Agile Engineering Training

PARTICIPANT MANUAL + WORKBOOK

Presented by

Leanintuit

practical support for agile change

Technical Debt

Definition of Technical Debt

The *technical debt* metaphor has become common in our industry.

Here is the wikipedia definition:

"A concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution – Wikipedia"

The metaphor was created by Ward Cunningham, the inventor of the wiki, and one of the creators of *extreme programming*.

Here is Ward's definition:

"When taking short cuts and delivering code that is not quite right for the programming task of the moment, a development team incurs Technical Debt. This debt decreases productivity. This loss of productivity is the interest of the Technical Debt – Ward Cunningham"

Our definition is much simpler:
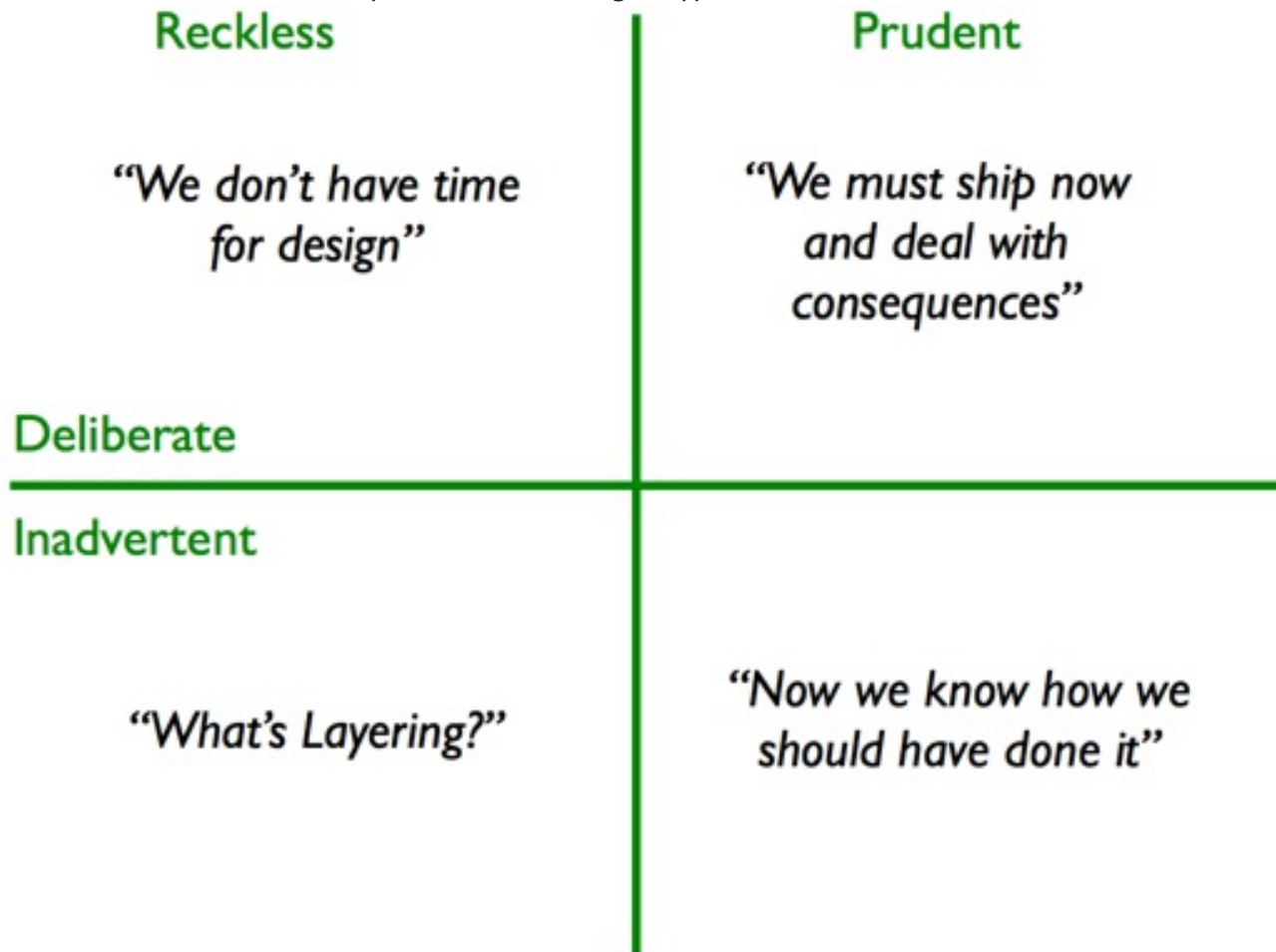*"Technical debt is anything that slows you down – Declan Whelan"*

The Problem With This Metaphor

We have seen the technical debt metaphor used to *justify* the creation of technical debt! The argument goes like this: "We really need to get this feature out to make this deadline! So we will create some technical debt now in order to make the date, and fix it later."

