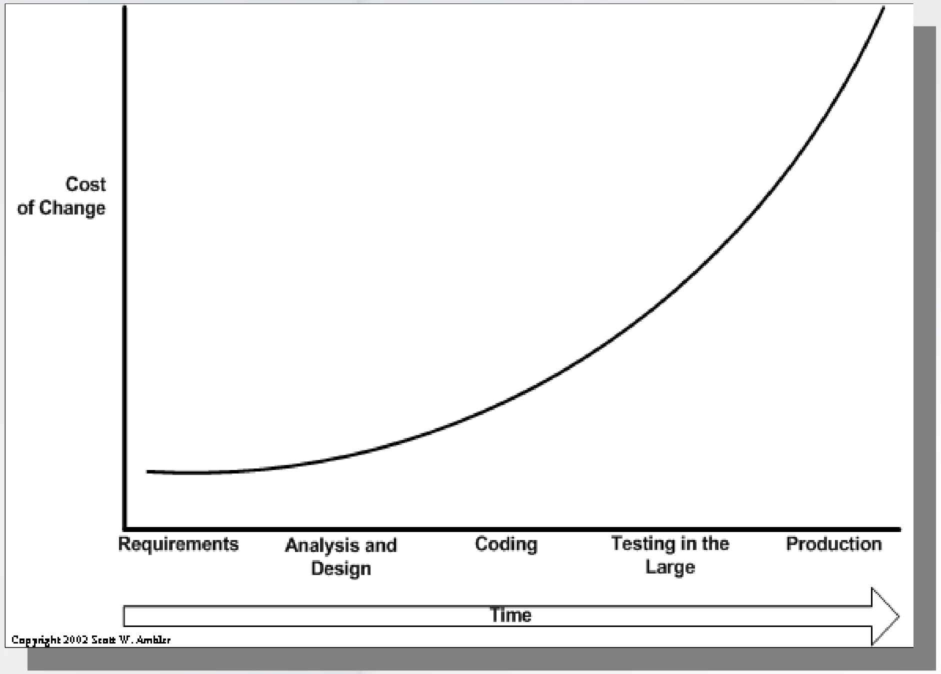
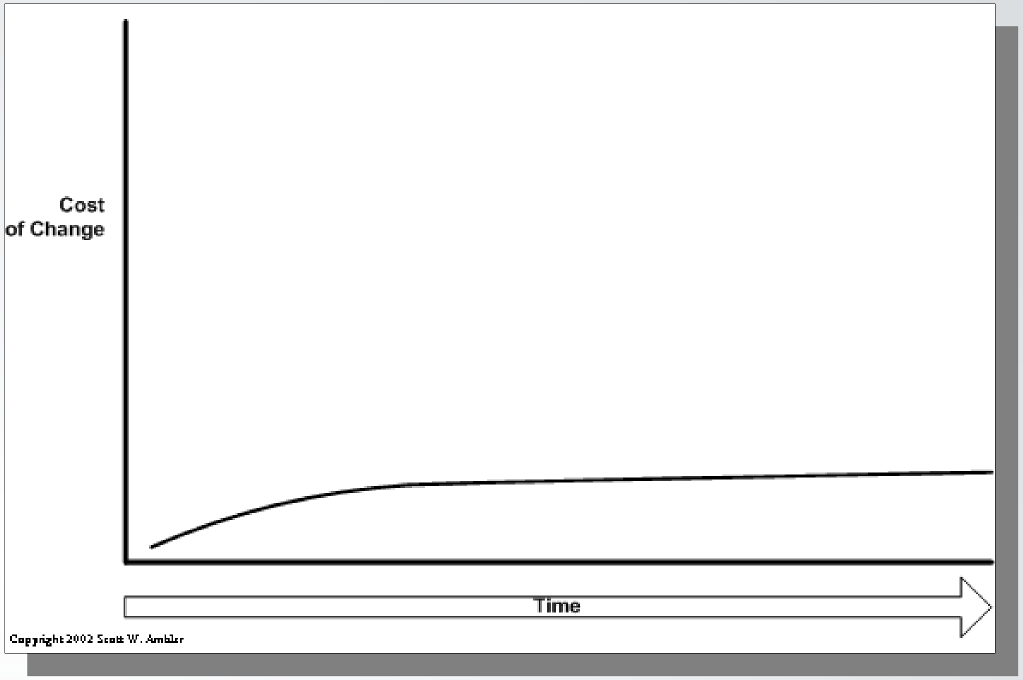
Peer Group Learning & Assessment For Software Craftsmanship

