Code Smells -> Refactoring -> Unit Tests

Runs: 3/3 Errors: 0 Failures: 0

com.mlevison.codesmells.GameTest
testDefaultMove (0.000 s)
testFindWinningMove (0.000 s)
testWinConditions (0.001 s)

Mark Levison

Agile Change Consultant

Blog: notesfromatooluser.com

mark@theagileconsortium.com

THE agileconsortium

http://www.istockphoto.com/stock-photo-4745224-striped-skunk.php

Background survey

- Who we are?
- What do you want from this session?

http://www.sxc.hu/photo/426123/

Just a general sampling survey – 5-6 people, no more than 60 seconds each – Tick marks trick.

Do you know SOLID code?

Why should you care?
Fewer bugs
Easier to write new code
Time to market
Easy to read code
Go home early on Friday

Agenda

• Technical Debt	5 min
• Code Smells	15 min
– Exercise	10 min
• Refactoring	15 min
• Unit Tests	15 min
– Pairing Exercise	10 min
• Q&A	5 min

SOLID Principles

SRP The Single Responsibility Principle *A class should have one, and only one, reason to change.*
OCP The Open Closed Principle *You should be able to extend a classes behavior, without modifying it.*
LSP The Liskov Substitution Principle *Derived classes must be substitutable for their base classes.*
DIP The Dependency Inversion Principle *Depend on abstractions, not on concretions.*
ISP The Interface Segregation Principle *Make fine grained interfaces that are client specific.*

Reference: Uncle Bob: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Its iteration 15 and the project is starting to slow down. For the past few iterations the team hasn't been able to complete quite as many stories as they did in the past. In addition more bugs have been found recently both in the new stories and also regressions. The manager knows that the team members haven't changed and they're still working the same hours. The customer is asking what happened? Is the team still working hard?

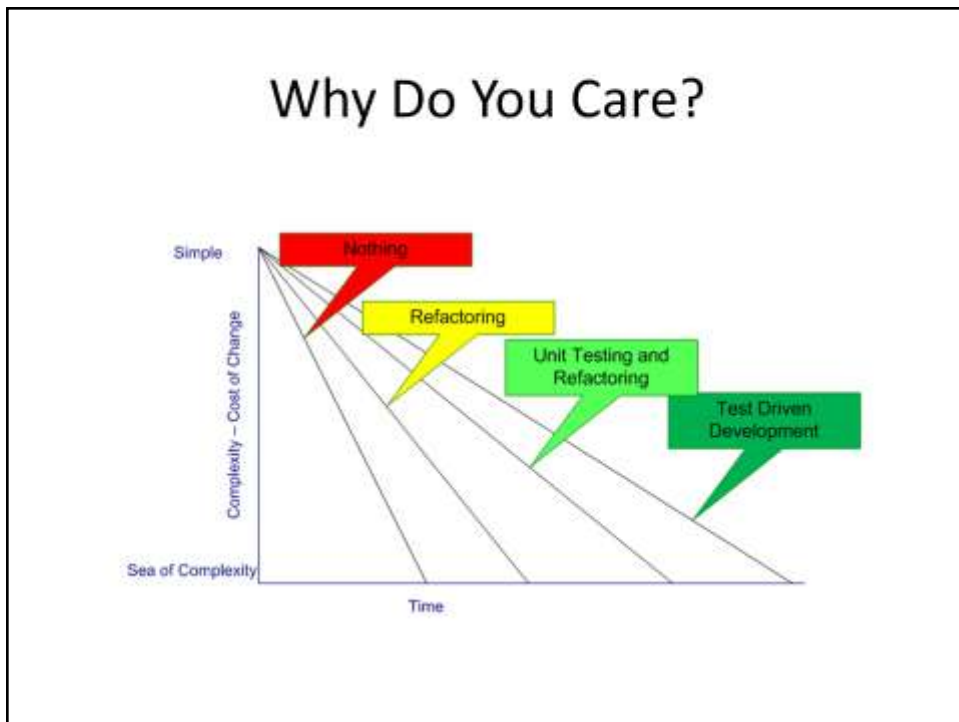
Many Agile teams achieve productivity improvements of 150-500% [ref: RPM Software, The Agile Impact Report: Proven Performance Metrics from the Agile Enterprise] the very best see 500-1000% [ref Jeff Sutherland Hyper Productive Teams] Yet your projects are seeing improvements in the 20-40% range. What happened?

