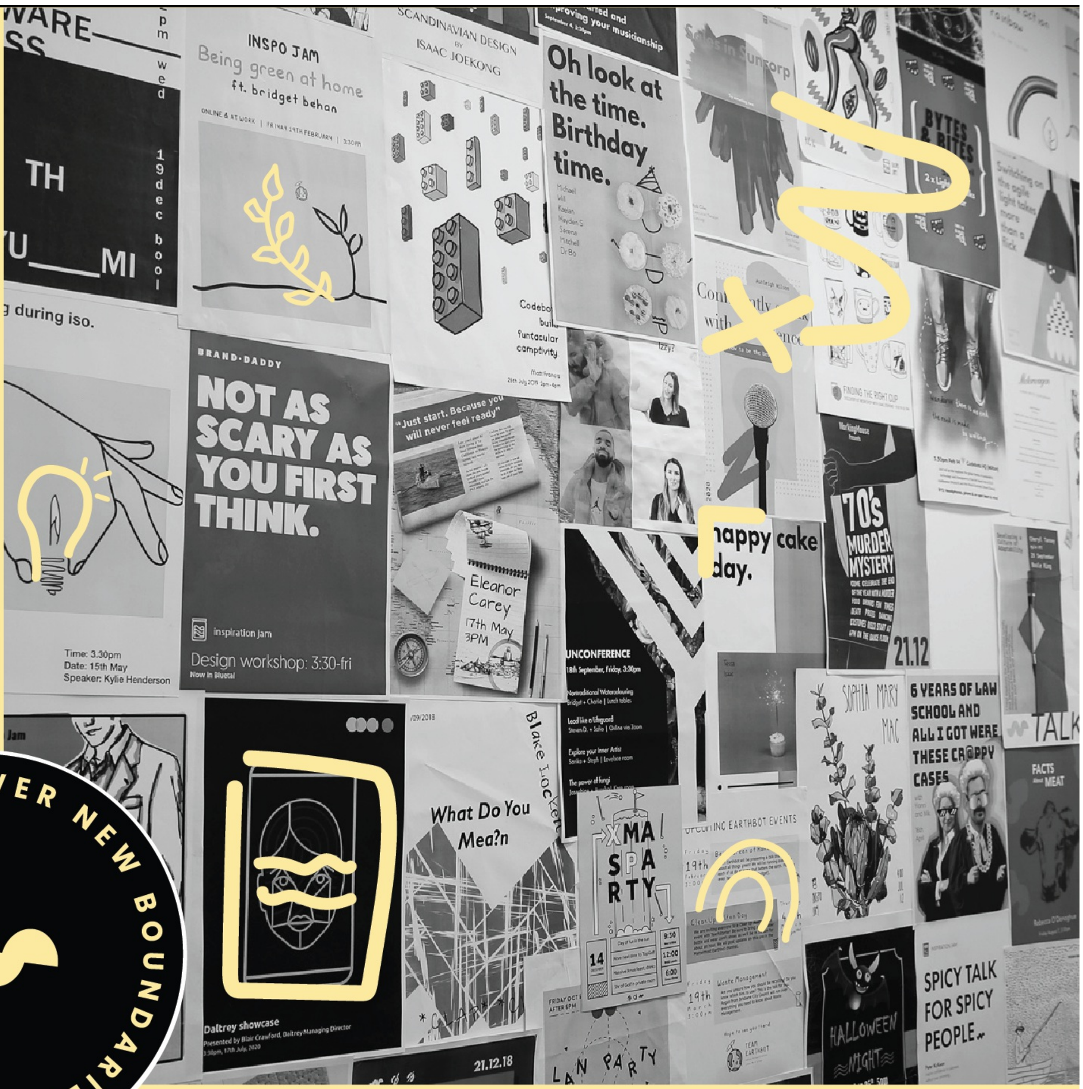

DISCOVER NEW BOUNDARIES • PUSH THE LIMIT •

workingmouse

Your Vision, Our Expertise.