Jason Gorman

Cost Of Change

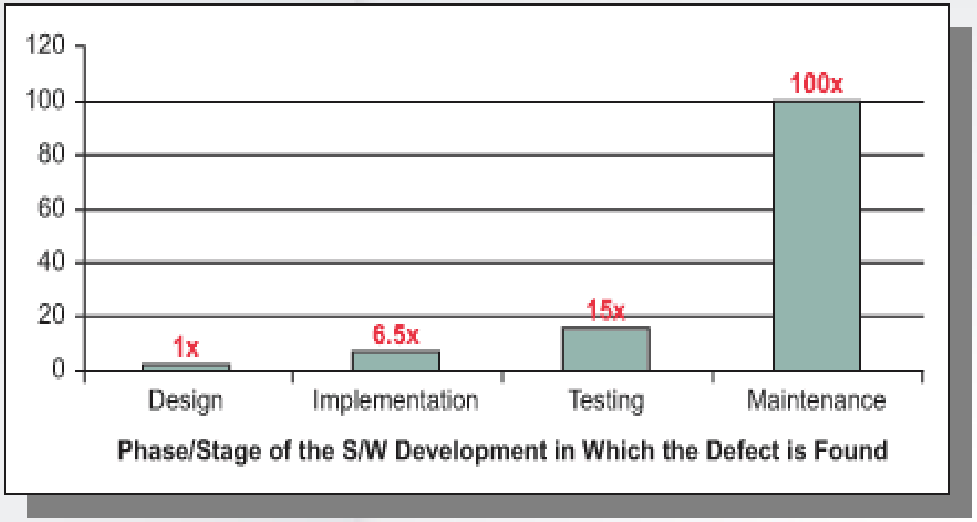


Source:Scott W. Ambler, agilemodeling.com

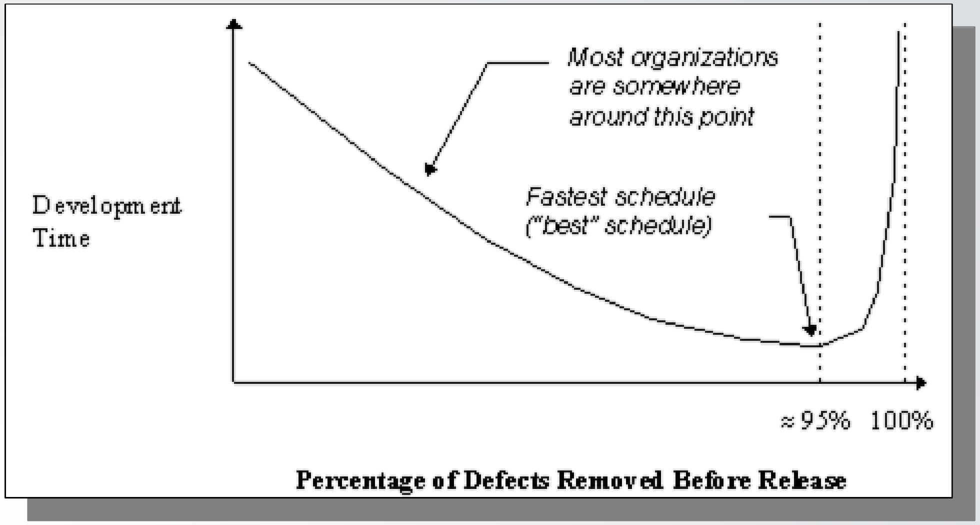


Source: Kent Beck, Extreme Programming Explained

Economics of Defect Prevention



Source: IBM System Sciences



Source: Steve McConnell, Software Quality At Top Speed

What Makes Code Harder To Change?