Of course after you deliver the debt-ridden code, there's another deadline, and another deadline, and you never go back and pay off the debt.

Technical Debt Quadrants

Martin Fowler has created a quadrant for looking at types of technical debt:

Reckless   **Prudent**

"We don't have time for design"

"We must ship now and deal with consequences"

Deliberate

Inadvertent

"What's Layering?"

"Now we know how we should have done it"

Source: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

The top half is *deliberate*, meaning that we choose to take on the debt. The bottom half is *inadvertent*, meaning we can't avoid it.

The Left side is *reckless*, and the right side is *prudent*.

Ward's original definition of *technical debt* referred to the *prudent/inadvertent* quadrant. It is inevitable that we create some technical debt as we write code. It becomes prudent if we go back and fix the problems afterwards!

Questions

What are some examples of technical debt (things that slow you down)?

How do you decide if taking on deliberate technical debt is prudent or reckless?

> *(blank box)*

What are some arguments we can make when our business people insist that we continually need to take on technical debt to make deadlines?

> *(blank box)*

What are some examples you have seen of these types of technical debt?

Reckless/Deliberate

Prudent/Deliberate

Reckless/Inadvertent

Prudent/Inadvertent

Quality Payoff

Our Hypothesis About Quality

We believe that producing quality software pays off eventually.

The Story of an Application

One of us (Shawn) worked on an application that he was really proud of.

The original application was developed with quality in mind. The small team worked together with business, controlled their own production environment, used continuous integration, wrote automated unit tests, and continuously improved the design and processes. (It was a little like Agile, although we didn't know the word at the time.)

The application provided great business value, and was very well received. In fact, it became so popular that the development team grew from three to forty.

As the group grew something happened. To make it easier for teams to work independently they started working on long-lived branches (months). Teams stopped writing new tests, and the existing tests all broke. It was impossible to improve the code because it made merges painful, so teams stopped improving the code. In order to work independently teams copied chunks of code into their own modules so that they could work in isolation.

After a few years it became extremely hard to develop new features. Bureaucracy and control was added to try to improve things. Deadlines tightened. Outsourced teams were added to increase productivity. This accelerated the decline in quality.

Eventually the decision was made to rewrite the application.

Six new application teams were created to write a replacement app. A lot of money was being spent, so the teams worked against tight deadlines. Off course, the rewrite needed to be kept up with the latest business changes, so it was a moving target. And the existing application still need to be maintained, so changes were made to two systems during the rewrite.

After a couple of years the new application was complete! The old application was decommissioned (partly -- there was still some functionality that wasn't moved to the new app for a couple of years).

Business and users were less than happy with the new app. It was different from the old one, so a lot of retraining was necessary. There were some ways that it was better than the old app, but some ways it was worse, and you better believe business and users focused on those deficiencies.

Also business wasn't happy with how long it took to add new functionality to the new system. It seems that the tight deadlines meant that IT took short-cuts with the design and quality of the new application.

After just a few years the decision was made to rewrite the new application....

The Rewrite Trap

The story above is, unfortunately, not unusual. Many developers you speak to have similar stories of an application being allowed to degrade in quality, until there seems to be no choice but a rewrite. In fact, we have a special word for this type of system: legacy.

Rewrites are *always* worse than you predict. We have seen this pattern many time. Often the rewrite is a partial success (although over budget), but sometimes it fails completely as you realize that the job is way larger than you had predicted. We have witnessed a rewrite cancelled after spending tens of millions, with nothing to show for the money.

We will talk about improving a legacy system later in this course. But what if it was possible to create and maintain an application so that it never became legacy?

Design Payoff Line

Take a look at the pseudo-graph below, from Martin Fowler. It represents the delivery of features over time.