Jidoka

Automation with a human touch

2023-05-23

Contents

Jidoka

> Automation with a human touch.

Jidoka is a software development methodology that aims to increase the automation and quality of a software project through the use of models and pipelines.

Our hypothesis is that Jidoka is well-suited for large scale modernisation projects where the benefits of automation are most acutely felt.

History

The software development industry has taken much inspiration from lean manufacturing. It is worth knowing as Agile, Scrum, Kanban, DevOps, and others have roots in this history. For example, TPS is a well-known implementation in lean manufacturing:

> The Toyota Production System (TPS) was established based on two concepts: “jidoka” (which can be loosely translated as “automation with a human touch”), as when a problem occurs, the equipment stops immediately, preventing defective products from being produced; and the “Just-in-Time” concept, in which each process produces only what is needed for the next process in a continuous flow. Toyota Production System

Most developers that read this interpret Jidoka as a mechanism to stop production to help ensure quality. While this has merit, our interpretation focuses on brining human and machine skills together to produce a result that neither could reach alone (read more on centaur chess teams).

Comparison

There are many well-known and published software development methodologies and other approaches to problem solving. Jidoka can be uniquely applied to projects as it has advantages others didn’t have available when they were conceived. The advantages have a central theme of automation:

- Years of R&D investment into our sister company **Codebots**
- **Model-Driven Engineering** (MDE) allows us to build models of anything and then do useful things with those models
- Pipelines can help us build software in a repeatable and scalable way using **DevOps**

and modern technologies like Docker, K8s, etc
- **Knowledge management** in its simplest form using markdown stored in a code repository
- Recent real world **experience** of building 100+ projects and **experimenting** with different methodologies to find what works and doesn't work

So, with these advantages in mind, let's examine some common acronyms and well-known development methodologies.

Approaches to problem solving

The following three acronyms are all different approaches to problem solving and finding a solution:

- **Proof of Concepts (PoCs)** are great for discovery and risk mitigation within a narrow problem set
- **Minimal Viable Products (MVPs)** are great for iterating through a number of build-measure-learn cycles when the solution is not known
- **Shortest Path to Value (SPV)** are great for the partial replacement of a legacy system

While each of these are great under some circumstances, when undergoing a large scale modernisation project, there are some significant drawbacks:

- **PoCs** are ultimately thrown away as there functional and non-functional requirements are too narrow to fit in a large scale modernisation
- **MVPs** are searching for a solution but in large scale modernisations the target is known making the search less important
- **SPVs** are designed to work in large scale modernisation projects but are used in conjunction with the strangler fig pattern that takes way too long

So, how will Jidoka learn from these:

- **PoCs** will be used for tech spikes to remove risk from a project as early as practical in the project. The code from the PoC will be used as a reference for upgrading a bots capabilities
- **MVPs** will not be used as we will replace the legacy system as close to like-for-like (L4L) as practical. We will lean towards deferring improvements to the legacy system until after the replacement system is on the new technology stack and in a far better position to consider changes
- **SPVs** will not be used as the integration complexity of a project increases as functionality is replaced (strangled) from the legacy system. We will consider larger, most times the whole, legacy system to be replaced

As crazy as that last point sounds, we are advocating a *big bang* approach. If a *big bang* approach can be objectively analysed, there are some significant benefits that could result. Here are some points to consider in favour of a *big bang*:

1. Each time a legacy system is divided (or strangled) many integrations are created

between the old and the new sytems. This boundary is slow to form and adds more risk. By minimising the number of integrations, overall time in the project can be saved

2. Traditional software teams (without a codebot) had to fulfil requirements by manually writing code. For these teams, a *big bang* would take far too long before they shipped something due to the manual work, making the project infeasible. A codebot circumvents this problem as many requirements can be fulfilled at scale
3. Under circumstances where the solution is not known, a *big bang* approach is not recommended because dividing the work into smaller parts gives more opportunity to explore and test assumptions (like for a MVP). In a modernisation project, the solution is known so the need to divide the work and explore, test assumptions, etc is far less
4. Taking longer by unnessesarily dividing a legacy system will expose an organisation to getting caught part way through the modernisation if the project is halted for external reasons (like budget reprioritisation). In other words, shorten the length of time so you don't get caught part way

Will a *big bang* be possible on every project? Most likely not, you may need to do a few smaller fire crackers but do as few as you can. Play to your strengths.