Readability

Complexity

Duplication

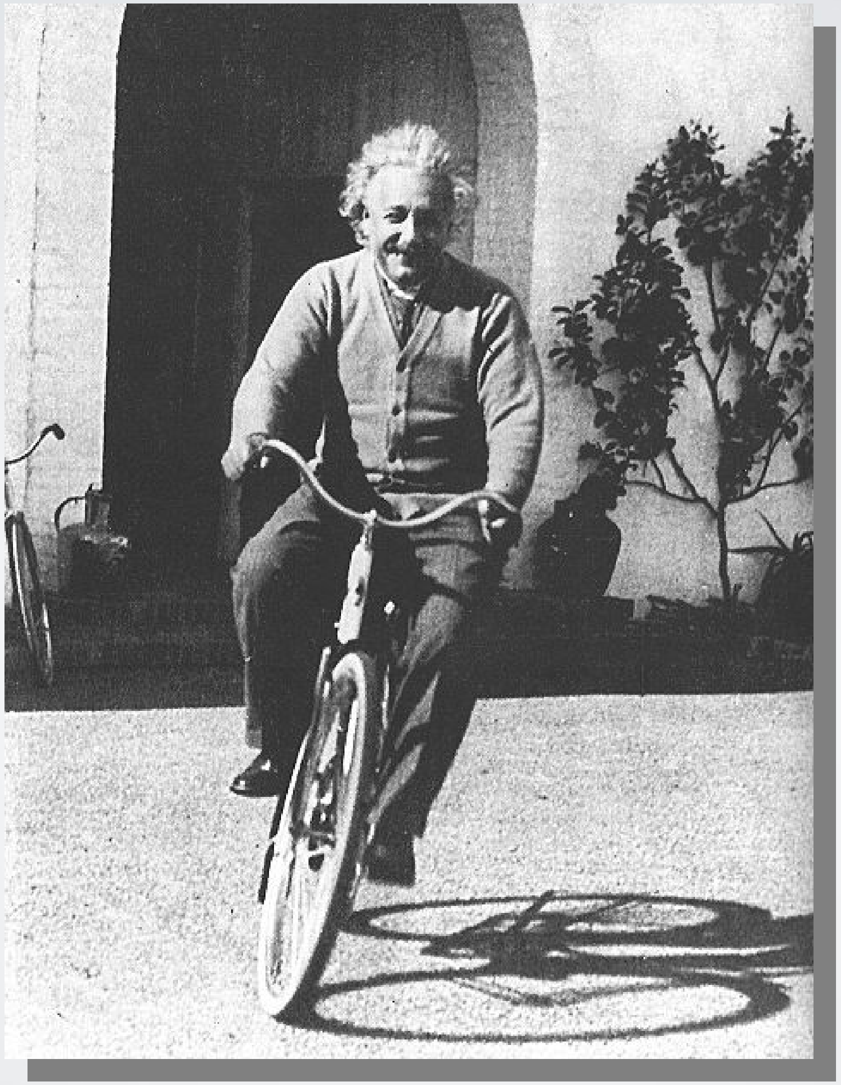
Dependencies & The “Ripple Effect”

Regression Test Assurance

Peer Group Learning & Assessment

Teaching vs. Learning

Knowledge vs. Ability



How Do You Get To Carnegie Hall?

How Do You Know If A Juggler Can Juggle?

Orientation - Peers Agree Good Habits

Test should fail first time it's written/run

Refactor to remove duplicate code after passing test

Test names should reflect intent, and names should be expressive

Re-run tests after every refactoring

Only write new code when a test is failing, each test should test new/different behaviour

Practice Regimen Builds Habits

Test should fail first time it's written/run	JG 23/2/10								
Test names should reflect intent, and names should be expressive	KJ 12/2/10								
Refactor to remove duplicate code after passing test	JG 23/2/10								
Re-run tests after every refactoring	KJ 12/2/10								
Only write new code when a test is failing, each test should test new/different behaviour	JG 23/2/10								
Write the assertion first	JG 23/2/10								
Minimise assertions in each test	KJ 12/2/10								
All tests should pass before writing the next test									
Only refactor when all tests are passing	KJ 12/2/10								
Write the simplest code to pass the test									
Have separate source and test folders, test code should follow structure of src									
Don't introduce dependencies between tests. Test should pass when run in any order.									

