TestWest 2015

Nick Jenkins
www.nickjenkins.net
#nickj69

I first encountered Agile in 1999 in London – and I wasn't impressed. But in about 2009 I came across agile again, as head of Testing at Bankwest; and agile had changed. It had got smarter – it had grown up.

But it wasn't until I started studying Lean that I began to understand where agile practices came from and why they were important – take **continuous integration**. Its a combination of a couple ideas from Lean, translated into the software development vernacular. It combines "one piece flow" and "error proofing" and "SMED".

SMED stands for **Single Minute Exchange of Dies**.

In car manufacturing, large sheets of metal are pressed into doors and bonnets by huge metal presses weighing several tons. You can do hundreds of panels an hour, or more. Changing over the die, or the shape of the metal however is hard, it requires hours of down-time to change and recalibrate a press.

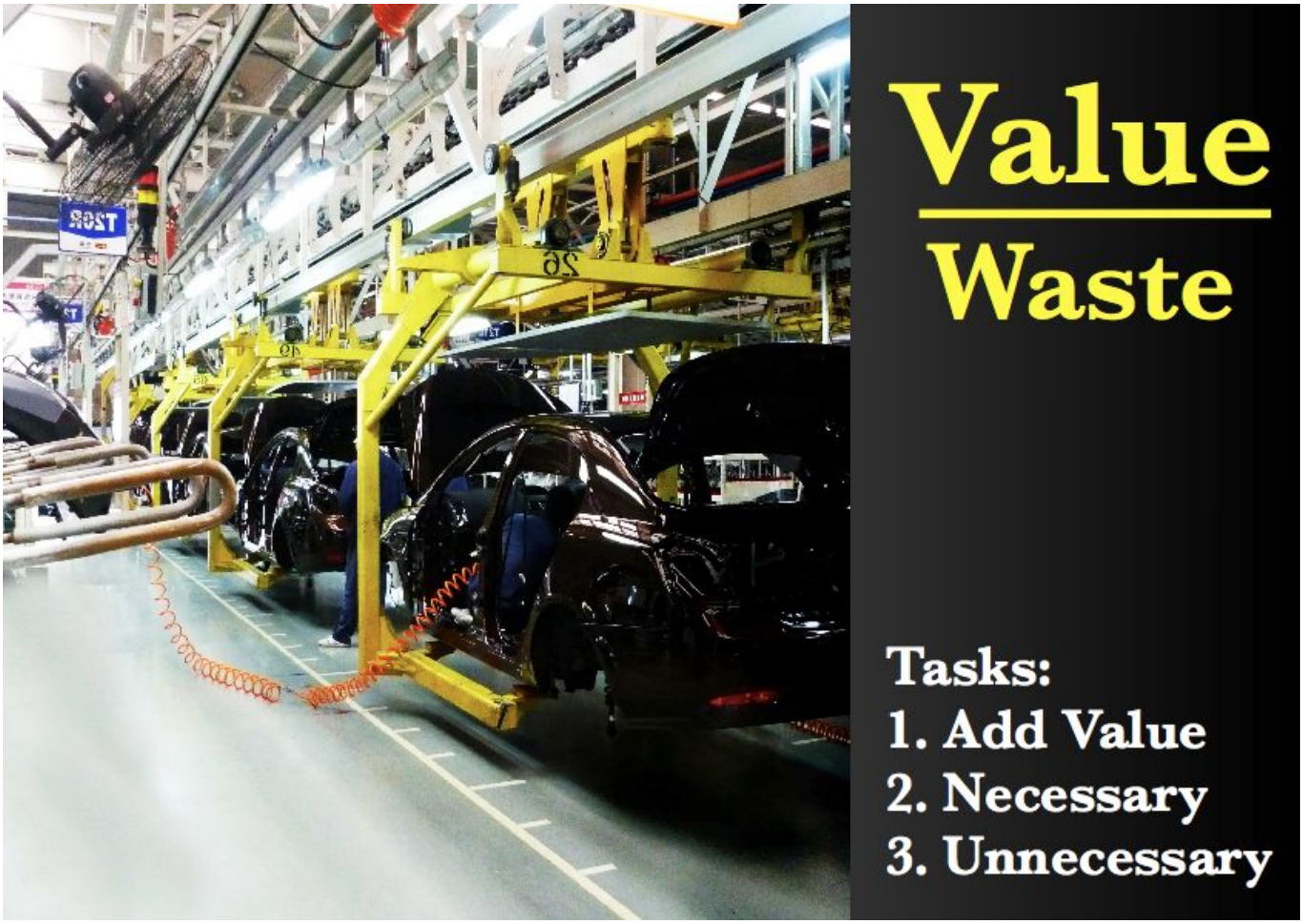
Or so the thinking went.