The gold line represents an application where continual attention is paid to quality. (Fowler uses the term "design", but we believe this applies to other aspects of quality as well).
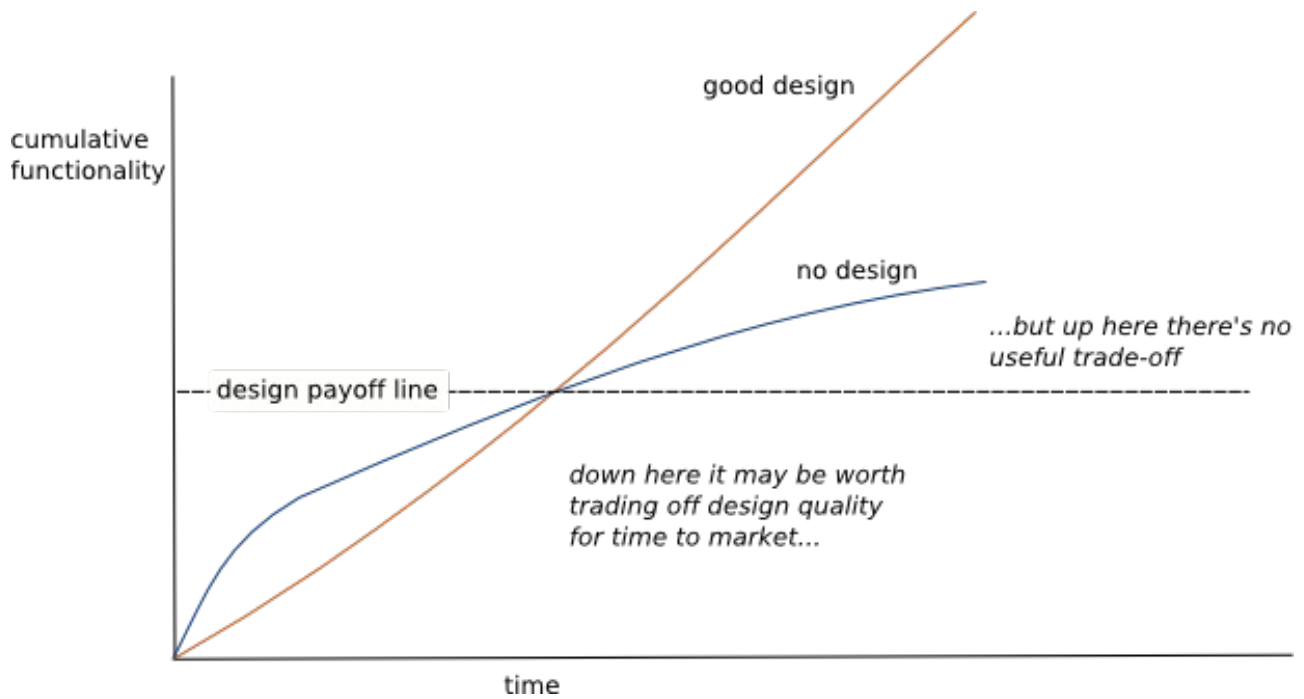
The blue line is an applications developed without attention to quality. It starts faster than the "quality" line, but over time the rate of delivery goes down as technical debt increases ("anything that slows you down").

The interesting point is the *design payoff line*. After this time the "good design" application has delivered more functionality, and continues to deliver faster, then the "no design" line.

Prior to the design payoff line it may be worth reducing quality in order to get quick time to market.

This is the essence of the *Deliberate/Prudent* technical debt. If we need to deliver to make a *real* deadline (such as a regulatory requirement, or a potentially lost market opportunity) perhaps it makes sense to me quality in the short-term. And pay off the debt afterwards.

Most often when we see people compromise quality in order to

good design

cumulative
functionality

no design

...but up here there's no
useful trade-off

design payoff line

down here it may be worth
trading off design quality
for time to market...