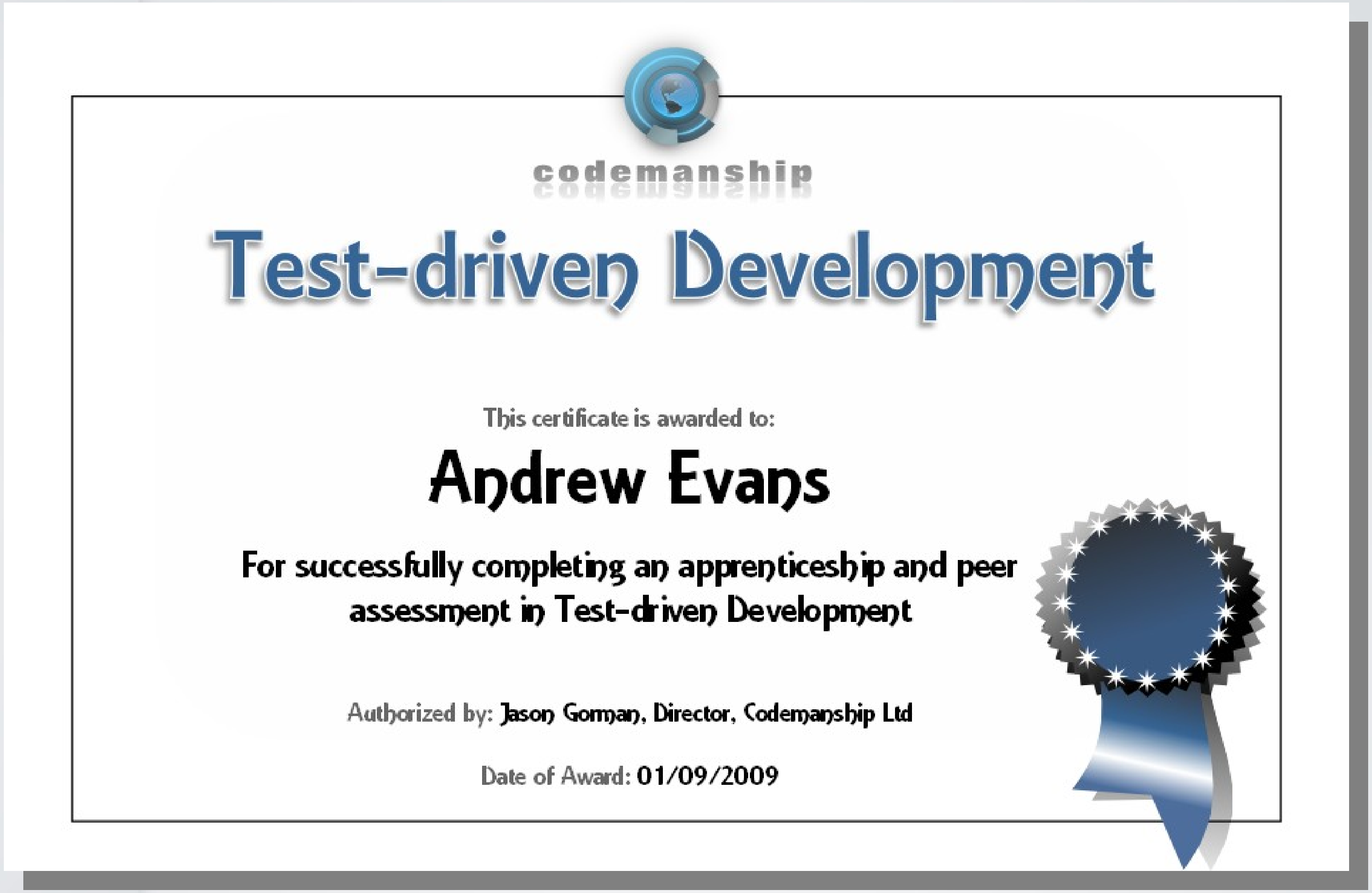
Assessments Test Habits

Test should fail first time it's written/run									
Test names should reflect intent, and names should be expressive	? 7:13								
Refactor to remove duplicate code after passing test	X 13:33								
Re-run tests after every refactoring									
Only write new code when a test is failing, each test should test new/different behaviour									
Write the assertion first									
Minimise assertions in each test									
All tests should pass before writing the next test	X 18:21								
Only refactor when all tests are passing									
Write the simplest code to pass the test									
Have separate source and test folders, test code should follow structure of src									
Don't introduce dependencies between tests. Test should pass when run in any order.									

