**Introduction**

When I had my thirteenth birthday, my parents bought me an awesome gift, they bought me a skateboard. Now, this wasn't your ordinary plain shaped skateboard. The deck on this board was what is called a cruiser deck. It's one of the coolest things you will ever see. Being inspired by this board, I realized my calling in life was to become a famous skate boarder.

To start off, I tried my hand at skateboard tricks - airs, backsides, kick flips, ollies, you name it - unfortunately, what I discovered with skateboard tricks is that they all have one thing in common - they require coordination.

Having failed at tricks, I moved on to the next best thing, what I lacked in coordination I would make up in speed. With this brilliant idea, I set out with my two younger brothers to find an appropriate hill to demonstrate my ability.

Having grown up in KwaZulu Natal, in a place called Forest Hills, I was in no shortage of options. In fact, just a block away from where we lived was a perfect specimen of a hill with a newly tarred road going down it.

To this day I remember the event clearly. It was a late January afternoon, my brothers and I walked to the top of the hill where I stood for a few moments staring down it as if I was a surfer looking at a large wave. Then, with nerves of steel, I climbed on my skateboard and started my descent.

About 10 seconds into the ride and about 5 meters down the hill I suddenly had a feeling that this might not be a good idea. While my skateboard was doing really well and rapidly picking up speed, it occurred to me that there were a few things I hadn't considered.

Firstly, the road down this hill came to an abrupt stop at the bottom as it joined a T Junction. Cars traveling along the bottom of the hill had zero visibility of anything coming down it.

Secondly, and probably more importantly, I wasn't dressed for the occasion. I was wearing shorts and a T-Shirt with no protective gear.

As these thoughts dawned on me and my speed rapidly increased I made the first good decision of that day - I jumped off.

As I jumped my world went into slow motion - the skateboard which just seconds before had been under my feet continued down the hill and then veered off and crashed into bushes.

I also continued down the hill for a few meters - but with no protective gear I had to rely on my bare hands and knees to come to a stop. Standing up and looking down I was surprized that where moments ago was healthy skin, was now replaced by blood and torn flesh.

**Relating Skateboarding to Software Development**

In my professional career of developing software I've seen and heard of many organizations doing exactly the same thing that I did on my skateboard that day.

These organizations want to develop software faster - they've sent everyone on an agile course, put up a scrum board and are doing daily stand ups AND they haven't really thought it through properly - they've missed something really important which is **if you want to develop software faster it is naive to think that the only thing you need to adjust is how you interact - you also need to adjust your engineering practices to be able to work at the same speed as your interactions**.

How do so many organizations miss this?

Part of the reason is because the two most popular agile methodologies in circulation, Scrum and Kanban, don't specify any specific engineering practices to adopt when applying them. That doesn't mean they advocate bad software engineering practices - they are just not specific on what good engineering practices are.

This is a problem. It has left many organizations not realizing that there are additional skills and practices they will need to adopt to be able to engineer software effectively.

Today I'm going to speak about some of the things that are rarely spoken about at a Scrum or Kanban course. These are the things that managers and "non" technical people often don't get to discuss but that are just as critical to understand to be effective.

I've called them the natural order of agile technical practices.

**What are agile technical practices**

So, what do I mean by agile technical practices? When I pose this question to people involved in software development I get a range of answers.

For the next 25 minutes I'm going to use my definition for the term - which is when I say agile technical practices **I am referring to the engineering practices related to creating software in an agile environment**.

Some of the things I would include under the banner of engineering practices are: Collaborative Coding, Continuous Integration, Continuous Deployment, Automated Acceptance Tests, Test Driven Development, Collective Coding Standards, Simple Design and Refactoring.

**Structure, Interactions & Technical Practices are intertwined**

With that, I want to briefly mention team and organizational structures. There is a really good talk Kent Beck did at Lean Kanban Central Europe a few years back called GeForce. In Kent's talk he spoke about how different rates of deployment require different team structures and activities.

Today I'm going to assume you have your team and organizational structures sorted. I am going to assume you have small cross functional teams that are under pressure to deliver working software regularly.

It's my observation that engineering practices and team structures are joined at the hip. Some practices are impossible to do effectively if you don't have a suitable team structure. If you are unsure whether your team structure is suitable I recommend watching Kent's talk.

**Help Create Slack**

Now, before I speak about the actual engineering practices I want to speak about something that is important for good engineering practices to happen - it’s the necessity of having slack. One of the most important things you can do to help your teams is help them create slack.