A few people at Toyota looked at the problem and asked the question : **why does it take so long** to change over a press? Can't we do it faster? And that changed the whole mindset.

In Toyota and other auto manufacturers around the world, they regularly changeover the die on a ten ton metal press in less than 60 seconds – **Single. Minute. Exchange. -of- Dies**.

The software development analogy to changing a die press is the **build and deployment process**.

If you decide that the smallest unit of code you can write, TEST and deploy is one line – then you maximise the throughput and a whole new world of possibilities opens up.



Value is a central concept to Lean.

Once you understand it, **it will never leave you alone.** You'll see it, or its absence, everywhere you look. Value is basically anything a customer will pay for. Anything that they will value.

Waste is the opposite**.**

When you are building a product there are basically three types of tasks:

1. Those that add value
2. Those that don't add value but are necessary
3. And those that don't add value and are unnecessary

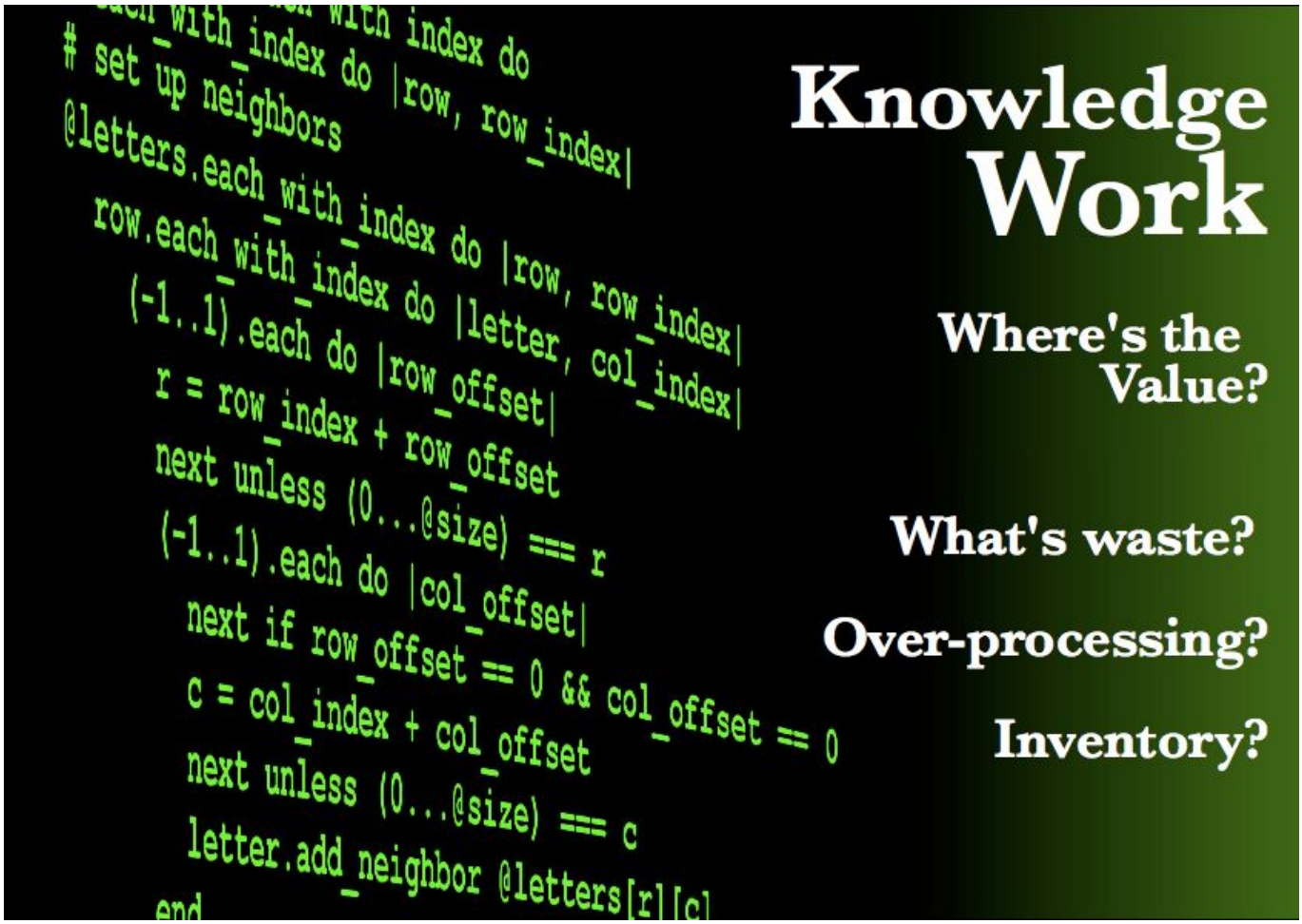
You should maximise #1, optimise #2 and eliminate #3.