ASSESSED BY JANE D. PEER

Name: John Q. Programmer

Impress Your Friends!



codemanship

Test-driven Development

This certificate is awarded to:

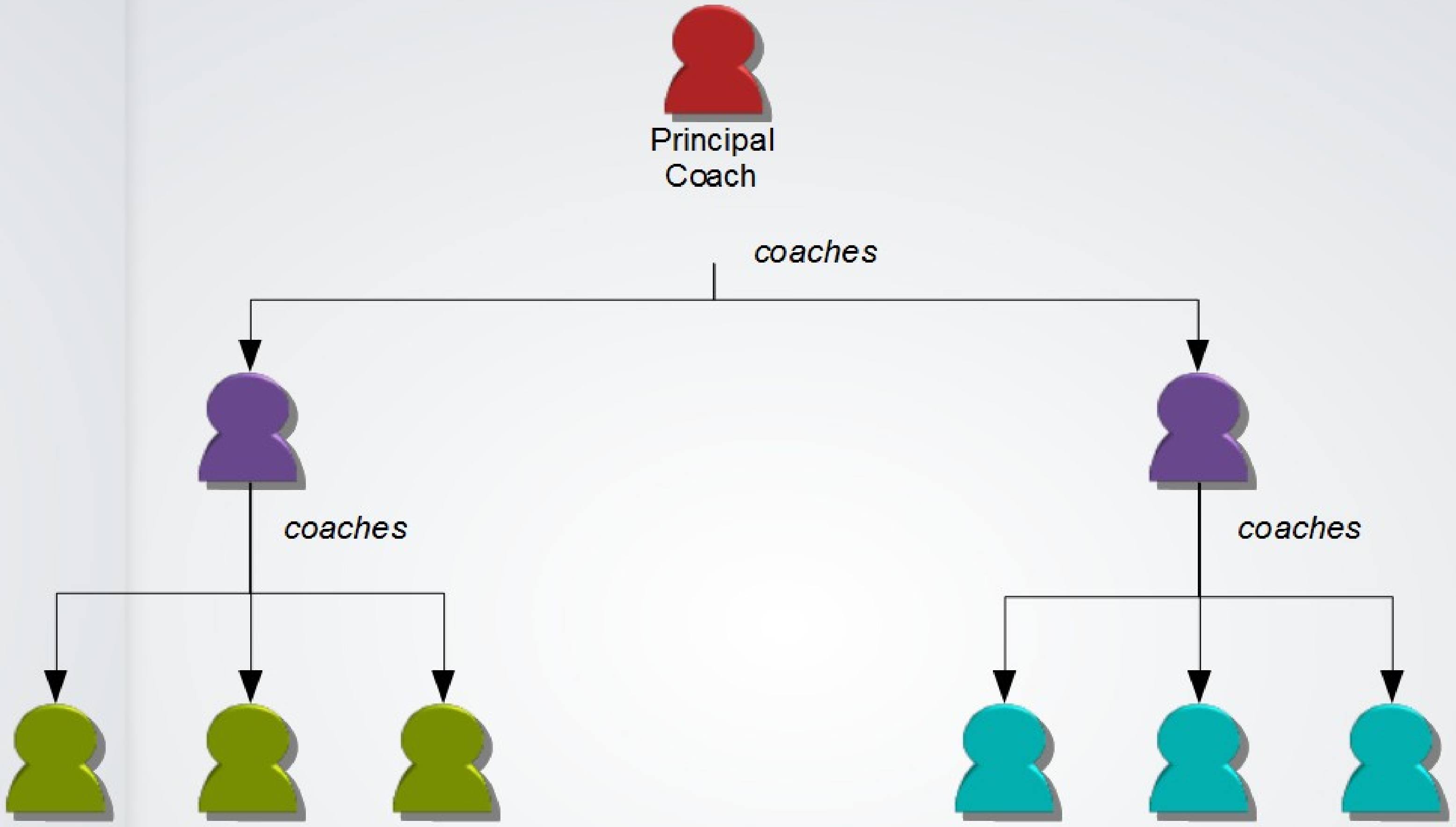
Andrew Evans

For successfully completing an apprenticeship and peer assessment in Test-driven Development