When I talk about slack, I am referring to the time a team has to improving their engineering process and upskilling. Without slack your teams will not be able to adopt new practices, which mean things won't get better.

A good rule of thumb to follow with regards to slack is to **prefer small regular intervals of slack over large infrequent blocks of slack**.

I have found that teams that have small daily and weekly sessions built into their normal routine learn and improve at a faster rate than teams that take a large block of time off to learn something new.

In the team I am currently in we have experimented with various approaches, from having weekly lunch sessions where we talk about design principles to kata's where we learn new techniques and video sessions where we watch recordings of conferences on topics we find interesting.

The important thing is this, **it doesn't matter how your team does it, but encourage learning new things regularly**.

**Supporting the Dips**

Another thing you need to do is support the dips.

Whenever a team or an individual learns a new engineering practice there is going to be an initial dip in performance.

You need to know that this will happen and have enough understanding of the long term advantages that will be gained from the practice to support them during the dips.

**Pattern to Create Slack**

So this may all sound great in theory, but in the real world we can't just tell people to stop working. We have deadlines!

I've heard this a few times. It's a misconception that comes from a resource utilization mind-set.

* Firstly, there is very little correlation to how busy someone is and the amount of work that gets done.
* Secondly, if you are under pressure there are various approaches you can take to create slack that don't require you stopping the ship.

For instance, one way is to invest in practices that have a shallow dip and a quick recovery period.

These practices may not necessarily be the most valuable long term practices AND they may be extremely useful to create the initial slack needed to tackle the harder and sometimes more time consuming ones.

The approach I like to use is as follows:

* As a team, identify where the bottlenecks are.
* Get an understanding of what practices can help reduce the bottlenecks and roughly how expensive the practices are to adopt.
* As a team pick an appropriate practice, then spend time learning and applying it to your system.

As the practice starts to realize a return you will have created additional capacity. Let your team re-invest the additional capacity to apply to the next identified bottleneck.

At some point you will create more time than you need for slack. When this happens your team becomes progressively faster and the interesting thing with getting progressively faster is that it becomes a drug. **The faster you go, the faster you want to go**.

So, with that said, let's move on to some actual practices.

**Effective use of Version Control System**

The first practice I would like to talk about is the effective use of version control.

So, most of the teams that I have spent time with are already using version control AND many of them are using it wrong - they see version control as a fancy word for backup tool.

**Version control is not just a backup tool, it is also a really powerful integration tool.**

One of the major advantages of a version control system is the pain it takes away in resolving merge conflicts.

This means that many developers can work simultaneously on the same code base and the version control system will do the majority of the work involved in putting their code together as a single solution.

One of the most common symptoms of misuse is when a team complains of regular merge conflicts or having to always manually merge their work.

**This is a symptom of the team not continuously integrating.**

Continuous integration starts with a discipline. Everyone needs to continuously integrate to trunk.

If your team has murmurings of their version control system not working or how they hate it, you are going to be dead in the water trying to get more advanced technical practices implemented.

The pattern with version control **is to do small little bits of work and merge or integrate frequently** (in this case frequently means hours, not days).

**Automating and Democratizing the Build**

Leading on from effective use of version control is the practice of automating and democratizing the build. I've put this as the next practice because it is so simple to do, yet I have come across many teams that just haven't done it.

When I refer to the build I am talking about **the set of files that are packaged or generated, that are required for a system to run on a test or production environment.**

**Some organizations refer to this as the build pack.**

For teams that previously released infrequently, this is usually a manual process owned by a single person.

When you automate and democratize the build it means that the build is automatically generated by a script file instead of a person AND that anyone in the team can trigger this process - even the non-technical people.

How do you know if your build is automated? A good way to gauge is to ask a team to generate a build pack - if it takes them more than a minute or two it is not automated enough.

Most teams I have come across that have not automated their build process yet will tell you that they are the special case. That it is extremely hard for their particular technology stack and that they have special circumstances that require a specific person doing it manually.

Personally, I have yet to come across a team that hasn't been able to automate and democratize the build after applying their minds to it.

For instance, I once worked with a team that was under a huge amount of pressure working on a very legacy application. There was one particular developer that stood out as always being busy. After some time with the team we noticed that every couple of weeks this particular developer would disappear for a day to make a build pack for the testers.

Because of the pressure he was under, occasionally he would miss a step during the build process and send through a build pack with an old file. This would only get picked up after a few hours of it being with the testers which meant he would then need to spend another day re-doing the build and the manual testing would need start over again.