Toyota has been doing this for fifty odd years and they think they've got to about 20-30% value in their production. Market leaders in other industries might have 10-15% value but most of the organisations we work for in our lives will produce **only about 3% value in their processes**.

By realising that exchanging dies was a type 2 process – it's necessary but adds no value – Toyota was able to optimise it, to remove it as a constraint to their production.

The assumption before SMED was that **time, quality and cost are dependent –** a bit like the iron triangle of project management.

But if you understand that quality doesn't flow from either time or cost. It doesn't flow from how busy people are or how much you spend. It flows from repeatable, optimised processes and careful design to meet customer's needs : if you don't swallow the lie of the iron triangle then **fast, good & cheap** products are possible.



So how do you reduce waste in software development?

The simplest way is to **reduce hand-offs**. And it turns out this is easy if you also reduce your inventory because you automatically reduce the amount of stuff there is to handle – so there's synergy there.

In manufacturing inventory is easy to spot – it lies around all over your factory in the form of half completed products that tie up space; **with software**, **the inventory is less visible but no less destructive**.

It is of course "knowledge" - or the artefacts in which knowledge is captured - like requirements, specifications, test cases and unused code. **The only thing that delivers value is working software in the hands of a customer** - everything else is waste.

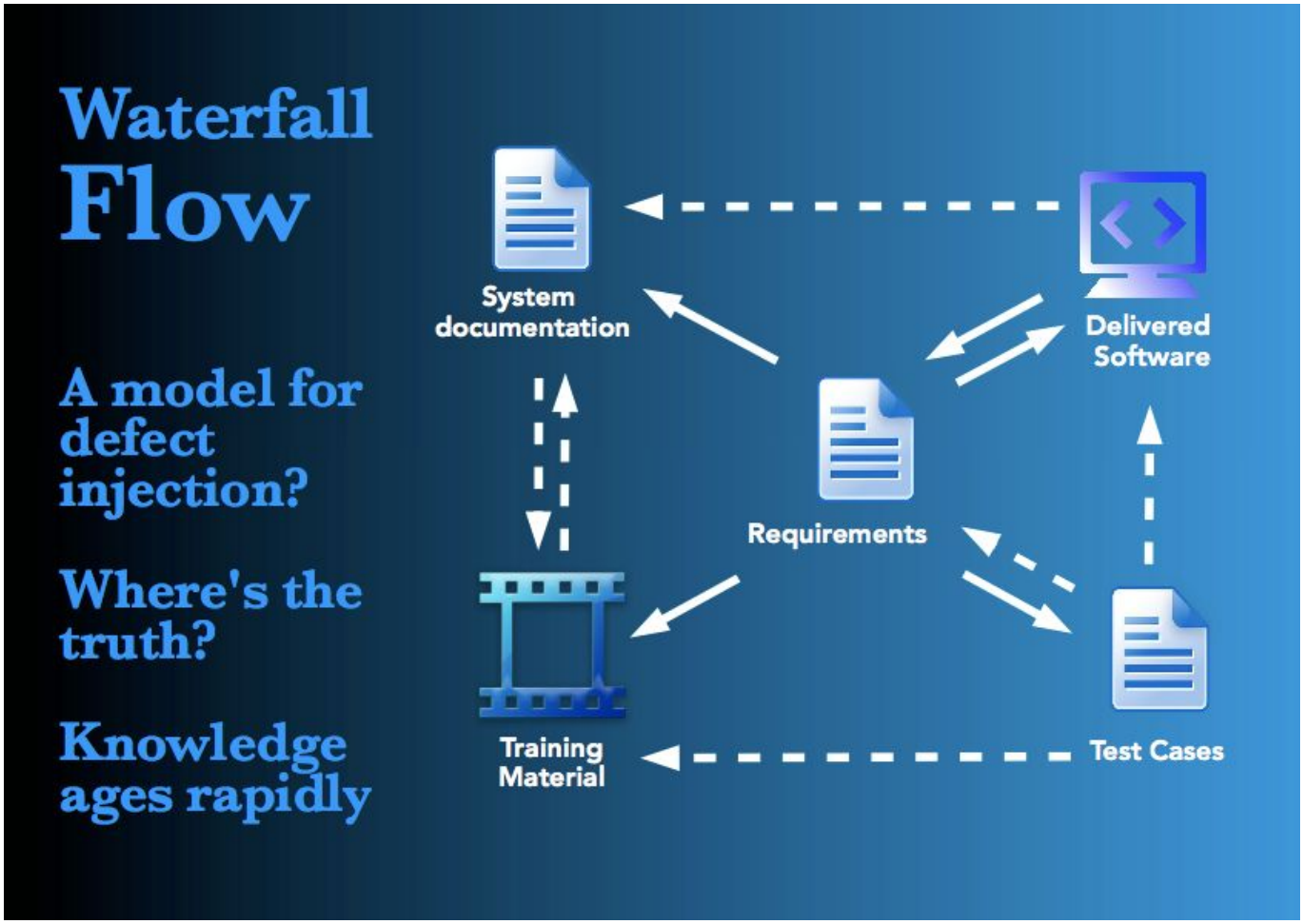
If you quantify the cost of holding, managing and maintaining all those knowledge artefacts you begin to understand the truly staggering overhead this inventory places on the software development process.

This isn't to say *some* of this isn't necessary – it just doesn't add any value – it's type 2 task. In software development **you want to maximise the ability of developers to change software and deploy it** – because it's the only thing that adds value to the customer.

For example, when you log a defect for a developer to fix a problem then you've added work (and inventory) to the development cycle, not removed it. You're adding waste not value. It's only once the developer changes the code and removes the defect that the process adds value.

The tighter you make this loop the tighter the code and the less rework is passed on down stream.

The tighter you make this loop the faster they learn.



Here's the waterfall SDLC drawn as flows of information

Primary flows of information are solid lines, and dotted lines represent secondary flows of information.

In the middle you have requirements which which drive everything – except they don't. Because software has constraints and your requirements must fit within those constraints. If you ask for software to fly you to the moon you're likely to be told that it isn't possible or costs too much and so you must modify your requirements.

So **software capability informs requirements** and there's a primary relationship there.