It's all "those *internal* things that you choose not to do now, but which will impede future development if left undone" [Ward Cunningham]. On its surface the application looks to be of high quality and in good condition, these problems are hidden underneath. If this debt isn't managed and reduced the cost of writing/maintaining the code will eventually outweigh its value to customers.

Why does Tech Debt happen?

There isn't one big problem. Instead there are a myriad of small issues. Sometimes these were expedient short cuts, some changes where the developer didn't have time to tidy up, sometimes the developers just don't know the language and others are bits of code that have grown like a bush and need severe pruning. All of these amount to Technical Debt.

- What is Techincal Debt – quote from EDC paper. Separate slide, credit card picture and the metaphor
- Note its not all bad, it happens to everyone – the trick pay it down



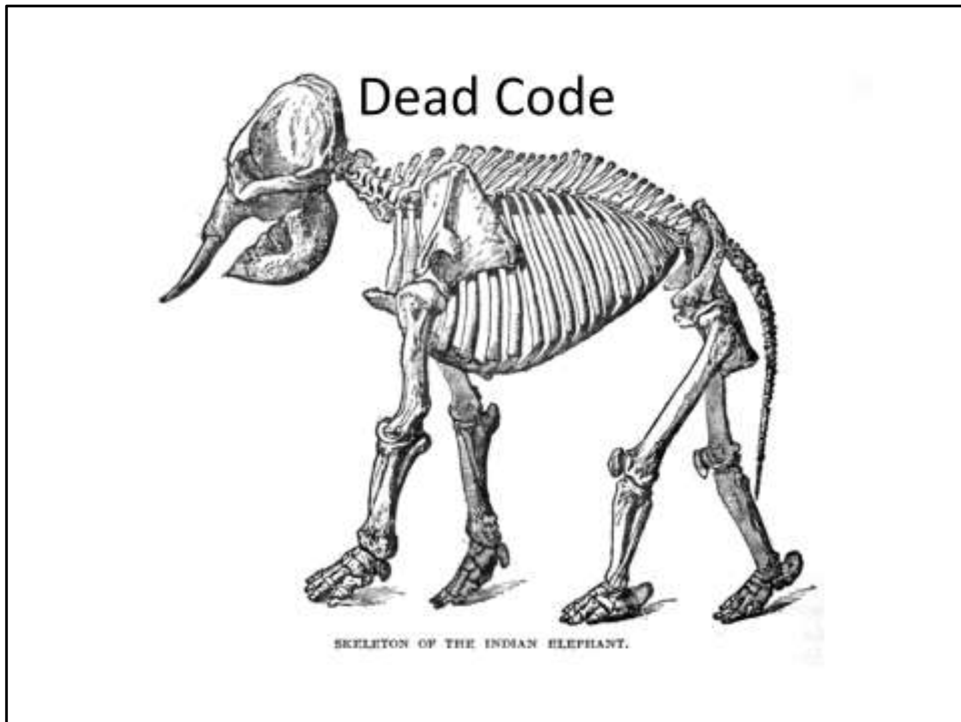
The “Sea of Complexity” is the point when the code base becomes difficult to manage where any one change cause a dozen other things to break.

High technical debt situations mean longer to respond to build new features and respond to market needs.

When your project reaches the “Sea of Complexity” its tough to make changes – all we’re trying to do is delay that day. Unit Testing is a just a way of pushing back the day when the code base is abandoned.

The sooner you jump tracks the better because its easier to change earlier on, while there is always an expense it’s a question of pay now vs. pay later. Do you pay the price of learning how to write good unit tests and simple code now? Or do you wait until your system is complex, brittle and tough to change. Now you have to learn not only unit testing but also the art of breaking up brittle code.