time

Source: https://martinfowler.com/bliki/DesignStaminaHypothesis.html

There comes a time where the total amount of functionality delivered with a quality application exceeds the
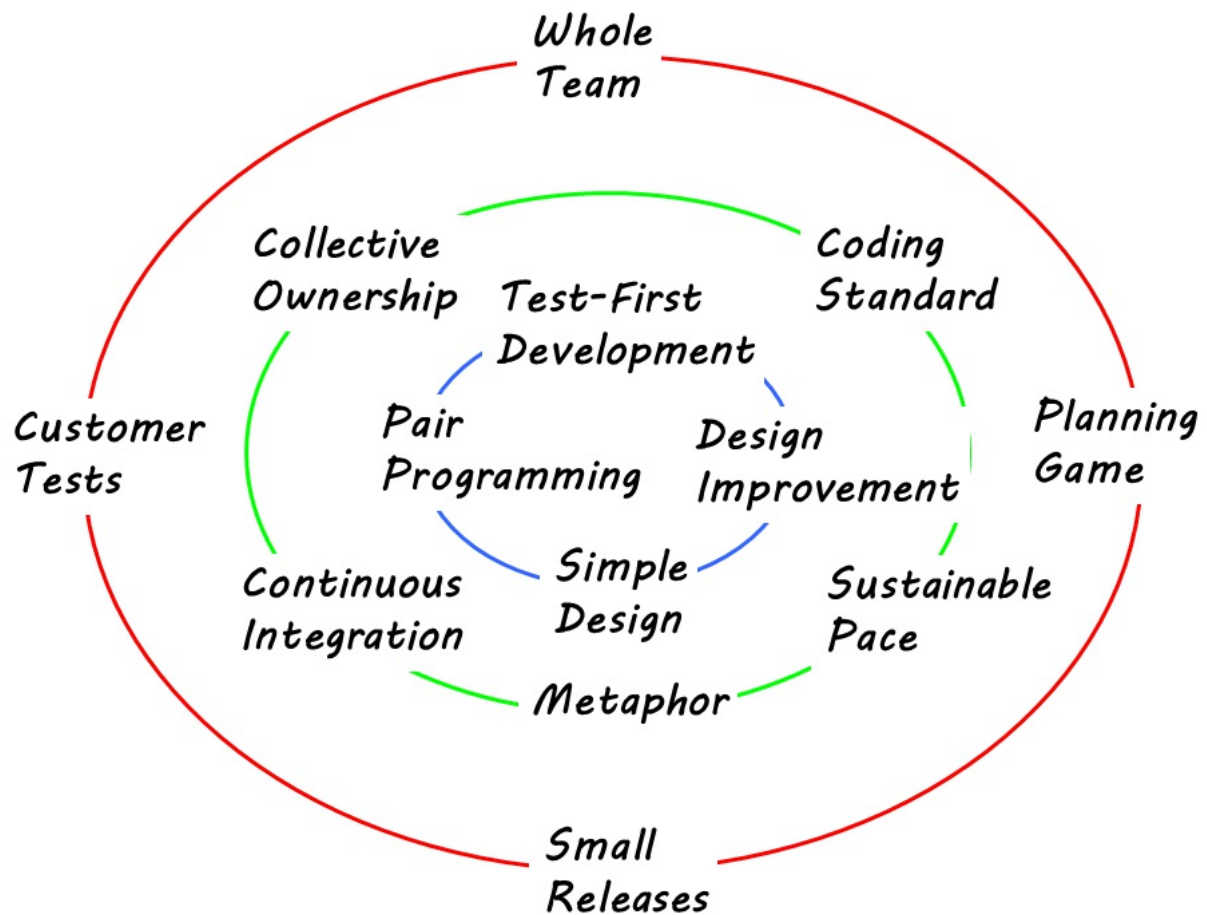
Application Stewardship

We like the term *application stewardship* for the

Questions

something?

Extreme Programming

History of XP

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. It was invented in the late 90s by Kent Beck (and others).

It is similar to Scrum, and has largely been displaced by it as a development methodology.

XP Practices

XP differs from Scrum in that prescribes specific development practices. Test-Driven Development (TDD), Pair Programming, Simple Design, Continuous Design Improvement (Refactoring) and Continuous Integration are all parts of the XP methodology.

Scrum chooses not to include specific practices (which probably has helped its adoption), but Scrum teams quickly discover that they will not be successful without paying attention to practices. XP practices are mandated as part of other methodologies, such as Scaled Agile Framework (SAFe).

XP / Devops / Continuous Delivery

Although Extreme Programming is not frequently practiced, the practices live on! We consider The Continuous Delivery and Devops movements to be built on the evolution of XP practices. Frequent integration of developers' work has evolved to frequent delivery right to customers! This has necessitated even more attention to XP practices such as keeping your code and design clean, automated testing at every level, and frequent customer interaction.