Software development methodologies

There are many different approaches to developing software and lots of talk about which one is the best software development methodology. Being a student of our industry, we have experimented with many and come to recognise there is a meta-methodology that could be used to describe them all. So, for Jidoka, it is important to set a few parameters and specify what parts of these other methodologies we keep, and what parts we leave behind.

Methodology	**What do we take?**	**What do we leave behind?**	**Summary**
Scrum	• Huddles • Ceremonies • Backlog • Deinitions	• Sprints • Story Points	The Scrum meeting pulse (huddles, ceremonies, etc) is amazing for communication and the attention to backlog refinement raises project quality. Constant sprinting is tiring and story points are a waste as they get converted

			to time anyway.
Kanban	• Kanban board • Checklists • Continuous flow	• WIP Limits • Specialised teams • Handover loss	The continuous flow of work across a Kanban board can far outpace other approaches that are stop/starting. However, the balanced queues of WIP limits lead to specialied teams with all the problems of waterfall reemerging.
DevOps	• Pipelines • Removing barriers	• Rules complexity • Burdensome governance	Breaking down the barriers between development and operations using pipelines to automate and scale is empowering. But some practioners use this to enforce overly complex company policy and can rebuild that wall that was orginally torn down.

Project types

To focus the methodology on only some projects, we have grouped projects into three different types. Even though Jidoka has application in a wider scope, our hypothesis is that the benefits of automation with a human touch would be most recognised in these types of projects.

Like-for-Like

One type of modernisation project is a like-for-like (L4L) replacement of the legacy system.

- The solution is known as there is a system already in place
- The legacy system has become a liability to the organisation so there is a need to replace it
- The initial replacement of the legacy system should avoid changing any organisational processes where possible
- Avoiding organisational change will increase the likelihood of a successful modernisation
- Retraining of staff will be minimised as the new replaced system will be familiar
- Changes to the new replaced system should only be considered after the replacement is completed
- These new requests will be satisfied far more efficiently given the new technology stack and environments

Spreadsheets

Spreadsheets are a common approach for organisations to use as they have a low barrier to entry.

- The solution has been developed into a spreadsheet so many of the complexities around the data and its formulas have been solved
- The spreadsheet has become a liability to the organisation as it cannot scale due to spreadsheet limitations
- The initial replacement of the spreadsheet should avoid changing any organisational processes where possible
- Organisational change around modernising a spreadsheet compared to a L4L replacement is a higher risk
- Retraining of staff must be carefully considered as the new system will not be familiar and they will have loss aversion for the old spreadsheet
- Changes to the new system should only be considered when there is an identified gap in the spreadsheet or business process that causes a new safety issue
- These new requests will be satisfied far more efficiently given the new technology stack and environments

Disparate systems

Disparate systems are common as organisations organically grow over time. The systems can be digital or manual systems and have been built without considering the whole.

- The solution has been built piecemeal so that some of the complex problems have been solved
- The disparate systems have become a liability to the organisation as it cannot scale due to the lack of cohesion
- The replacement of the disparate systems will likely incur changes to organisation processes as multiple systems are considered
- Organisational change must be carefully managed
- Retraining of staff across the organisational change must include regular communication channels and feedback mechanisms
- Changes to the new system must be considered meticulously as the downstream effects will slow the momentum of the project
- If changes can be deferred, these new requests will be satisfied far more efficiently given the new technology stack and environments