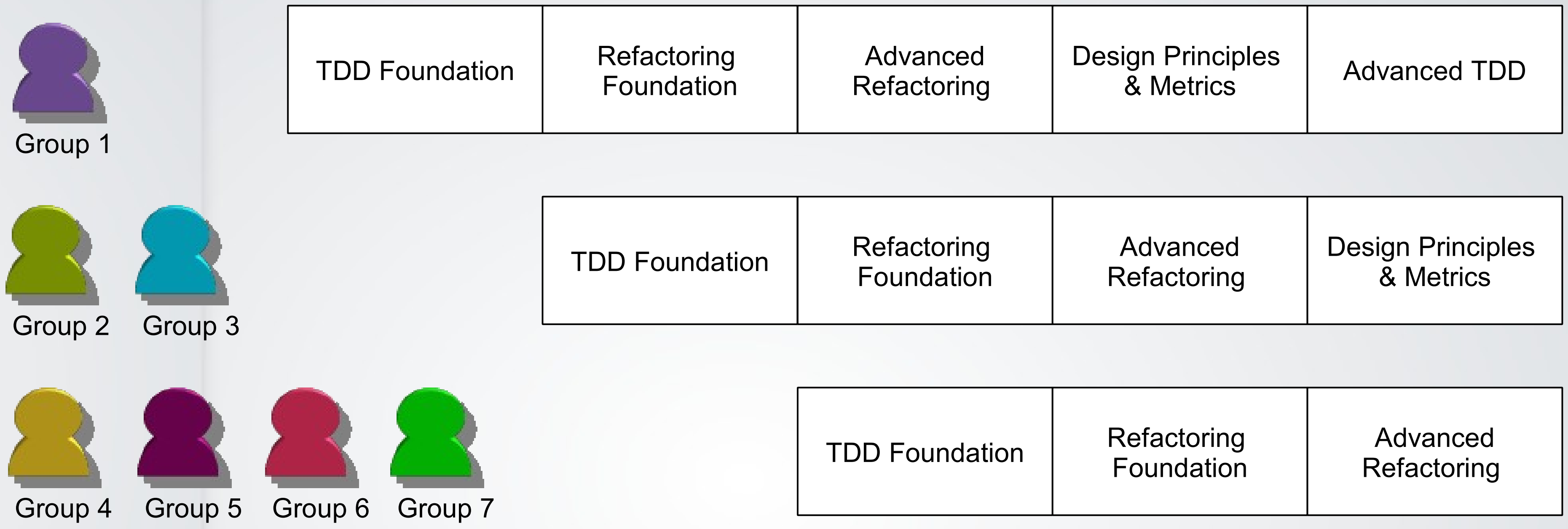
Authorized by: Jason Gorman, Director, Codemanship Ltd

Date of Award: 01/09/2009

“Apprentices” Become Coaches



Codemanship "Academy"



Proof Of The Pudding

	Before TDD Apprenticeships Started	After TDD Apprenticeships Started
% Code Coverage	57%	78%
% Long Methods (< 10 LOC)	0.94%	0.85%
Average Method Complexity (Paths)	2.1	1.9
% Duplicate Code	8%	3%

codemanship

jason.gorman@codemanship.com

A key goal of Agile methods like Extreme Programming is to deliver value at a sustainable pace, responding to changing requirements for as long as the customer needs us to.

In reality, as our code grows it has a tendency to become overly complicated, difficult to understand and exponentially harder to change if we let it.

Agility requires us to keep our code as “liquid” as we can to ensure our responsiveness to change can be maintained.

Countless industry studies over recent decades show beyond reasonable doubt that practices that it bugs cost exponentially more to fix the longer they go untackled. Teams have found that, to a point, effort taken to prevent bugs pays off in improved productivity – which runs counter to what many of us might believe.

A key factor in our ability to change code is our ability to understand it. Developers and managers seriously underestimate how much of time is taken up trying to get our heads around not only code written by other developers, but code we rote ourselves in the past.

