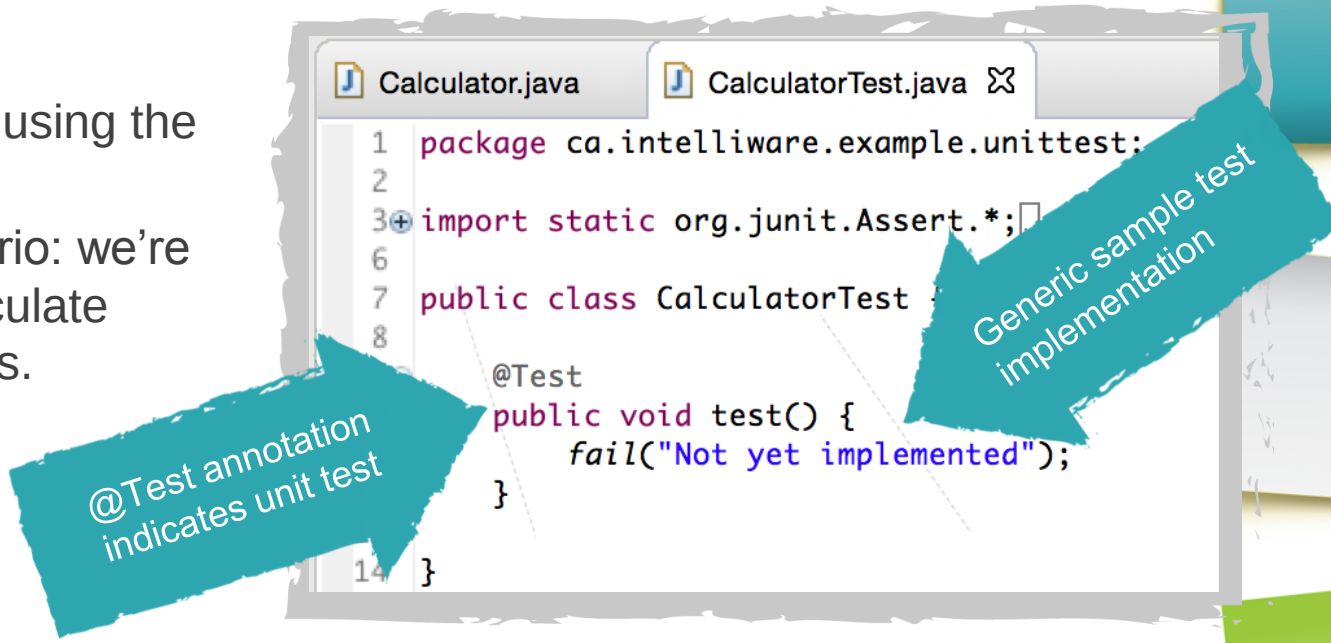
Source: Wikipedia

The TDD process



Creating a simple unit test in Java

- Unit tests are most commonly created using the JUnit framework.
- Imagine this scenario: we're writing code to calculate mortgage payments.



Create code that tests a specific expectation

Calculator.java | *CalculatorTest.java | Mortgage.java

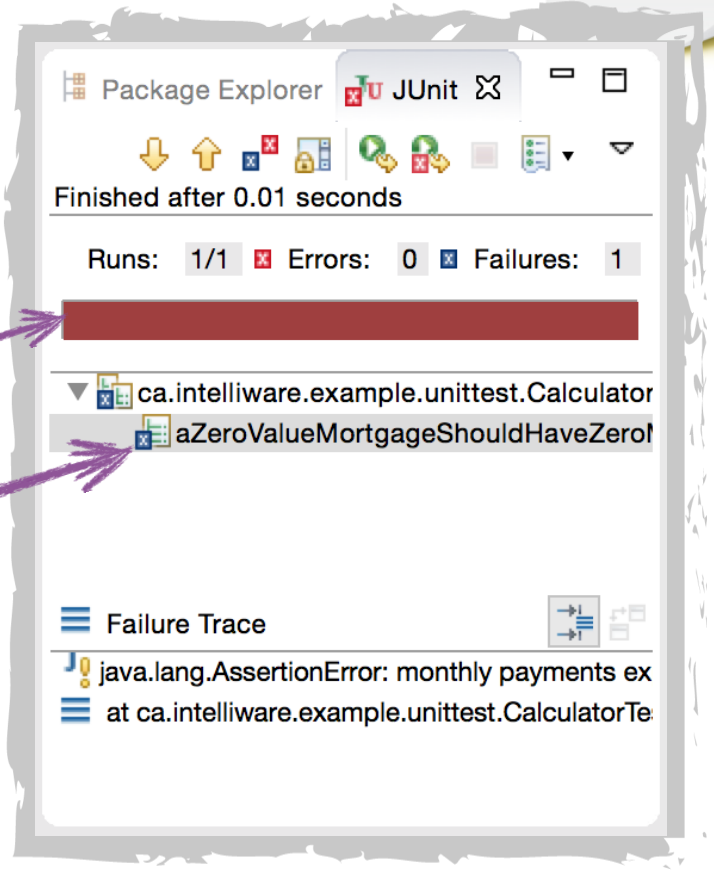
```
1  package ca.intelliware.example.unittest;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.junit.Test;
6
7  public class CalculatorTest {
8
9      @Test
10     public void aZeroValueMortgageShouldHaveZeroMonthlyPayments() {
11         Mortgage mortgage = new Mortgage(0);
12         Calculator calculator = new Calculator(mortgage);
13
14         assertEquals("monthly payments", 0, calculator.calculateMonthlyPayments());
15     }
16 }
```