Agile Testing

Testing Quadrants

To help bring clarity to the complicated subject of testing Brian Merrick created the "Testing Quadrants."
He has separated types of testing into four domains, as in the diagram below.
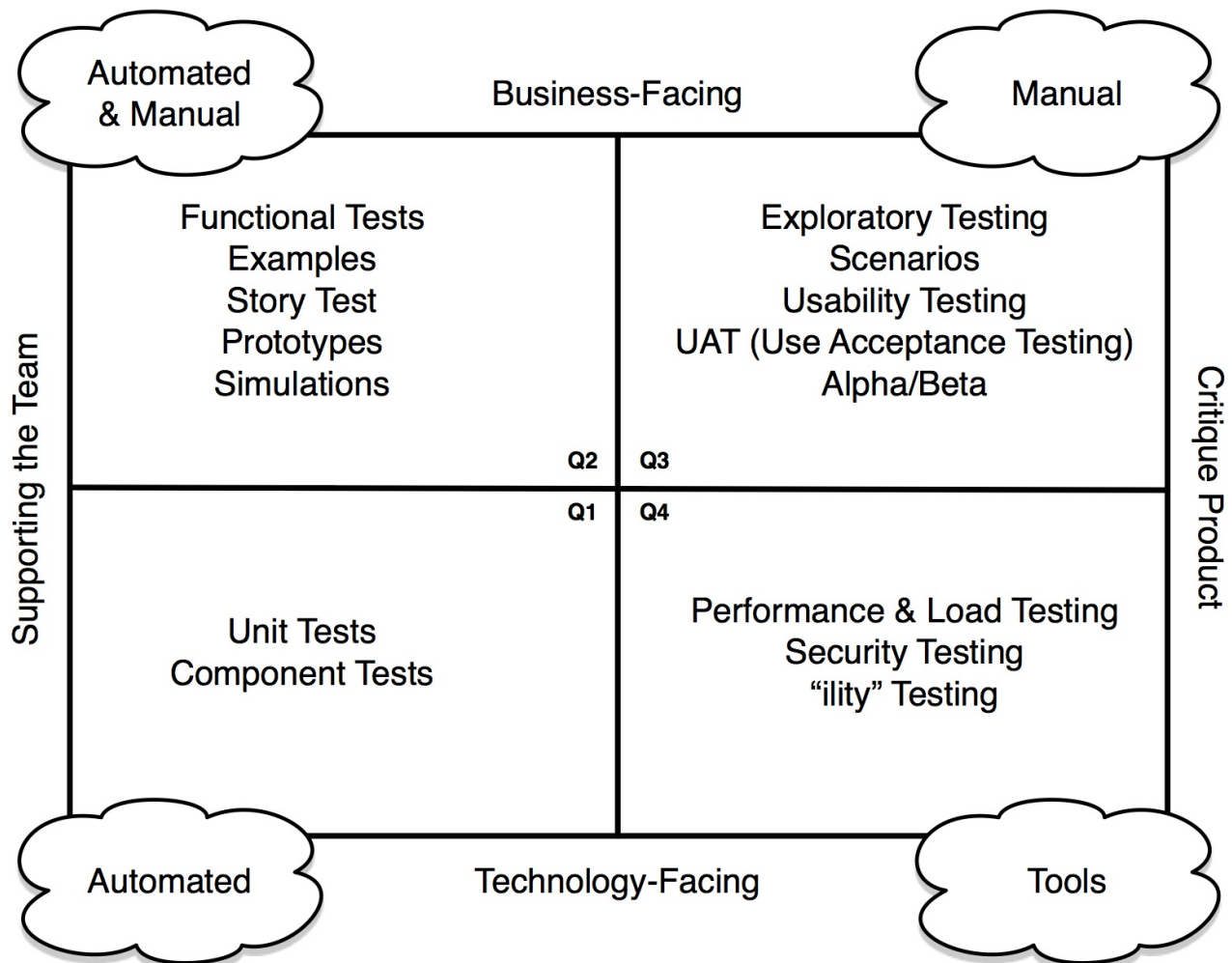
Tests on the left *support the team* during development. They are used to help drive development of the product.

Tests on the right *critique the product*. They are used to verify that the product meets, and continues to meet, the requirements.

Tests on the top are *business facing*. Business can understand them, and should be interested in them.

Tests on the bottom are *product facing*. They test the product, but might not be interesting for non-technical business folks.

Bottom-Left - Technology-Facing Supporting Development

This is the domain of automated unit and component tests. Your business is probably not interested in seeing these tests (although they love the confidence that comes from knowing you are writing them). They support the team because they allow the team to quickly evolve and new features to the code.

In particular Test-Driven Development (TDD) is in this quadrant. Writing code using TDD helps you think through the requirements in an incremental fashion, and having the TDD tests allows to you refactor continuously to keep your code clean to allow you to move fast.

Top-Left – Biz Facing Supporting Development

These tests help you to clarify your understanding of the product with business. Processes like Acceptance-Test-Driven-Development (ATDD) use the clarity that comes from writing tests collaboratively to ensure that there is alignment between business and developers. You might automated some of these tests to create an automated regression suite run as part of your pipeline, but some might remain manual.