Traditional wisdom tells us that code that is hard to understand should be explained using comments or other kinds of documentation, but reality teaches us that documentation very quickly gets out-of-date and can even be misleading. As one commentator put it - “documentation is useful until you need it”

Experience is teaching us that there is no substitute for code that communicates its intent and design clearly.

Just as it pays dividends to invest time up-front on practices that detect bugs early, it pays dividends to invest time in making sure our code is as clear and self-explanatory as possible.

The our code evolves and grows, the more complex it is likely to become. Complexity is a real productivity killer. Complex code is harder to understand, harder to test, harder to manage and much more likely to be wrong.

The Software Engineering Institute's Maintainability Index has shown a very direct link between the complexity of code and the cost of maintaining it.

Programming languages give us mechanisms for breaking down complexity into manageable, testable, comprehensible bite-sized chunks, but many developers are not in the habit of breaking their code down as it grows. Instead, functions and modules that are large have a tendency to grow larger.

As well, for whatever underlying reasons, programmers have a marked inclination to “be clever” with their code, and are not often inclined to come up with a simple solution to a problem – perhaps because they subconsciously feel that it would not demonstrate their technical prowess as well as a more sophisticated solution.

In the heat of battle, programmers – including myself – can get a bit lazy. We have a block of code that does something useful and we need another block of code that does something very similar. We could extract that block of code into a reusable function or module, but it's much less effort initially to just copy and paste it and make the necessary changes.

Act in haste, repent at leisure. The common code that's duplicated by copying and pasting may itself need to change, but we now have to change it in multiple places, which will take us several times longer than it would have done if we'd created that reusable function or module in the first place.

As with other aspects of maintainability, time invested up-front tackling duplication as it emerges pays dividend later on. And usually much sooner than you might think.

When we reuse code it means referencing modules and invoking functions. While reuse is very desirable to combat duplication and it's crippling effects, it has a side-effect – dependencies.

If lots of code reuses something, then making a change to it could cause the dependent code to break. And this in turn could break code that was depending on the dependent code. The waves of disruption from even a tiny change can "ripple" out through the software, turning it into a big and expensive change.

To minimise the risk of small changes mushrooming into major projects, we must effectuvely manage the dependencies in our code so that things that depend heavily on each other are packaged together (localise the ripples) and that the things we depend on are unlikely to change (e.g., depend on things that can be easily substituted for, like interfaces)

How quickly we detect bugs is a major factor in how much it will cost to fix them. Because of factors we've already covered, every time we change code, there's a significant risk of breaking dependent code.

The level of assurance we get from retesting (“regression testing”) the code therefore plays a big role in how likely it is that one of these new bugs will be detected.

Code that is not properly tested can conceal bugs, and more bugs can accrue in the time elapsed before retesting.

The more of the code is tested, and the more often it's tested, the less bugs can creep through and the less time we will waste fixing them later.

Teams that suffer low regression test assurance often report low confidence in changing code.

I've been a trainer and coach for over a decade, and I've noticed a trend.

The developers who improved the most were invariably the ones who wanted to learn. In a very real sense, their improvement was vastly more a product of their learning and their enthusiasm to learn than my teaching.

Telling developers “you will learn TDD” or sending them on training courses JUST DOESN'T WORK.

If they want to learn, they will learn. Period.

TDD is a great example of a skill that's actually very easy to explain. Most developers have read about it. I've discovered that many developers know what they SHOULD be doing. But most of them don't. Not in practice.

(Indeed, I've worked with “experts” in TDD and refactoring who don't apply what they know on real code)

Peer Group Learning & Assessment is only partially about knowledge. In reality, there's not much to know. Instead of focusing on what we SHOULD do or SHOULD know, we focus on what we do DO.

It is mostly about building habits and getting the “knack” of these disciplines. Just like playing a musical instrument or riding a bike.

Practice, practice, practice.

Although developers doing PGLA may learn some new things and a bit of extra theory to underpin their skills, almost the entire focus of their learning is on practice. Working on simple exercises, or on their own code (ideally a mix of the two.)

Practice needs to be focused and it needs to be done over an extended period of time for the knowledge to turn into habits.

How do you know if you're “in the habit”?