A language to describe the problems we see in code. Its just a way of describing the problems we find in our code.
Its easier to spot problems when you can articulate them. In addition the names give you some way of sharing information with your colleagues.



http://en.wikipedia.org/wiki/File:ElephantSkelLyd2.png

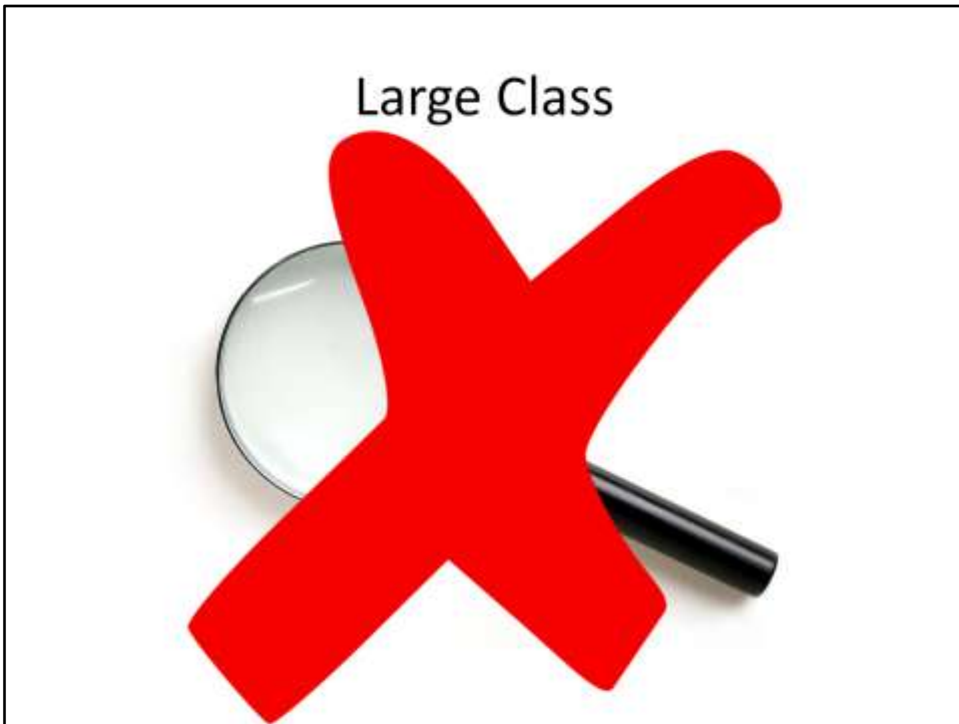
- What is it?
-Examples
- The private method that used to be
-– I frequently find that there are lots of extra constructors in a class and if you dig around some of them aren't being called. With a bit of extra juggling its often possible to reduce a class to a single constructor.
- How do you find dead code: Tools and a keen eye

Duplicate Code

- Duplicate
- Duplicate
- Duplicate
- Duplicate

- Copy & Paste
- Copy & Paste
- Copy & Paste
- Copy & Paste

-Obvious – same code fragment repeated over and over.
- Subtle sometimes all that differs is the variable names



More than one responsibility, more than a few fields, many public methods. More than one reason to change.

Long Method

Hints that your method is too long – can’t fit all on one screen. Method name contains and/or. Reads like a run on sentence.

Data Class

http://www.sxc.hu/photo/452873/

Dumb data container – just a bunch of setters and getters. Other people are reaching into this class and manipulating it. All of a sudden a bad value somewhere in your code – if passed through a Data Class how do you track it down? If a class owns some data then it should manipulate it. On a rare occasion you need to collate a bunch of parameters together aka Parameter object – but in that case the object doesn't need any setters and shouldn't live very long – i.e. just passed into one method. In fact I consider setter's a code smell of their own.

Conditional Complexity

```
if ((frequentRenterPoints > 5)
  && (totalAmount < 10)
  ||(each.getMovie().getPriceCode
  () == Movie.NEW_RELEASE)) {
```

With several if statements like this scattered through a method it quickly becomes difficult to follow.

Comments

Tools

- Java
  - PMD
  - FindBugs
  - Simian
- .NET
  - FxCop
  - NDepend

Discussion

- Sample Code
- Code Smells Cheat Sheet

Work in groups of 3-4. Walk through the sample code and discuss the problems you see.

Tell the audience some of the smells I expect that they will see

Smells:
CustomerTest – has lots of duplicate code especially init
Movie – has dead code in terms of the spare constructor
Movie and Rental are DataClasses
Customer – statement is full of comments
Customer.statement is a long method (too many things it tries to do)
Game and Customer are full of Conditional complexity

Deodorant?