Bottom-Right – Technology-Facing Critiquing The Product

These tests verify that the product meets non-functional requirements such as performance, load and security. Business people probably don't want to see these tests, but, again, love to hear that you are running them. Often people will use tools to perform these tests, but this introduces an element of manual effort. Continuous Delivery requires that you include these tests as part of your pipeline.

Top-Right – Business-Facing Critiquing The Product

These tests verify that the product does what it should from a business and user perspective. Often manual, these tests use human intelligence to ensure the product is as good as we can create.

Questions

In which quadrant would you put these tests?

a. Unit tests written after the code is complete

b. A/B tests checking the usage of various versions of a web page in production

c. Integration tests checking that your application correctly calls a database

d. Tests ensuring that the contact of your web service is correct

```
a.

b.

c.

d.
```

2. What would need to change with how you test in order to do Continuous Delivery?

```
                                                                        
                                                                        
                                                                        
                                                                        
                                                                        
                                                                        
                                                                        
```

3. Some people in the testing community prefer the team **checks** for automated tests,
    reserving the term **tests** for tests done by a human. Why do you think they make this
    distinction?

```
                                                                        
                                                                        
                                                                        
                                                                        
                                                                        
                                                                        
```

Types of Automated Tests

Any automated testing strategy will require the use of more than one type of test. For example, unit, component, integration, How do you decide which type of test and how many of them you need?

Testing Pyramid

Mike Cohn created the *automated testing pyramid* to illustrate the types of automated tests you might use:
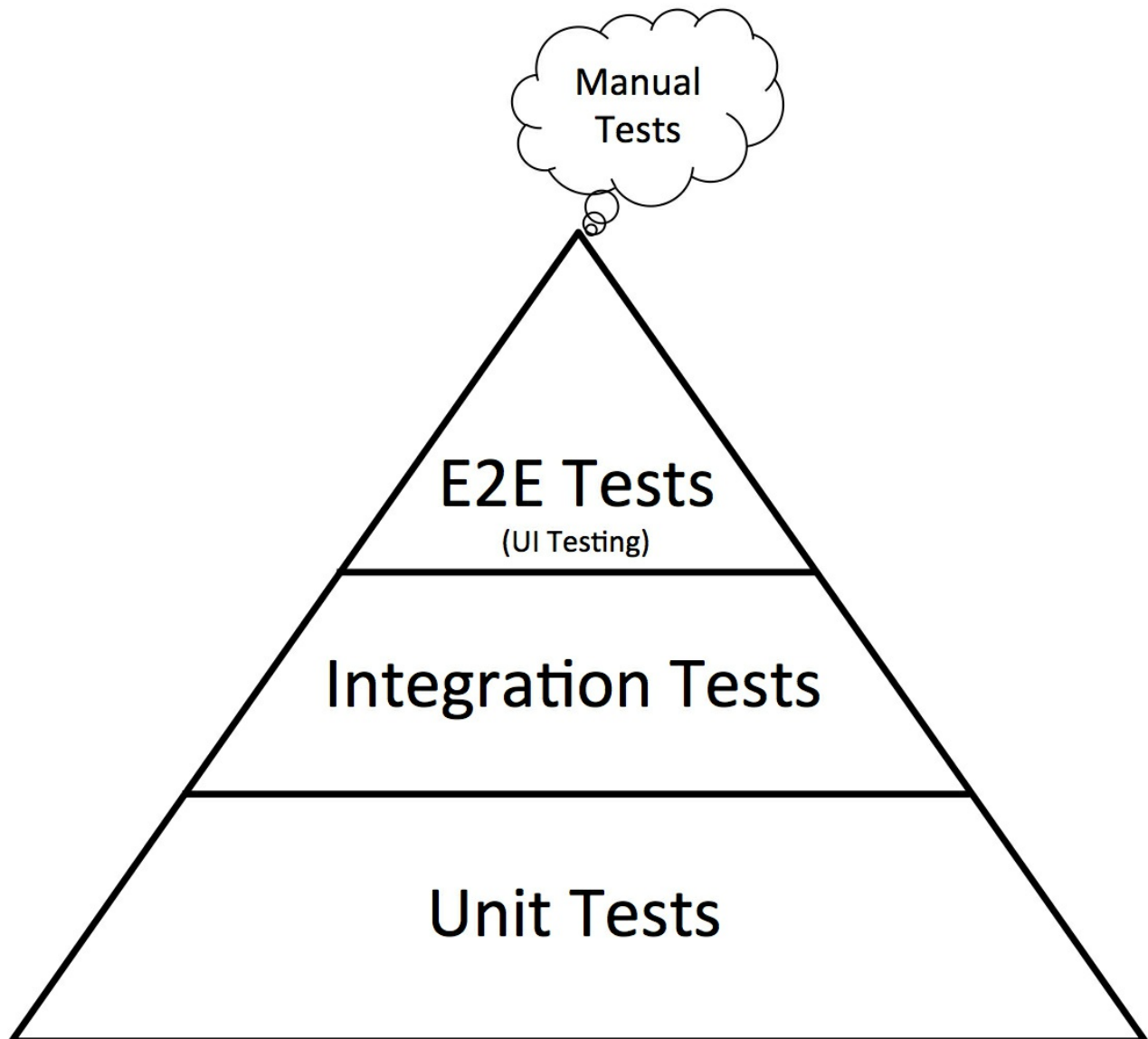
The Inverted Test Pyramid Anti-pattern