The only sensible way to know if you really do the things you know you should be doing is to do them and assess how effective you are in those respects.

Peer Group Learning & Assessment begins by gathering the group together and agreeing what it means to be doing something effectively.

Members of the peer group write down on post-it notes the habits and good practices they associate with the discipline. The group goes through these, explaining what they mean and eliminating obvious duplications, so they are left with a good set of candidate practices.

If this set of candidate practices is unmanageably large (we're finding > 12 is stretching things), they are prioritised and whittled down by a group vote to a manageable (and workable) subset.

If some practices are considered to be too advanced for the group as a whole at their current skill level, then they can be deferred to a later “semester” for reconsideration.

The group creates a worksheet based on the selected practices they wish to build into habits.

Members of the group pair with each other on a regular basis (e.g. once or twice a week) and try to apply these practices on a TDD exercise (e.g., Fibonacci Number Generator) or on real code they're working on. In each pairing session, peers can sign each other off on a maximum of 4 of the practices based on whether they feel their partner did those things effectively during pairing.

Working this way, they progress through the worksheet, collecting signatures in all the available boxes. On average, it takes about 40-60 hours of pairing to complete a worksheet, usually over 4-6 months.

Peer group members must collect signatures from at least 2 of their peers for every practice on the worksheet, as well as one signature for each practice from their coaches. (More about coaches in a bit.) This reduces the risk of two group members giving each other an easy ride.

When the peer group have completed their worksheets, they take a practical assessment.

This involves performing TDD exercises known to the group and the coaches in 3 one-hour blocks, divided into 25-minute pomodoro-style sessions. Sessions are recorded without audio using a desktop capture tool so we have video records of exactly what they did.

After each one-hour block, another member of the group watches his peer's videos back and keeps a close eye out for instances where they fail to apply the agreed practices (as defined on their worksheet). He marks a cross in the appropriate box when he sees a lapse in habits, and notes the time in the video where it occurred for reference.

If a peer group member accumulates 3 crosses in any one-hour block, the group analyses the video and takes a vote as to whether they have been successful in their assessment.

If a peer group member gets more than 3 crosses against them in any one-hour slot, they have failed their assessment. They join another peer group to repeat the semester, or appeal against the assessment (in which case, one of their peers will rewatch their videos and reassess them from scratch).

The final one-hour block is assessed by the Principal Coach (more about her later)

Every client team who has undergone PGLA in TDD has requested certificates. What's a guy gonna' do?

There is also a LinkedIn group for Codemanship Almuni, and there will be a dedicated area on my web site listing all those who have successfully completed an assessment.

Reports from those who've completed assessments is that they felt a genuine sense of achievement, and it's right that this achievement should be recognised.

But this is NOT a certification program, nor is it the Institute of Software Craftsmanship!

The program is designed to scale up without a linear increase in external coaching overhead. “Apprentices” become coaches once they have successfully demonstrated they know what they're doing in a given discipline.

The Principal Coach oversees the program and QA's the coaching by pairing with everyone in every peer group once, as well as being present at every orientation and administering assessments.

It is important to have a Principal Coach who is very experienced in all the practices being covered, as well as being a very experienced software development coach.

The PGLA program works in “semesters” which all start with an orientation and end with an assessment. A pilot group begin on their own, followed by two new groups who are coached by members of the pilot group and so on.

The “curriciulum” looks like the one illustrated above. Within 3 years, a peer group will have covered many of the critical aspects affecting code manintainability – TDD, refactoring and OO design.

There is no reason why the program can't continue beyond that to cover disciplines like BDD/ATDD, advanced TDD/mocking and so on.

Talks are afoot to align the PGLA program at the BBC with university activities and possibly to incorporate PGLA programs into degree and postgrad courses.

Data taken from a dozen projects in BBC TV Platforms show encouraging early results for the TDD PGLA we've done so far. 8 engineers out of a team of some 30+ have completed an assessment, and this highlights the effect those 8 people have been having on the code they've been working on.

As we might expects, the test coverage has gone up significantly. With basic refactoring during TDD to remove duplicate code, we have also seen methods getting shorter and less complex, and a significant reduction in duplicate code detected by Simian in those projects.

We are quietly confident about greater improvements over the next year and beyond as more engineers begin PGLA.