Give the individual test a descriptive name

Make an assertion of what the code should do

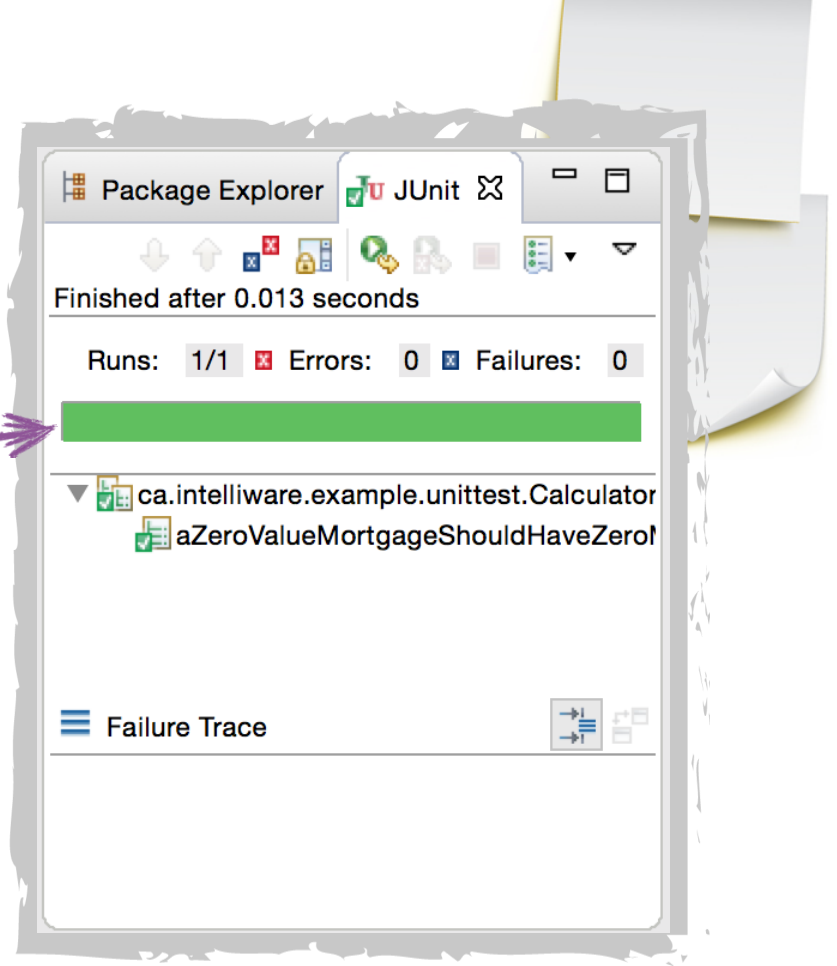
Run the unit test

- In this example, we're using Eclipse's built-in JUnit test support to run the unit test.
- The red bar indicates that our test doesn't yet pass successfully.
- Note how the test name appears to clarify which test has failed.
  - The name gives an indication of the expected behaviour.



Fix code and rerun

- The developer should write enough code to make the test pass.
- The green bar indicates that all the expected behaviour is correctly implemented.



Rules about tests

- Often, once the tests are green, the developer's next step is to refactor code.
  - Refactoring involves improving the design of existing code.
    - Make it more readable, increase the efficiency, or remove duplication.
  - The tests help to ensure that the developer hasn't broken any functionality during the refactoring.
- Run all tests before committing or pushing code in to the source code repository.
  - If there are any broken tests, you should not commit.
- Automated build processes should run all of the tests as an indication of the validity of the build artifacts.

Some high-level properties of good tests

Properties of good tests

Power: A test should reveal problems when they exist.

Valid: When the test states that there's a problem, it is a genuine problem.

Value: Tests should reveal problems that the client cares about.

Credible: The test should be trusted to actually test the things that people expect it to test.

Representative: Of events that are likely to be encountered by the user. (Not necessarily "commonly", but likely). Hacking attempts, for example, might be likely, but not common.

Non-redundant: The test shouldn't reproduce the work of another test.

Motivating: Your client will want to fix the problem exposed by this test.

Source: Cem Kaner

Properties of good tests (continued)

Performable: Designing a test that can't actually be performed is useless.

Maintainable: If the end product changes, it's important to be able to update the tests.

Repeatable: It should be easy and inexpensive to execute the test again and/or reuse it as appropriate.

Coverage: The test exercises a part of the code that isn't already handled by another test.

Easy to evaluate: It should be relatively easy to determine whether the test has passed or failed.

Insightful: A good test should reveal things about our assumptions.

Source: Cem Kaner

Properties of good tests (continued)

Supports troubleshooting: If the test fails, it should help the programmer identify the problem.

Appropriately complex: No more complex than necessary.

Accountable: Can you explain why the test exists, and can you demonstrate that it was executed?

Cost: the cost to create/execute/maintain the test should compare favourably to the value.

Opportunity Cost: The test obviates the need to perform some other work.

Source: Cem Kaner

Testing in different languages

Common unit test frameworks

Java

- JUnit, TestNG.