It is drawn as a pyramid to illustrate the relative number of tests you should write. There should be a lot more of the tests on the bottom (unit tests) than of the tests on the top (E2E tests). We generally see an order of magnitude in difference for each of the levels. For example if you have 10 E2E tests you might have 100 integration tests and 1000 unit tests.

A common testing anti-pattern is the *inverted test pyramid*. This is when you create a lot of E2E tests, while having few or no tests at the lower levels.

This is common when automated testing is done by an external testing group. Since they don't have access to the code they have no choice but to write end-to-end tests.

Why End-To-End Tests Are A Trap

Why is the *inverted test pyramid* an anti-pattern? It seems to be a way to create a good regression suite, and it makes sense to testers who are used to doing all of their testing through the customer-facing interface. It allows testers to create a regression suite without slowing down developers by asking them to help write tests.

Unfortunately there are some problems:

1. Data management is difficult. Often we don't have a dedicated environment so we run into data contention with other testing. Even if we have control over our data it is difficult to and slow to load the data for our tests.

2. Test failures are common and hard to diagnose. A complex system with many moving parts means that there are lots of places it could fail, and tracking down the failure requires searching through logs to track down the problem.

3. Tests are hard to write. E2E tests generally require a lot of set up. In addition you run into combinatorial problems. A test scenario with n steps with only two choices at each would result in 2^n possible test runs!

4. Tests are slow. E2E tests generally run many orders of magnitude slower than unit tests. To get complete test coverage could take many hours.

Moving Testing To A Lower Level

So what's the solutions? Whenever possible move your tests down to a lower level in the pyramid. For example, rather than E2E tests through a GUI, could you perhaps write tests against a service?

Could you write one E2E test for the happy case, and one failure test, just to ensure everything is working, and then cover the rest of the logic through unit tests?

Rather than writing integration tests, could you write one to make sure the components integrate, and then write unit tests to cover the majority of the logic?

Whenever you write a test ask yourself *"Is there a way I could write this test at a lower level?"*

Testing Is A Continuum

The picture above shows three layers of testing: E2E, Integration and Unit tests. In practice when you are creating your test suite (and moving tests lower!) you might find that the tests fall into many different levels, and the terms no longer apply.

For example, let's say you have an app that is composed of an Angular front-end, a Java services app with some business logic that calls back-end legacy systems.

In each of the two tiers you will have unit tests that tightly test your logic, component or service tests that test your higher level services, and integration tests that ensure your app integrates with its dependancies. In fact for the integration tests you might have a couple of types: ones that call

an actual dependancy, and ones that call a stubbed back-end.

You're likely going to create some tests that make sure your two apps are integrating, and to move those tests lower you'll want to stub out the legacy system.

You'll also want a few tests that call the actual legacy system, true end-to-end. Even with these tests you might stub out portions of your legacy if it is very hard to write tests for portions.

So, when you write your tests be prepared to get creative. Don't fall into the trap of just writing end-to-end through the GUI.

Testing Done By Entire Team

Above we mentioned that sometimes the *inverted test pyramid* is a consequence of the testing done by an external test team.
Creating a good automated test suite requires the entire team to be involved. Testers understand what to test, and developers understand how to keep a code-base clean, and how to create a framework to support quickly creating tests. Also developers who write tests will ensure that their app is testable!

Questions

What are some reasons that testing using End-to-end tests is expensive? Why might it be more difficult than testing at lower levels?