"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." — *Refactoring*, Martin Fowler, page 53.

Make this quote float above a good picture

Its like Copy Editing – for your code

We do it to make future changes easier, to make the code easier to read. It happens in baby steps – small change and then test, small change then test. Examples – rename method is simple refactoring that can introduce clarity.

Refactoring preserve the existing behaviour they don't change it. Refactorings are small and quick. Sadly as refactoring has crept into our language people have taken it to cover much larger changes – i.e. a fellow team member says I need to spend the next two days refactoring the XYZ – that's too large to be one refactoring. Refactoring isn't a one off expense that will cure problems in the code once and for all. Refactoring is liking weeding the garden – ongoing maintenance is required to keep the chaos down to an acceptable level.

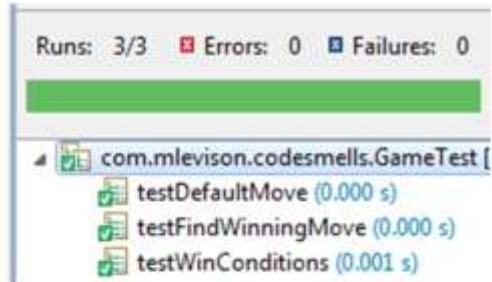
Or a developer on the team says this class is broken I need to refactor it. A refactoring doesn't change the externally visible behaviour of the code – so this developer isn't refactoring they're fixing a bug.

Where possible we only do refactorings on code that we have good unit tests for, the unit tests tell us when something is broken and we're not safe to check in. In addition the very act of writing tests for old code often causes us to discover dead code. We go to test it and wonder if its worth the effort, a few moments later with our IDE and we discover a dead method/class.

Our IDEs and their refactoring tools get better with every generation – when I started using eclipse the refactoring menu was only half a dozen items long. Now it has

Pair Refactoring

Unit Testing



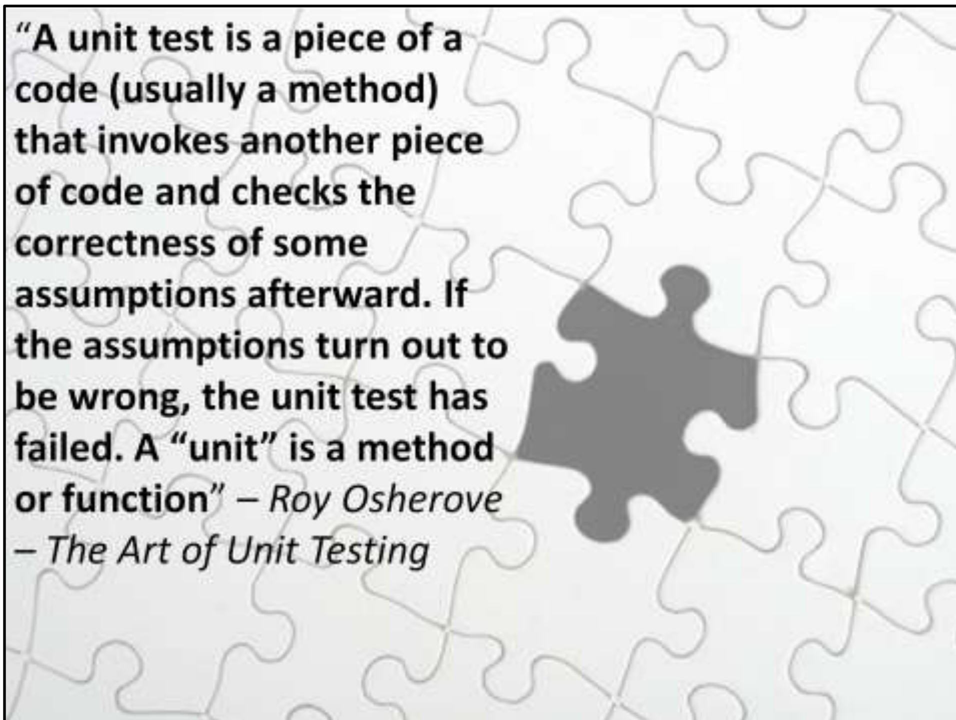
Before Unit Testing

- Code Lots
- Test Manually
- Brittle
- Tough to Find Bugs

http://www.sxc.hu/photo/485647/

Before Unit Testing. If you fix a bug (perhaps with another if statement) it may reappear without any warning. Complexity increases without anything but your own personal discipline and that of the developers around you.