.Net

- Microsoft Unit Testing Framework and NUnit.

JavaScript

- Mocha, Jasmine, Unit.js, Nodeunit, QUnit, JSUnit, and many others.

iOS / Objective-C

- XCUnit (successor to OCUnit).

Python

- Nose, unittest (a.k.a. PyUnit).

Unit testing with JavaScript and Mocha.js

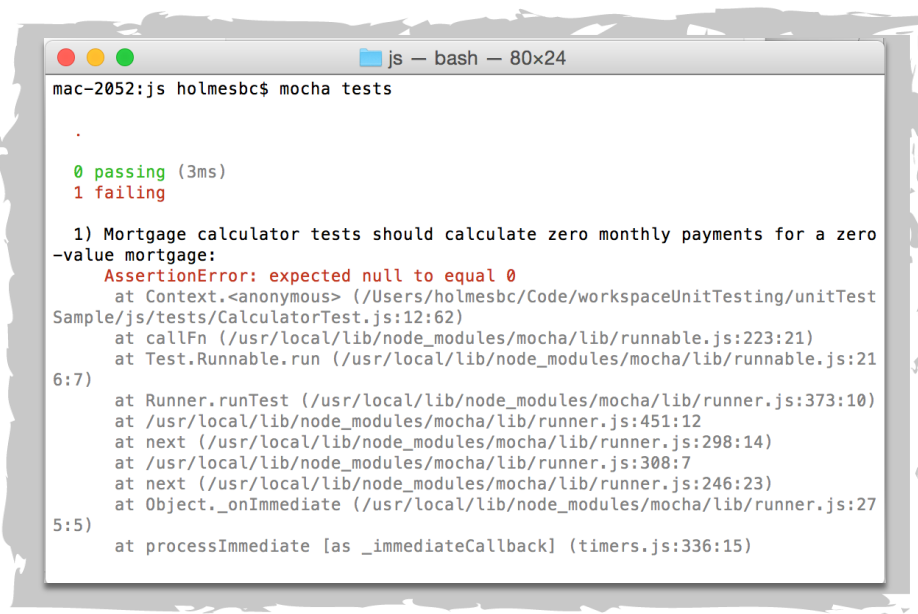
Slightly different syntax from JUnit, but most concepts are comparable.

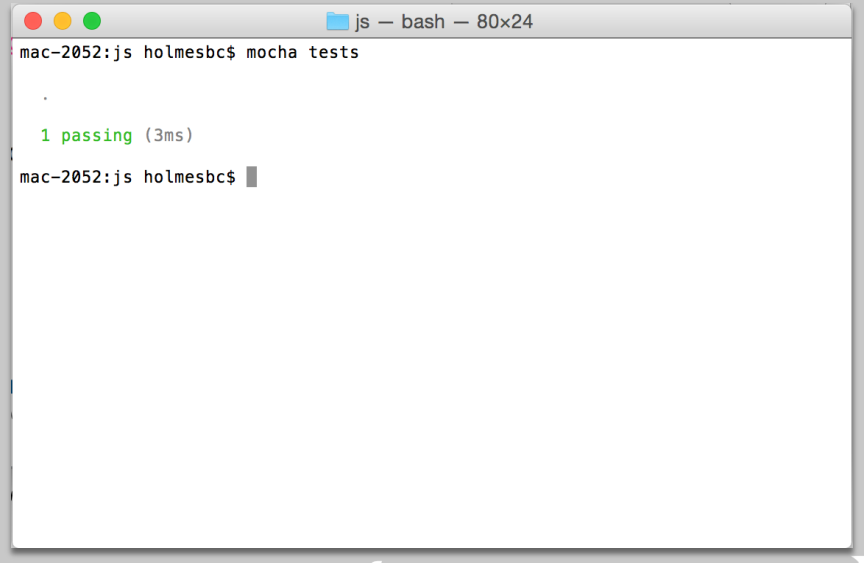
CalculatorTest.js (no symbol selected)

```
1  var chai = require('chai');
2  var mortgage = require('../lib/mortgage');
3  var calculator = require('../lib/calculator');
4  var expect = chai.expect;
5
6  describe('Mortgage calculator tests', function() {
7
8      it("should calculate zero monthly payments for a zero-value mortgage", function() {
9
10         var myMortgage = mortgage.createWithValue(0);
11
12         expect(calculator.calculateMonthlyPayments(myMortgage)).to.equal(0);
13     });
14 });
```