What are some ways that people of every role can contribute to automated testing?

```
Developers:



Testers:



Architects:



BA/BSAs:



```

What would have to change on your team for testing to become a *whole team* responsibility?

```



```

Fizzbuzz

Background

Fizzbuzz is a group word game for children to teach them about division.

Players take turns to count incrementally, replacing any number divisible by 3 with the word "fizz", any number divisible by 5 with the word "buzz", and any number divisible by 3 and 5 with the word "fizzbuzz".

Kata

Create a `Fizzbuzz` class with a `valueFor(number : int) : string` method.

It should return a string according to the rules of Fizzbuzz:

```
game = new FizzBuzz();

game.valueFor(1);  // "1"
game.valueFor(2);  // "2"
game.valueFor(3);  // "fizz"
game.valueFor(4);  // "4"
game.valueFor(5);  // "buzz"
game.valueFor(6);  // "fizz"
...
game.valueFor(15); // "fizzbuzz"
```

Players take turns to count incrementally, replacing any number divisible by 3 with the word "fizz", any number divisible by 5 with the word "buzz", and any number divisible by 3 and 5 with the word "fizzbuzz".

Bonus Variation

Also return "fizz" if the number contains the digit "3" and "buzz" if the number contains digit "5":

```
game.valueFor(1);  // "1"
game.valueFor(2);  // "2"
game.valueFor(3);  // "fizz"
game.valueFor(4);  // "4"
game.valueFor(5);  // "buzz"
game.valueFor(6);  // "fizz"
...
game.valueFor(13); // "fizz"
game.valueFor(14); // "14"
game.valueFor(15); // "fizzbuzz"
...
game.valueFor(20); // "buzz"
game.valueFor(21); // "fizz"
game.valueFor(22); // "22"
game.valueFor(23); // "fizz"
...
game.valueFor(30); // "fizzbuzz"
```

Roman Numeral Calculator

Background

The Romans used a special method of showing numbers, based on the following symbols:

Arabic Roman
1   I
5   V
10   X
50   L
100 C
500 D
1000 M

Forming Numbers

Use the following table to look up the corresponding roman numerals and concatenate the results:

	x1	x2	x3	x4	x5	x6	x7	x8	x9
Ones	I	II	III	IV	V	VI	VII	VIII	IX
Tens	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
Hundreds	C	CC	CCC	CD	D	DC	DCC	DCCC	CM
Thousands	M	MM	MMM						

For example:

1984 => 1000 + 900 + 80 + 4 => "M" + "CM" + "LXXX" + "IV" => "MCMLXXXIV"

Kata

Create a methd `calculate(input : int) : string` method that converts the input arabic number to its roman numeral equivalent.

Some examples:

Input Output
1   "I"
9   "IX"
1984 "MCMLXXXIV"

Bowling

Background

The game consists of 10 frames. In each frame the player has two opportunities to knock down 10 pins. The score for the frame is the total number of pins knocked down, plus bonuses for strikes and spares.

A spare is when the player knocks down all 10 pins in two tries. The bonus for that frame is the number of pins knocked down by the next roll.

A strike is when the player knocks down all 10 pins on his first try. The bonus for that frame is the value of the next two balls rolled.

In the tenth frame a player who rolls a spare or strike is allowed to roll the extra balls to complete the frame. However no more than three balls can be rolled in the tenth frame.

Kata

Write two methods: `roll()` and `score() : int`.

`roll(pins : int)` is called each time the player rolls a ball. The argument is the number of pins knocked down.

`score() : int` is called only at the very end of the game. It returns the total score for the game.

Gilded Rose Kata

Hi and welcome to team Gilded Rose. As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date.

We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures.

Your task is to add the new feature to our system so that we can begin selling a new category of items. First an introduction to our system:

```
- All items have a SellIn value which denotes the number of days we have to
sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item
```

Pretty simple, right? Well this is where it gets interesting:

```
- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never increases or decreases in Quality
or SellIn vaulue.
- "Backstage passes", like aged brie, increases in Quality as its SellIn value
approaches; Quality increases by 2 when there are 10 days or less and by 3 when
there are 5 days or less but drops to 0 after the concert is over.
```

We have recently signed a supplier of conjured items. This requires an update to our system:

```
- "Conjured" items degrade in Quality twice as fast as normal items
```

Feel free to make any changes to the UpdateQuality method and add any new code as long as everything still works correctly. However, do not alter the Item class or Items property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership (you can make the UpdateQuality method and Items property static if you like, we'll cover for you).

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.