•Code Lots - you spend a lot more time writing the code and much less thinking about why its written a particular way or how to test it
•Test Manually – is slow and repetitive, you don't do it often enough and humans easily to miss problems.
•Brittle – without Unit tests to force classes into smaller independent chunks small changes can have effects across the system
•Debugging – a lot of time is spent in the debugger trying to track down the location for the problem



http://www.everystockphoto.com/photo.php?imageId=1446019

One test class per class of application code. Draw two UML – one good, one bad. Show the good one and then the bad with an X through it.

Find a good quote – Fowler, Beck, Ron?

Then show a UML diagram that shows a simple method and several related test methods.

```java
public class GameTest {
  @Test
  public void testWinConditions() {
    Game game = new Game("---XXX--");
    assertEquals('X', game.winner());
  }
}
```

Highlight the bits that really matter – comment on the expected vs actual portion of the assert

Scope?

- Single Class
- Single Method
- Single Path
- One Concept

Its all about small focused tests. Each test tests one thing only.

How to Test

- Boundary Conditions
- Normal data
- Behaviours
- Exceptional Condition

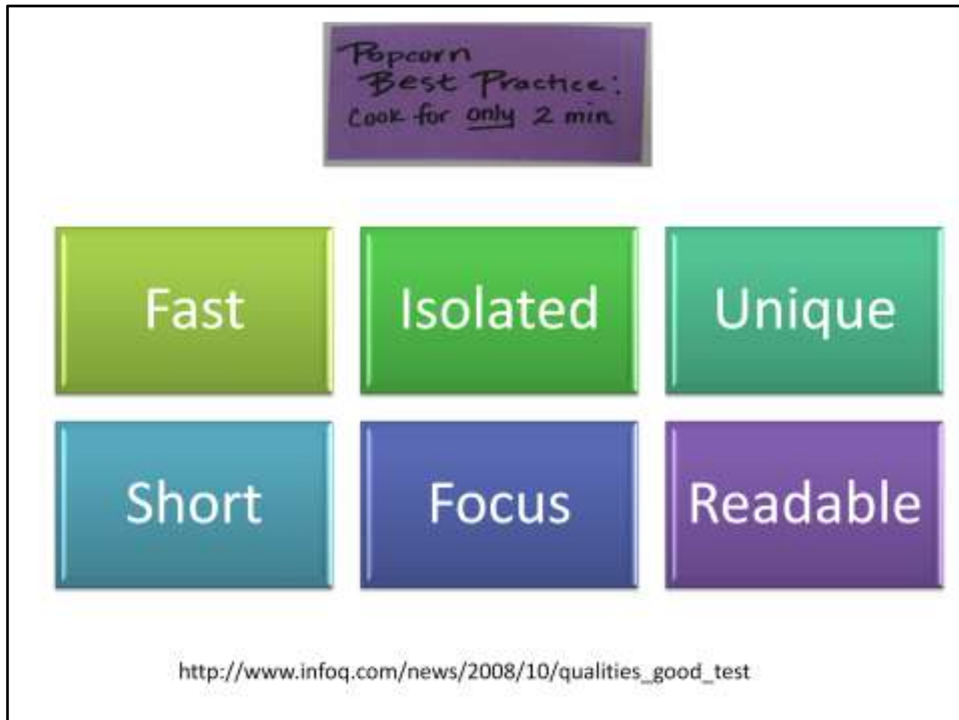
Give the example and ask the audience what the boundary conditions are in this case.

An example lets say we're going to web site owners $2 per 1000 clicks, we would want to test some boundary conditions
-1, 0, 1, 999, 1000, 1001 clicks. Maybe even -1 if this is the public API of the program.

I would also test some normal data – i.e. 423, 1333 clicks. Run of the mill stuff

Focus on testing the behaviours of the class under test not the data i.e. getters/setters.