Executing Mocha.js on the command line

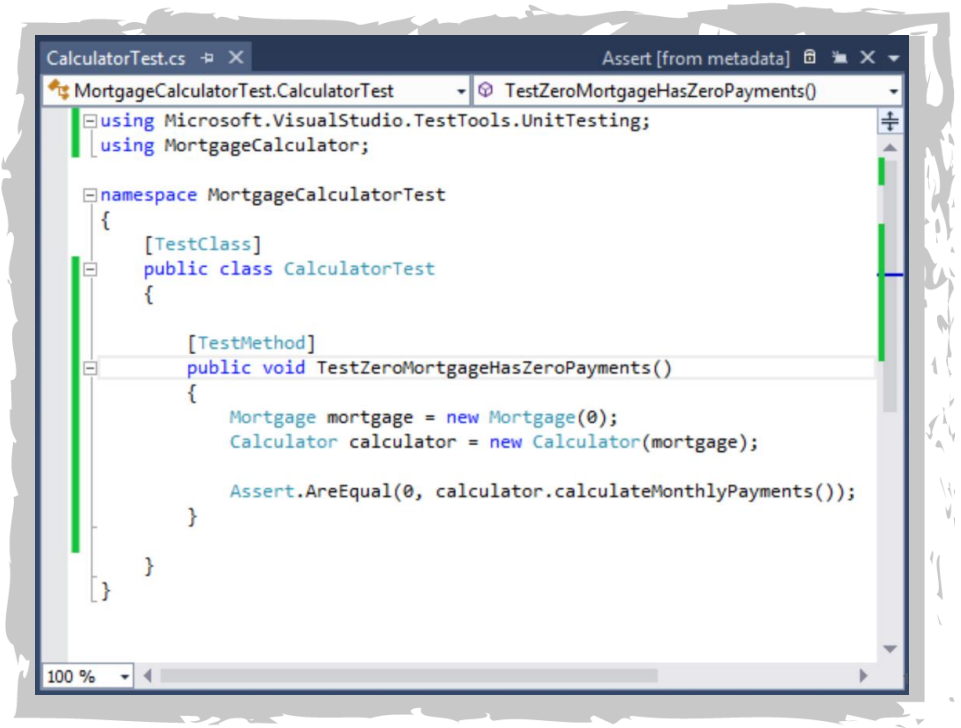
Most JavaScript unit test frameworks are command-line-based.



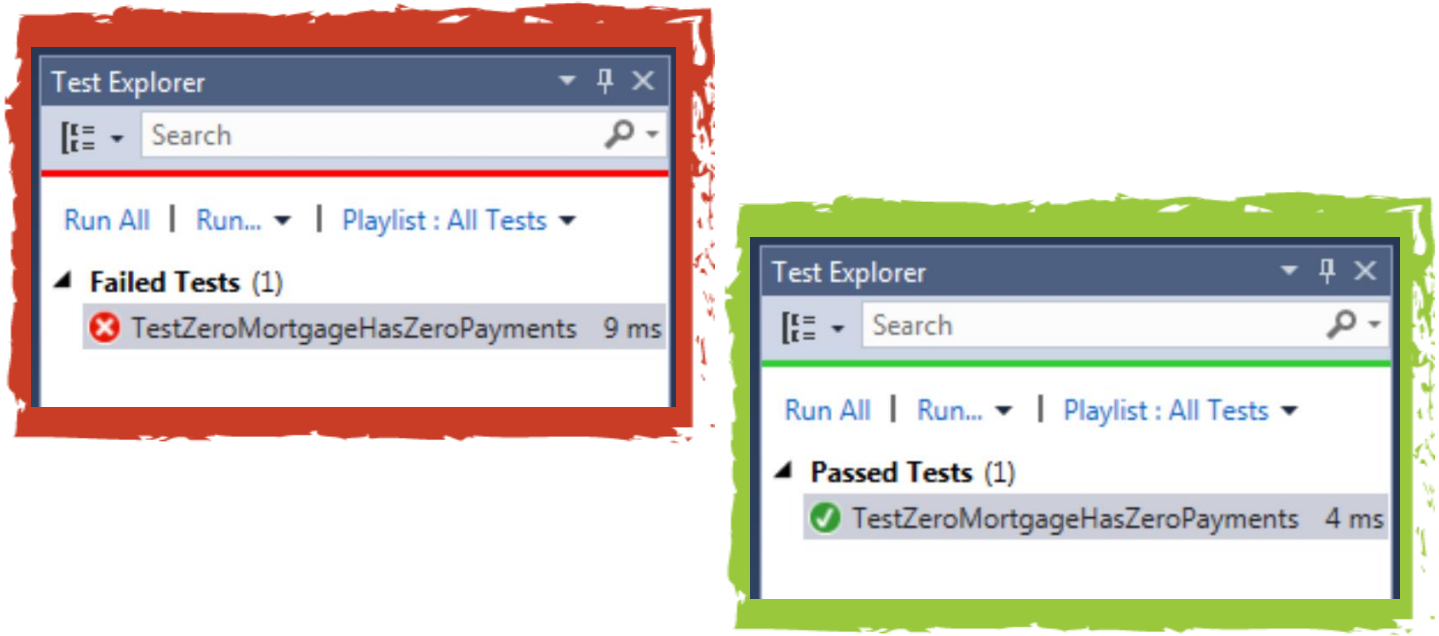


Unit testing with C# and Microsoft's unit testing tools

Again, slightly different syntax, but same basic concepts.