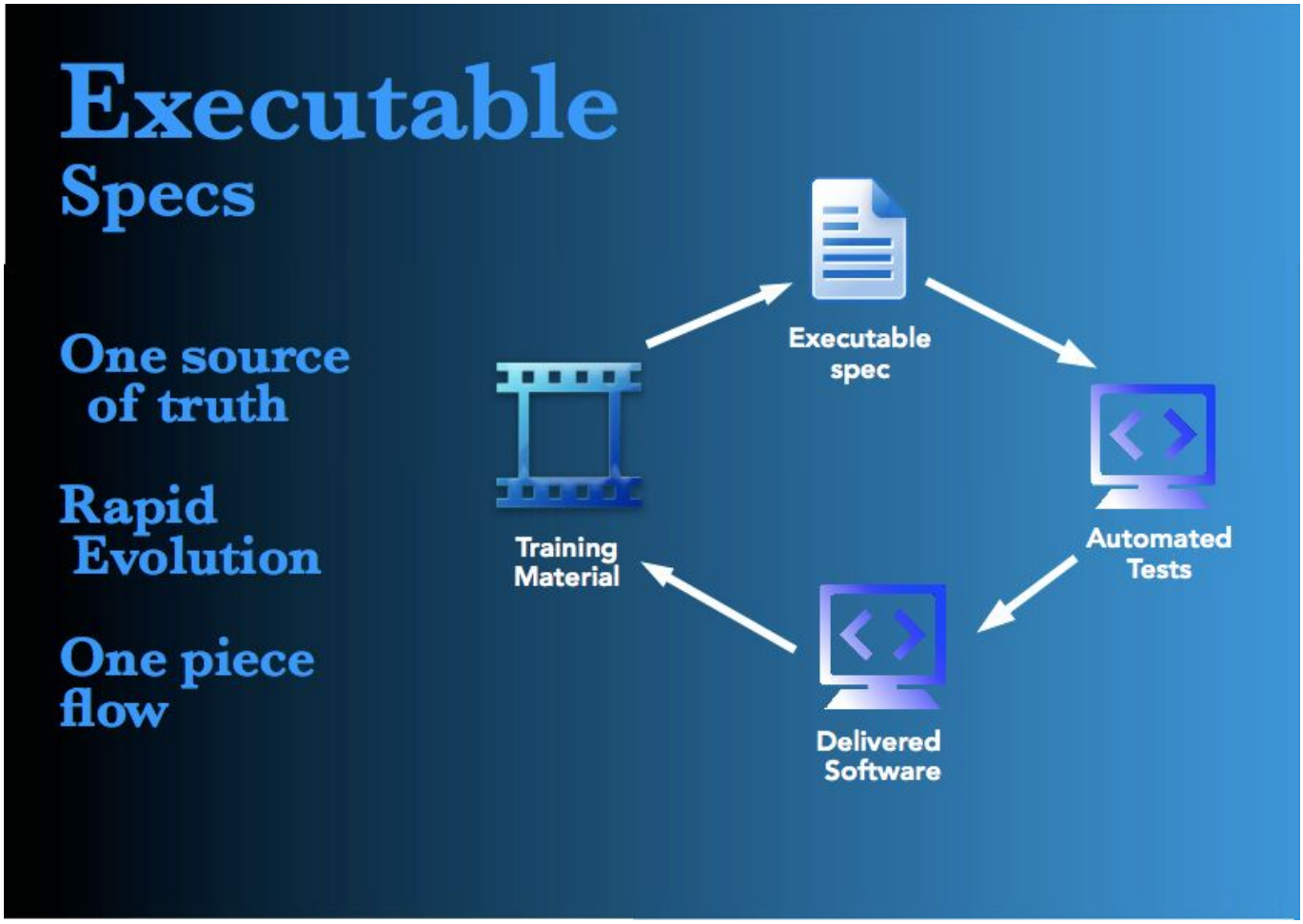
Then you have system documentation, which might be a spec or it might be a user manual. That talks about how a system should behave to fulfil your requirements. If you want Y then you do X and the software does Z. But system documentation must change when the software changes.

And the software changes in response to changes in requirements or the discovery of **gaps between what is desired and what has been delivered – or defects**. Defects are found largely by executing test cases. Test cases are derived from requirements but of course information flows the other way too because some times the defect can be in the requirement.

And finally we have **supporting documentation**, training manuals, help and instructional videos – the kind of thing that is needed to round out specifications and requirements to teach people how to use the software.