You can imagine how popular he was when this happened and how much he enjoyed his build days.

After quite a bit of encouragement we convinced him to give us time to help him automate the build.

It took us a few days, but we managed to automate the build so that anyone could generate it. What once was an error prone process that took almost a full day of a developers time was reduced to a matter of seconds.

The net result of this wasn't just creating an additional a day and a half a month of developer time.

Automating the build made making the build painless. Things that are painless are done more often which in this case resulted in the team making smaller, more frequent builds.

Smaller more frequent builds resulted in more frequent releases which in turn meant that business got their features sooner.

**If you have not automated and democratized your build yet, seriously consider it as a first line candidate.**

**Make the Continuous Integration Server Visislbe**

Doing smaller bits of work and integrating frequently means that people work closer together. When people work closer together, it becomes important to make build and integration issues visible to the team as quickly as possible. To do this you need a continuous integration server.

A CI Server, also called a build server, is a centralized server that builds your project whenever a check-in is done to the version control - it's responsibility is to make sure that everything is there and playing nicely.

One of the big payoffs from a ci server is that it can make the general health of your code base visible to everyone regardless of whether you are technical or not.

For instance, in a couple of teams that I have been involved with we plugged a monitor into our ci server that we put up in the team room.

Whenever a developer committed work to version control, our ci server would automatically detect it, rebuild the solution and tell everyone how the build went.

It used 3 colours - orange meant it was currently building, green meant that the last build was a success, and red meant we had broken something.

Our team goal was try and keep the ci server green as much as possible.

Whenever it went red, we would stop what we were doing and fix it immediately because we knew it had a major impact on everyone.

Making the ci server physically visible to the entire team helped us focus on sorting out integrations issues quickly AND helped everyone to keep it in the back of their minds that we were working together as a team on the same system.

If you have already automated your build scripts and have your version control system working correctly, setting up a ci server is often trivial. It is something that every team needs to have.

**Collective Code Ownership**

So, let's say your team is using version control effectively, you have your build scripts humming and your ci server is continuously integrating. What else can you get value from?

It is at this point I would like to talk a bit about the practices that support the principle of collective code ownership.

Collective code ownership is the principle that **code belongs to the project, not the individual**.

This means that anyone working on the system should be able to make changes to any part of the system without fear that they are stepping on someone’s turf.

**Common Code Style**

One practice falling under collective code ownership is a team having a common coding style.

By this I don't mean a thick document outlining exactly how everything is done and dictated by the architects in the architect forum - that doesn't work.

I rather mean a team having a common understanding on what coding style their group has.

It stands to reason that to have a common style, teams need to regularly look at and discuss code they are writing as a group.

**Collaborative Coding**

One of the best ways to achieve a common style is to support collaborative coding. There are several different types of collaborative coding.

On the diagram I have put up, the further right you go the more consistent I have found a team be in their common style.

Some these terms may look unfamiliar - most notably pair & mob programming - let me briefly explain them.

Pair programming is the practice where **two people sit in front of one computer solving a problem together**.

Mob programming is very similar, **except instead of just two people, you have several people in front of a single machine solving a common problem**.

Typically when I talk about these two practices and there are business people in the room, someone will ask how one can justify the costs of two or more people doing one person’s job?

If you have two programmers, surely they should be programming on their own work on their own machines otherwise they are not being efficient!

The best way I can explain why these forms of programming work is by explaining what we mean when we say that someone is programming.

**Programming is not about typing, it's about problem solving.**

Depending on the problem being solved, there are many advantages to having more than one person working on the same problem together.

**Test Driven Development & Automated Testing**

So, I have time for just one more practice. There are many other practices you could look to adopt at this point.

I would be doing you an injustice if I didn't speak about test driven development and automated testing.

**TDD vs Automated Testing**

In recent years I have noticed the growth in popularity of test driven development.

However, in discussions with people who have not practiced any form of test driven development before, I find they get confused between test driven development and automated tests in general.

Test driven development or TDD is a particular workflow that developers use when writing software. **It involves first the developer writing an automated test that defines a desired improvement or new function, the developer then produces the minimum amount of code to get the test to pass and finally refactors the new code to acceptable standards**.

The benefits of TDD include helping create a simple design and inspiring confidence in the developer that the code is doing what it is meant to be doing.