Visual Studio’s built-in test support

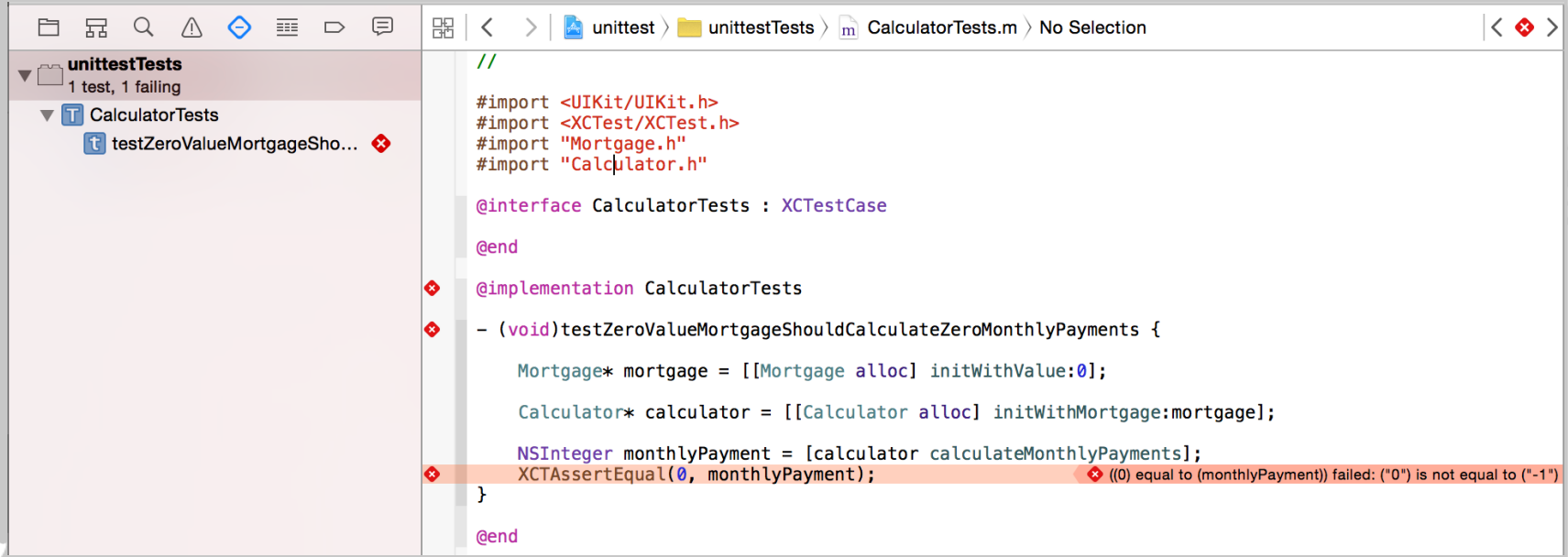


Testing with Objective-C and XCUnit

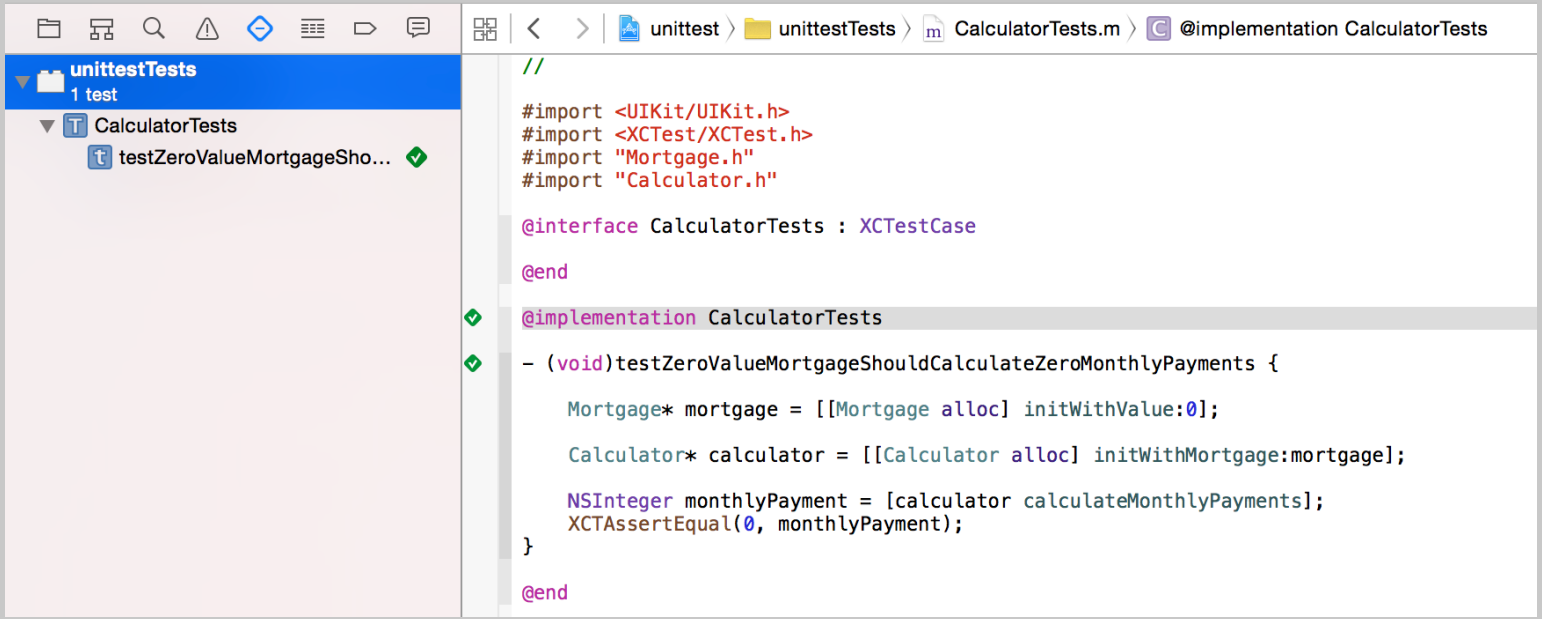
unittest 〉 unittestTests 〉 CalculatorTests.m 〉 -testZeroValueMortgageShouldCalculateZeroMonthlyPayments

```
//

#import <UIKit/UIKit.h>
#import <XCTest/XCTest.h>
#import "Mortgage.h"
#import "Calculator.h"

@interface CalculatorTests : XCTestCase

@end

@implementation CalculatorTests

- (void)testZeroValueMortgageShouldCalculateZeroMonthlyPayments {

    Mortgage* mortgage = [[Mortgage alloc] initWithValue:0];

    Calculator* calculator = [[Calculator alloc] initWithMortgage:mortgage];

    NSInteger monthlyPayment = [calculator calculateMonthlyPayments];
    XCTAssertEqual(0, monthlyPayment);
}

@end
```