This is just short of chaos. It's not the nice, orderly process the waterfall diagram makes it out to be. Every time you touch an artefact in the waterfall process you trigger a cascade of updates that introduces the risk of error in each transaction. The waterfall process multiplies the chances of introducing defects into the process.

As Mary Poppendieck describes it, waterfall is a process designed for defect injection.



One method to embrace is "**executable specifications**" otherwise knowns as ATDD or BDD or Specification by Example, depending on which camp you're in.

Gherkin is a natural pattern language adopted extensively in agile as user stories. It uses two constructs : a feature and a scenario. Using these two you are able to specify, build and test a feature in software.

A *feature* is a high level description of what the end-user hopes to achieve and it uses the pattern :

> As a <*user*>
> I want <*a feature*>
> So that <*I get some benefit*>

It focuses on the why - the intent of the feature. It stands as a placeholder for the wider conversation.

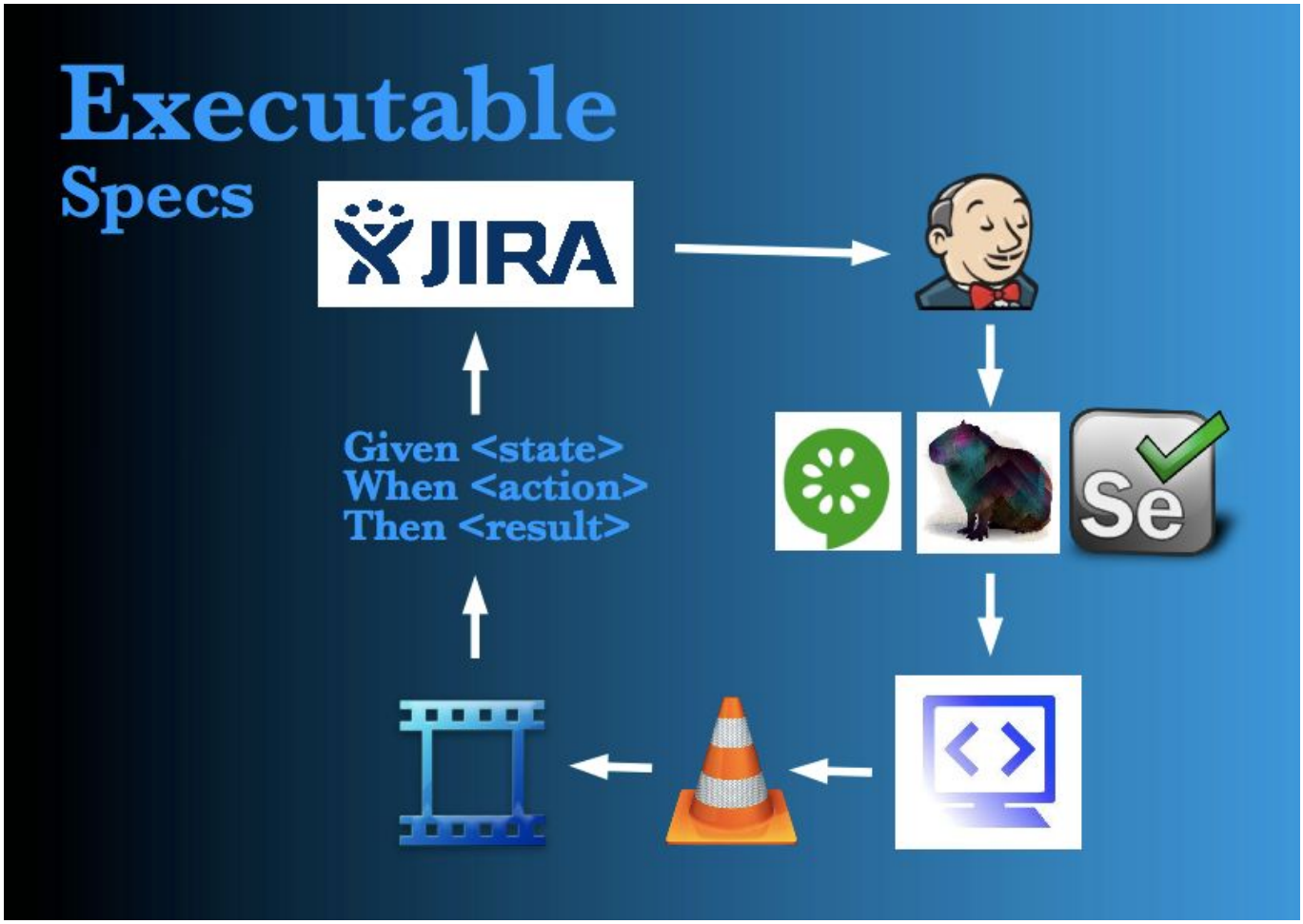
Below a feature, you have **scenarios which specify the system behaviour** in specific contexts.