Automated tests on the other hand are merely that, a set of tests that can be run automatically and written by anyone at any point in time to verify that a system is functioning as expected. Some automated tests can be written post development by testers. These types of tests merely verify that the system is operating as expected and yield no benefits to design.

**Why TDD is useful**

TDD is extremely useful. Firstly it teaches developers good design - in particular writing software that is loosely coupled.

The best analogy I can think of regarding why loosely coupled code is important is this:

If programming new features was like building blocks, then pre-tdd I was making really big wooden blocks and using super glue to organize the blocks in the shape I wanted.

Learning TDD taught me to make my blocks a lot smaller and then adjust them slightly so that they worked like Lego blocks - now, instead of having to glue things together I could just clip things in and out rapidly.

In an agile world, where we embrace change, having code like Lego blocks allows us to change direction really quickly which becomes a major competitive advantage.

Now let me warn you upfront, if your engineers have not done TDD before, it can be challenging.

The first time I saw test driven development I saw someone demonstrate it by writing some code, running something, and a bunch lights on the screen went green. It looked really simple, almost magical. I thought, great, all I need to do is go home and do TDD and then I won’t have any bugs.

So, I went home, spent a very frustrating two days trying to write a useful test against the current system I was working on at. I failed dismally! I could not write a single useful test let alone write it before I wrote its implementation. You see, the problem was that the code I had written up to that point in my life was not unit testable - it was super glued wooden blocks.

It took me more than a year of occasionally dabbling with TDD to understand how to do it, and another year to start practicing it, AND once I began to get into it, I began to realize how powerful it is.

**Best way to learn TDD**

So, if TDD is so hard to learn, what is the best way to learn it? At a high level, there are two things you need to learn TDD :

**Time and Exposure.**

For me, I needed a year to think about it and several failed attempts at applying it before I got it.

In terms of learning resources, probably the most I learnt about TDD was at a code retreat.

Code retreats are a daylong event of intentional practice in a collaborative coding environment. In Johannesburg, there are community ones once or twice a year on a Saturday that are free to attend.

Now, sometimes it’s challenging to get people with busy lives to attend events on their Saturdays. If you or your team are in this position I would recommend spending the money getting someone in to host an internal company code retreat.

I warn you though, don't expect to get TDD ninjas after one day at a CodeRetreat.

It is going to take several positive exposures to the practice and an extended period of time applying it before you will see returns.

That said it is worth every bit of effort you put in to it because the return is so high.

**Closing**

With that, we are coming to the end of this session. I have only covered a few technical practices.

These are not the only technical practices necessary to operate at the release rate that agile methodologies work at, and you may find that the order they were proposed does not necessarily make sense for you particular team.

That's fine, it's not hard to find your teams own natural order of progression. All you need to do is:

**1) Embrace** the fact that we are all learning new things every day and that your team needs slack to learn

**2) Identify** your team’s specific bottlenecks and the potential practices that can improve it

**3) Be strategic** in picking a new practice. Be willing to soldier the dips when adopting it

**4) Make improvement** part of your daily process, not a special event

**Story about MaxCut**

With that I'm going to end off with a story - I call this my hope story - it's about a program called MaxCut.

MaxCut is a program I have been involved in developing for several years. The first version was written many years ago before I was introduced to any of the practices I've spoken about today.

In the early days of MaxCut we noticed the same effects we had seen in other projects - as we added feature on feature, it was taking longer and longer to make changes and keep everything working together.

One day I got email from a user asking for a really useful new feature to be added.

At the time it I knew that this feature would require us working on a part of code base that was very tricky. It was going to be several weeks if not a month or two to implement.

Since we didn't have that sort of time, I parked the feature request and put it in the "**Nice Ideas but not going to happen**" section.

A year later I was exposed to many of the practices I've spoken about today. In particular, two of the things that made a big impression on me were ci servers and test driven development.

As we learnt these practices we started applying them to MaxCut.

One Saturday morning I once again came across the feature request I had shelved a year earlier. I wondered to myself, with the improvements we had done, how long it would take to implement the feature now.

It just so happened I had some free time so I decided to try.

As I started working on the feature I was surprised at how confident I was working in the refactored code.

I was getting continual feedback from my tests and the techniques I had learnt from TDD made changing things really easy.

By the end of the day I had completed the feature, my ci server had run a full set of tests over it and I had an installation available to distribute.

**What once would have taken weeks if not months now had been accomplished in hours**.

To me, that is the power agile technical practices. They make us go faster with time, not slower. Thank you.