Test our Existing Code



•Isolated (unaffected by the presence, absence, or results of other tests)
•Unique (providing confidence not provided by other tests/uncorrelated with other tests)
• It is short, typically under a dozen lines of code.
• It invokes only a tiny portion of the code, most usually a single branch of a single function. Asserts one concept
• Always pass – the more often you run the tests the easier it is to find problems with your code base. I don't like to go more than 5 minutes between test runs.
• Readable Intention Revealing - the best unit tests make it clear to the reader how an objects API is intended to be used
• Gateway to Commit – commit only when all tests pass
• It avoids most or all usage of 'awkward' collaborators via a variety of slip-and-fake techniques.
• For more see: http://www.infoq.com/news/2008/10/qualities_good_test

•**If Unit Testing doesn't change the way how your code looks then we have failed.**

Worst Practices

- Database
- File System
- Multiple Classes
- Manual Setup
- Order Dependancy

```
AppHelper.openModalDialog(t
AddressHelper.getAddressBoo
setList();
    }
} catch (Exception e) {
}
```

Good Tests run quickly, tests that access a database (on disk) or the filesystem directly will run slowly. If tests run slowly they will be run less often.

If it tests more than one class its not a Unit Test

If they must be run in a specific order to work. Its not a Unit Test.

Myths

- 100% Code Coverage
- Test every method

-Code coverage tells you what has been visited. Not what has been tested.
- Code coverage doesn't tell you whether assertions have been written to actually test the code.
- Code coverage doesn't tell you whether an if statement tested as valid for one all reasons. I.E. If 'A' or 'B' then…

    -Code coverage will only tell you that you visited the inside the If statement not which parts of the statement were really tested.

-Code coverage is only useful in conjunction with the human brain.
- Not everything is worth testing – why test that the getters/setters/simple constructors? If they're simple there maybe no value. If there is any complexity – read if statements, loops, … anything where you have made more than one mistake before. Test it.

Bad Test Code

```
1 [TestFixture]
2     public class UnitTest1 {
3         // Movies
4         Movie m_Cinderella;
5         …
22        [Setup] public void Init() {
23            // Create movies
24            m_Cinderella = new Movie("Cinderella",
 PriceCodes.Childrens);}
41
42        [Test]public void TestMovie()    {
44            // Test title property
45            Assertion.AssertEquals("Cinderella",
 m_Cinderella.title);
46            Assertion.AssertEquals("Star Wars",
 m_StarWars.title);
47            Assertion.AssertEquals("Gladiator",
 m_Gladiator.title);
48
49            // Test price code
50            Assertion.AssertEquals(PriceCodes.Chil
 drens, m_Cinderella.PriceCode);
51            …
53        }
60        [Test]
61        public void TestRental() {
62            // Test Movie property
63            Assertion.AssertEquals(m_Cinderella,
 m_Rental1.Movie);
```

- Hungarian Naming
- What Class is being Tested?
- Focus?
- Behaviour?

•UnitTest1 – doesn't name which class is under test – generally I like test classes to be named by adding the word test to the class under test.
•Focus – the test methods are testing a scattershot of functionality one class each i.e. TestMovie is testing everything in the movies class. There should be a Movie Test class which has methods like TestCinderellaGetPrice(), …. Which raises the bigger problem.
•Behaviour – the tests in the movie class are testing Data and not behaviour. Since generally there isn't a ton of value in testing constructors/getters (assuming they're simple – i.e. no if statements), these are not valuable tests. This leads to the code smell that as written the movie class is data container not real class. As such it should either find a purpose (i.e. real behaviour) or cease to exist.

…

```
73          [Test]
74          public void TestCustomer() {
83              m_MickeyMouse.AddRental(m_Rental3);
91
92              // Test the Statement() method
93              string theResult =
  m_MickeyMouse.Statement();
94    96              char[] deli
95              // Parse the result
miters = "\n\t".ToCharArray();
97              string[] results =
  theResult.Split(delimiters);
98
99              /* The results[] array will have the
  following elements:
100              *        [0] = junk
102              *        [2] = title #1
103              *        [3] = price #1
118             // Test the title and price items
119             Assertion.AssertEquals("Cinderella",
  results[2]);
120             Assertion.AssertEquals(4.5,
  Convert.ToDouble(results[3]));
121             Assertion.AssertEquals("Star Wars",
  results[5]);
122             Assertion.AssertEquals(6.5,
  Convert.ToDouble(results[6]));
123             Assertion.AssertEquals("Gladiator",
  results[8]);
124             Assertion.AssertEquals(15,
  Convert.ToDouble(results[9]));
125         }
```

- Size
- Comments
- Parsing
- Obscure Format
- Fragile

-This one test method is over 50 lines long, even the 20 displayed on screen is too much.