XCode's built-in support for XCUnit



Successful execution



Key insights

Unit testing in a variety of languages (and test frameworks) is largely the same.

- Test a specific function and give it a descriptive name.
- Set up some necessary classes and/or data.
- Invoke the function/method under test.
- Check/assert that calculated values match expectations.

Most test runner tools use red and green indicators to tell developers the current state of the tests.

Test suites and code coverage

Test suites

Over time, as more and more tests are created, a sizeable amount of the application is backed by unit tests that are executed many times a day to ensure that previously-created functionality still works.

- Hundreds or even thousands of unit tests are not uncommon for a single application.

As more and more tests are added, the amount of time it takes to execute the tests increases.

- If that amount of time is too long, developers will start becoming discouraged from running the tests regularly.
  - A few seconds is awesome.
  - Less than 30 seconds is still pretty good.
  - A few minutes isn't terrible.
  - More than that is likely to be irritating.

Code coverage

Some languages/environments have tools that can calculate the amount of your application's code that gets executed during the running of the unit test suite.

- These tools are called "code coverage" or "test coverage" tools.

High coverage usually indicates that the application is well-tested, and that usually results in developer confidence that they can change the application and not break the existing functions.

- How high is high enough? There are a lot of factors and not all projects are the same, but 75% or 80% coverage is often used as a starting goal.

Some stuff is hard to test

Visual output

- Does a web page look attractive? Is an animation jerky or glitchy? Is one component of a web page covering or obscuring another component?
- Graphical output — especially image-based.
  - Example: a class that outputs a bar chart as a PNG image.

Code that's date/time-driven or based on timers

- But if you parameterize the date/time information, you can often mitigate the difficulty.

Legacy code

- The best time to start unit testing code is at the beginning of a project.
- Legacy code, especially tightly-coupled stuff, is harder to make testable.

Using mock objects to help isolate units

Mock objects: dealing with collaborators

"In object-oriented programming, **mock objects are simulated objects that mimic the behaviour of real objects** in controlled ways. A programmer typically creates a mock object to test the behaviour of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behaviour of a human in vehicle impacts."

Source: Wikipedia

Tools for working with mock objects

Usually involves a mock framework

- Java
  - Mockito, Jmock
- JavaScript
  - Sinon.js
- Objective-C
  - OCMock

Example

We want to test functionality related to renewing a mortgage.

- Renewing a mortgage should be based on same payment type, but current interest rate.

Mortgage collaborates with a class that provides the current interest rates.

- But we're currently testing the renew functionality, not the providing of interest rates.

Let's try an example using Java and Mockito.

MortgageTest.java

```java
package ca.intelliware.example.unittest;

import static org.junit.Assert.assertEquals;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MortgageTest {

    @Mock
    RatesProvider ratesProvider;

    @Test
    public void renewMortgageShouldUseCurrentRate() {

        float newInterestRate = 0.042f;
        Mockito.when(ratesProvider.getVariableInterestRate()).thenReturn(newInterestRate);

        float originalInterestRate = 0.035f;
        Mortgage mortgage = new Mortgage(15000000, originalInterestRate, PaymentType.VARIABLE);

        Mortgage renewedMortgage = mortgage.renew(ratesProvider);

        assertEquals("new interest rate", newInterestRate, renewedMortgage.getInterestRate(), 0.0001);
    }

}
```

Need a special test runner

Mock object

Configure the mock object's behaviour

```
11 @RunWith(MockitoJUnitRunner.class)
12 public class MortgageTest {
13
14     @Mock
15     RatesProvider ratesProvider;
16
17     @Test
18     public void renewMortgageShouldUseCurrentRate() {
19
20         float newInterestRate = 0.042f;
21         Mockito.when(ratesProvider.getVariableInterestRate()).thenReturn(newInterestRate);
22
23         float originalInterestRate = 0.035f;
24         Mortgage mortgage = new Mortgage(15000000, originalInterestRate, PaymentType.VARIABLE);
25
26         Mortgage renewedMortgage = mortgage.renew(ratesProvider);
27
28         assertEquals("new interest rate", newInterestRate, renewedMortgage.getInterestRate()  0.0001);
29
30         Mockito.verify(ratesProvider, Mockito.atLeastOnce()).getVariableInterestRat
31     }
32
33 }
```

Can additionally verify that mock object was called

Additional thoughts

Mock object frameworks provide a powerful way to ensure that you're testing only a single unit at a time.