They use the pattern:

> Given <*certain preconditions*>
> When <*I perform an action*>
> Then <*I get a result*>

Scenarios focus on the 'what' of the feature. They should be based on real examples and be easily readable. Typically you have multiple scenarios per feature and together they form a testable user requirement.

By combining a scenario with multiple examples in the form of scenarios you have a requirement, which is a specification, which is also a test case : reducing three artefacts in the lifecycle to one.



To leverage the BDD/gherkin construct you can tap into the wealth of automation available.

You can store your stories in a **task tracking system like JIRA**. Your analysts, SME's, product owners or even customers can help write the features and scenarios collaboratively. They can then go through a a defined workflow of review and edit until they are signed off as complete and accurate.

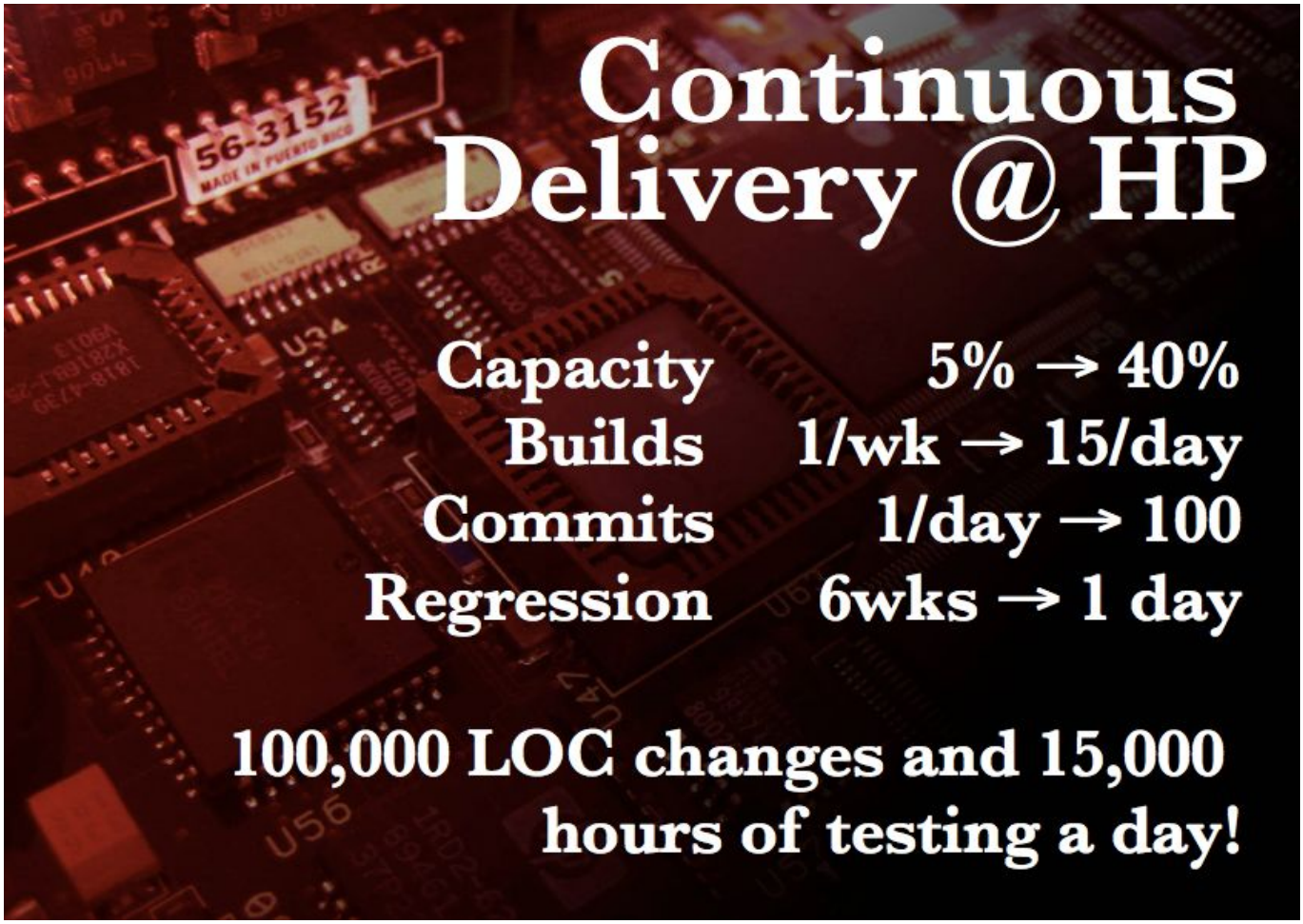
Then your developers can start work on them and iterate through solutions and requirements with end-users until you have a workable solution. Or if you are going the "test first" route you can suck the stories out of JIRA and format them as **feature files for Cucumber**. You then write 'step code' to drive the feature via something like Capybara and a Selenium web harness.