-Tests shouldn't need comments if its complex enough to need a comment its

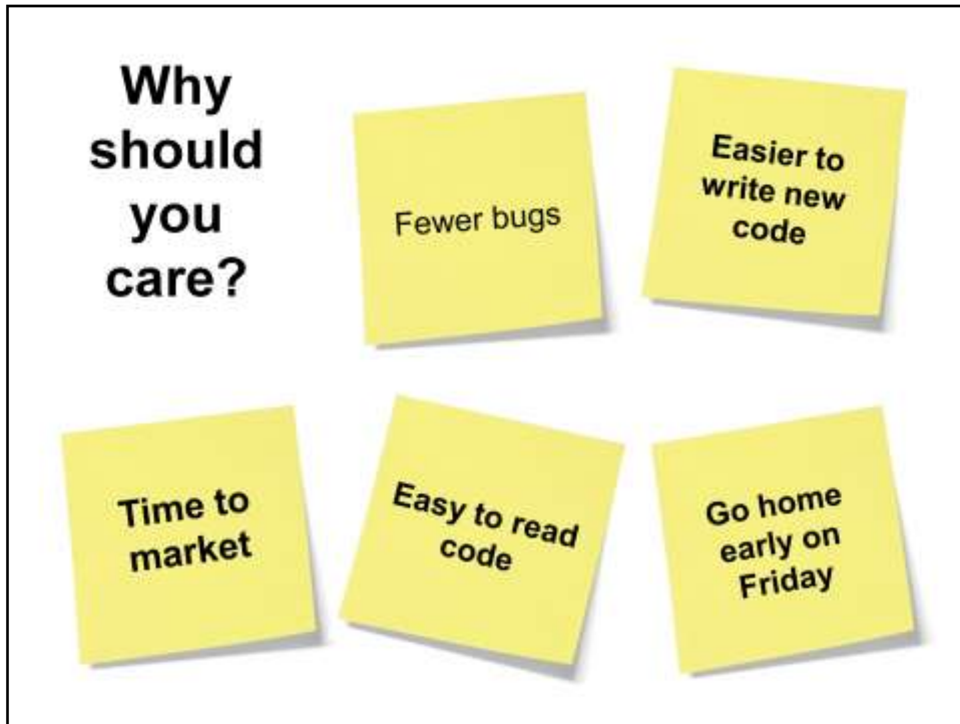
  -Too big

  - Fragile

  - Testing the wrong thing

- Parsing the output of a string is a really bad smell.

-Obscure Formats like the one shown are brittle and if you have multiple tests that rely on them then there will be many breakages when they change.



Linda – flies though the benefits.

CALL TO ACTION

Starting tomorrow

Search for: **"Code Smells",** Ask **"How to Refactor",** and Are there **"Unit Test".**

Create an **ACTION PLAN for** reducing your technical debt. 10 minutes at a time.

Pair with your co-workers - listen to them, learn from them

Mark talks

References

- The Art of Unit Testing: with Examples in .NET (Amazon.ca) – *Roy Osherove* (.NET and soon a Java version)
- Refactoring: Improving the Design of Existing Code (Amazon.ca) – *Martin Fowler*
- Refactoring to Patterns (Amazon.ca) – *Joshua Kerievsky*
- xUnit Test Patterns: Refactoring Test Code (Amazon.ca) - *Gerard Meszaros*
- Pragmatic Unit Testing in C# with NUnit (Amazon.ca) – *Andy Hunt and Dave Thomas*
- Pragmatic Unit Testing in Java with JUnit (Amazon.ca) – *Andy Hunt and Dave Thomas*

Mark Levison
Agile Change Consultant
mark@theagileconsortium.com
www.notesfromatooluser.com

THE
agileconsortium

http://www.istockphoto.com/stock-photo-4745224-striped-skunk.php