- Helps isolate an individual unit.

If the configuration of your mock object starts to get complex or messy, that's an indication that your class is too tightly intertwined with the mock object's class.

- Remember: loosely-coupled classes are easier to test, and usually a better design choice.

Defining your units

A “unit” is often a class or small collection of related classes.

- Anything not contained in that “unit” is a dependency or collaborator.
- We usually create mock objects to take the place of the dependency/collaborator when we build the unit test.

Consider special case: a class that reads data from a database.

- In Java, such classes are often referred to as “Data Access Objects” or DAOs.
- Key question: does the “unit” that you’re testing include the database?
  - Or should the database be considered a dependency/collaborator?

My gut is telling me “maybe”

The case for “yes”

- It’s often hard to mock out an entire database API.
- Including the database allows you to test more: object-relational mapping configuration, query correctness, etc.
- If you use database features such as triggers or stored procedures, this might be the only way you can test those.

The case for “no”

- Unit tests that connect to the database are often slower than most other unit tests.
  - Both because of set up and because of I/O.
- You need to consider the contents of the database: does your test expect certain records? How do they get there? What if those contents are changed? Managing the initial state or “home state” of the database is a non-trivial exercise.

Database tests

Don’t get hung up on the name.

- Whether or not they’re “unit” tests, it’s often fruitful to include tests that touch the real database.

Dealing with “Home State” is often worth the initial investment.

- Home state is a version of the database (or other resource) that contains the data that the tests expect.

Approaches to home state

Restore the database prior to running tests.

- Always brings the data back to home state, but might take time.

Have a separate database specifically for unit tests to use.

- But need to apply necessary table definition changes as the project evolves.

Don’t let unit tests change the state of the database.

- Rollback any changes or undo database changes when the tests finish.
- Doesn’t help if the same database is used for ad-hoc testing.

Try to minimize the expectations.

- Not always possible or easy.

Home state management isn’t easy, but the benefits are worth it.

Beyond unit testing

Beyond unit testing

Although Agile testing is often described in terms of creating unit tests, there are other forms of testing that can prove useful for an agile project.

Common test types:

- **Integration Tests**: tests that integrate a number of units, to ensure that the parts interact with one another properly.
- **UI Tests:** tests that automate interaction with a UI and the complete application codebase to ensure that all parts function together properly.
- **Performance Tests:** tests that automatically capture information about the response time of the application, possibly under load.
- **Static Analysis Tests:** tests that analyze code, looking for violations of coding conventions and development rules.
- **Acceptance Tests**: tests that demonstrate to the customer that the requested functionality has been implemented.

Integration testing

Sometimes, the integration between multiple units is sufficiently complex that it's fruitful to test it.

- Especially true of application-specific code that needs to be integrated with and configured in major frameworks.

Examples:

- Ensure that a certain number of classes are correctly configured and wired together in a Spring configuration file.
- Ensure that a SOAP interface has been correctly configured to send or receive the necessary data.

The goal is to integrate no more parts than absolutely necessary.

Integration tests (continued)

Because more parts need to be set up and initialized, integration tests may take longer than other types of tests.

- Integration tests might end up being run less frequently.

Still interested in this question: “Do the integration tests run quickly enough that we can expect the developers to run them prior to synching with source control?”

- A “yes” answer provides more value than a “no” answer, but a “no” answer can still be helpful.

UI tests

Can be considered a special case of integration tests — essentially integrating all of the parts.

Examples:

- Web UI Tests.
  - Launch the full application in a web server.
  - Selenium is a very commonly-used web UI test tool.
- Mobile UI Tests
  - There are many approaches to building mobile apps.
    - Native, HTML5 or Mobile Application Development Platforms (MADPs) like Worklight, Kony or Appcelerator Titanium.
  - As a result, no one mobile UI test tool has become *de facto* standard, so mobile UI testing is still an emerging discipline.