Using a **Continuous Integration server** (like Jenkins) you can automate your deployment and testing workflow on the route to production. You can trigger build/deploy/test automatically or invoke a pattern like daily builds or staging servers and blue/green builds.

If you're building to scale or your test suite is extensive you can use auto scaling test architectures like Selenium Grid to run the tests on multiple machines in parallel to reduce execution time and give us flexibility.

A little extension that we added in one project included hooking up our test stack to a **video capture engine** (VLC) so that we recorded test execution as it happened. This was useful for debugging but with a little captioning and some voice-overs, gave us a set of training material based on actual, verified requirements.

The **requirements specification** became **executable test cases** which became the **system documentation**.



It works at scale too - here's an example from HP.

At HP they had a problem in their LaserJet firmware team. **They couldn't build anything new** because fixing bugs and integrating the fixes into the mainline code consumed more than 85% of their time. Worse still the cost of maintaining that legacy tail of products was doubling every year.

They embarked on a program to reengineer how they built, tested and deployed their firmware. Before the change it would take **8 weeks** before a developer would find out whether or not a code change was successful!

After they implemented **massively parallel build and test servers**, they could run 15,000 hours of testing a day on more than 100 commits. Developers would know within hours if their code had worked and could turn it around the same day - and the improvements freed up developer's time to work on new features. Gary Gruver the VP from HP described it this way "the key is to use continuous delivery to reach the point where it is easy to release **more frequently** than the business wants and remove the delivery of software a a constraint".

There's a Toyota saying I love - "**no problem, is a problem**".

It is the spirit of continuous improvement. It means that if you can't see your problems then you can't work on them and you can't improve.

There's a great anecdote from a plant manager who moved from General Motors to Toyota. At Toyota he had a Japanese mentor and after three months he went to report on progress to his boss. When his boss inquired how many times he had stopped production in three months he proudly reported that they had not a single down-time incident, and 100% utilisation.

His boss shook his head sadly and said, "Think about how many problems you must be hiding with excess inventory and waste. **You must lower the water level so you can see the rocks**".

A friend of mine has another phrase - "the era of manual testing won't end because of a lack of manual test cases" - he's paraphrasing a Saudi oil minister who said "the stone age didn't end because of a lack of stones." The stone age ended because they had a better option.

The job of testers has always been to lower the water level so people can see the rocks.

But now we can use pumps and not buckets.

And software can become an enabler of business, not a constraint.