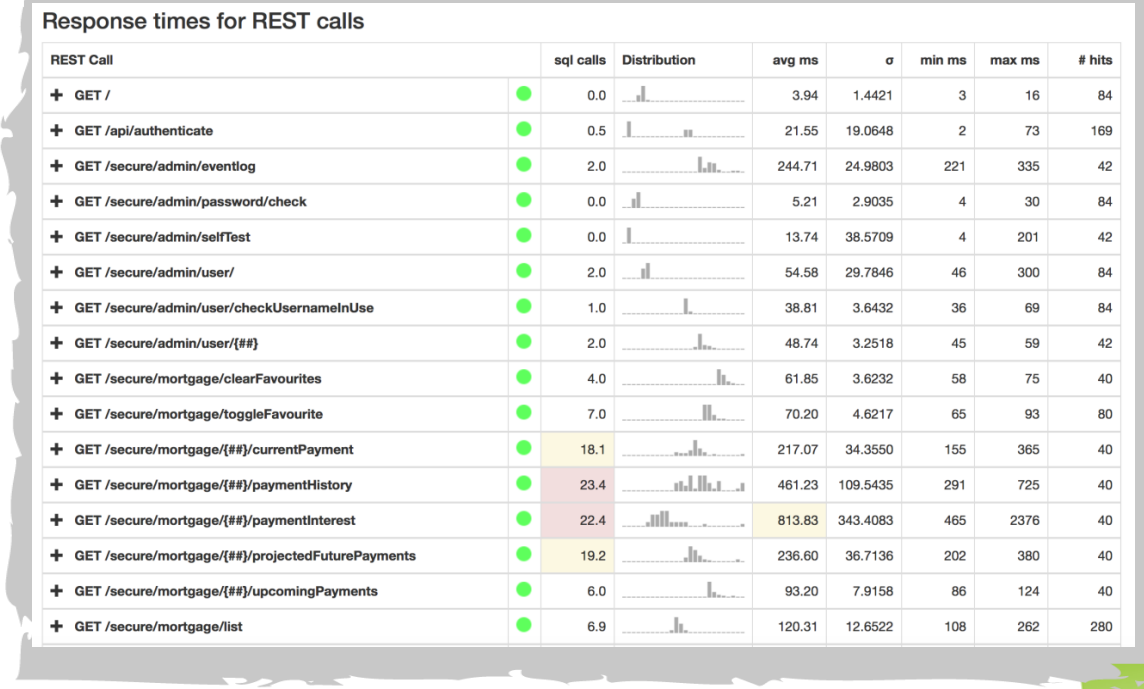
Performance tests

- If one of the requirements (user stories) of your application involves performance targets, testing is the ideal way to ensure that those targets are met.
- While Agile developers often caution against premature performance optimization, it's also not necessarily a good idea to wait until the end of a project to test performance.
- Performance testing often tests the integration of many of the parts of the application, rather than just individual units.
- We frequently instrument applications to gather performance metrics, and run a suite of performance tests that can give us a performance checkpoint with every build.

Automated performance test report

Three types of data:

- What's the current performance, at the highest level (see diagram)?
- What's the current performance at different layers of the application (Service Layer, DAO Layer).
- What the historical performance?



Response times for REST calls

REST Call		sql calls	Distribution	avg ms	σ	min ms	max ms	# hits
+ GET /		0.0		3.94	1.4421	3	16	84
+ GET /api/authenticate		0.5		21.55	19.0648	2	73	169
+ GET /secure/admin/eventlog		2.0		244.71	24.9803	221	335	42
+ GET /secure/admin/password/check		0.0		5.21	2.9035	4	30	84
+ GET /secure/admin/selfTest		0.0		13.74	38.5709	4	201	42
+ GET /secure/admin/user/		2.0		54.58	29.7846	46	300	84
+ GET /secure/admin/user/checkUsernameInUse		1.0		38.81	3.6432	36	69	84
+ GET /secure/admin/user/{##}		2.0		48.74	3.2518	45	59	42
+ GET /secure/mortgage/clearFavourites		4.0		61.85	3.6232	58	75	40
+ GET /secure/mortgage/toggleFavourite		7.0		70.20	4.6217	65	93	80
+ GET /secure/mortgage/{##}/currentPayment		18.1		217.07	34.3550	155	365	40
+ GET /secure/mortgage/{##}/paymentHistory		23.4		461.23	109.5435	291	725	40
+ GET /secure/mortgage/{##}/paymentInterest		22.4		813.83	343.4083	465	2376	40
+ GET /secure/mortgage/{##}/projectedFuturePayments		19.2		236.60	36.7136	202	380	40
+ GET /secure/mortgage/{##}/upcomingPayments		6.0		93.20	7.9158	86	124	40
+ GET /secure/mortgage/list		6.9		120.31	12.6522	108	262	280

Static analysis tests

- Developers tend to think of functional business requirements as the things that drive unit tests.
  - Unit tests can additionally be built around non-functional requirements or even development rules or standards.
- Examples:
  - Organization has a web standard to use relative units (e.g.: rems) rather than pixel units (px) in web page style sheets (CSS).
    - Write a unit test that scans all of the CSS and fails if px units are found.
  - Architectural standard that Java code should not have cyclical package dependencies.
    - Write a unit test that fails if package cycles are found.

Static analysis tests (continued)

- Provides proactive identification of problems, rather than after-the-fact need for clean-up.
- Interesting consequence:
  - Tests continue to enforce rules as more code or more developers are added to the project.
  - Developers can’t say, “nobody told me about that rule!”.

Acceptance tests

- An acceptance test is a description of a scenario that demonstrates the expected behaviour of a software component.
- Ideally, the acceptance test is recorded in a way that's readable by a customer (and developers), but still supports automated execution.
  - Several tools such as JBehave or Cucumber, attempt to make acceptance tests readable while still providing automated execution.
- Acceptance Testing is often tied to the idea of "Behaviour Driven Development".
  - Emphasizes "specifications" and "scenarios".
  - Many BDD proponents like the "Given-When-Then" structure.
  - **Given** a particular context, **when** an action is taken, **then** a particular result is expected.

About Intelliware Development Inc.

Intelliware is a custom software, mobile solutions and product development company headquartered in Toronto, Canada. Intelliware is a leader in Agile software development practices which ensure the delivery of timely high quality solutions for clients. Intelliware is engaged as a technical partner by a wide range of organizations in sectors that span Financial Services, Healthcare, ICT, Retail, Manufacturing and Government.

- /company/intelliware-development-inc-
- /intelliware.inc
- /intelliware_inc
- /GooglePlusIntelliware

www.intelliware.com

Check out other entries in our Agile methodology series