Process

Over a series of milestones, the team diverges and converges on goals by following a set of principles to deliver both functional and non-functional requirements. There are 3 stages to the process (each consisting of 1 of more milestones) that carefully consider the current state of the project:

1. **Discovery**: In this stage we lay the foundations for success and formulate a plan that the whole team is confident will work.
2. **Modernisation**: In this stage we execute on the plan and play to our strengths by modernising the legacy system with as little change management as practical
3. **Optimisation**: In this stage we ensure the system is monitored and remains secure, while incremental improvements are now unblocked as we are on the modernised technology stack

Stages

Some important dot points that highlight a stage are:

1. Discovery
   - Diverge to converge on a plan
   - Use the activity kit to build knowledge
   - Set governance and standards
   - Tech spike high-risk issues
   - Establish a backlog of functional requirements
   - Set goal posts for non-functional requirements
   - Promote, fork, merge a bot plan
   - Create a communication plan with the user base
   - Do at least one milestone

2. Modernisation
    - Excute the plan with one or more milestones
    - Harden the target environments and montior:
        - Develop
        - Beta
        - Prod
    - Emphasise the data-driven pipeline reports:
        - Testing
        - Code Quality
        - Security
        - Performance
    - Communicate milestones to users
    - Adhere to governance and standards
3. Optimisation
    - The DevOps checklist is the standard that must be kept
    - Keep pipelines green
    - Monitor the target environments
    - Bug fix
    - Upgrade and patch
    - Make improvements
    - Open ended milestones are allowed without an end date
    - Adhere to governance and standards

Milestones

- Each stage can have many milestones
- Milestones have goals with date ranges and we aim for the closest date
- Milestones are not fixed length like sprints
- If the squad feels it necessary, they may break a milestone down into a number of iterations (of variable length), but this is not necessary
- A kanban-style continous flow of issues is worked on throughout the milestone until the goal is reached
- Ceremonies, meetings, checklists, definitions, etc are all performed according to the plan
- The depth of details in an issue is proportional to the issue's risk
- We plan for breadth-first rather than a depth-first by taking advantage of automation, modelling and pipelines (read more below)

Most problem solving approaches use a depth-first search, i.e. a depth-first search starts at the root node and explores down the branch that shows the most promise and then backtracks if no solution is found. In a way, this is how **MVPs** and **SPVs** work. They approach the problem incrementally as the solution is not known.

Another approach is a breadth-first search, i.e. breadth-first search starts at the root node and explores all the nodes at the present depth before moving onto the next. In this way, we can approach a **L4L** replacement as the legacy system is known and we are not searching for something we don't know. Identifying this as a key difference presents an opportunity on how we approach a milestone.

By considering the whole legacy system for replacement, unlike the strangler fig approach of **SPVs**, we will have a better understanding of the gaps across the entire system, not just a narrow part.

In a milestone, we are:

1. Model and generate as much of the legacy system as possible
2. Analyse the gap and what requirements are not being satisfied
3. Look for patterns in the gaps that are common
4. Extend the bots capabilities to fill the biggest gaps
5. Anything one-off, we hand code
6. Go back to 1.

Estimations

There are many ways to estimate a software project and we have experimented with many and the Jidoka approach is to find **balance**. What we have learned is that it is very time consuming and can create over-estimations when using issues to estimate as they are too fine grained. On the otherhand, we don't want to use #NoEstimates as it has too much of a negative impact on expectation management. With this in mind, the high-level, balanced approach to estimations is:

- A milestone has a goal and the team uses MoSCoW (must-have, should-have, could-have, won't-have) to priortise requirements
- Estimations are perfomed at the milestone-level and not the issue-level
- The team must agree on a realistic amount of time they need to complete the milestone
- The team doesn't stop at the must-haves and continuously works Kanban-style on the backlog with the available time
- The squad lead and account manager will calculate an appropriate buffer based on the risk associated with the milestone and communicate this date to the product owner
- The team still aims for the closest date
- Nothing builds trust faster than finishing eariler and delighting a customer

Lastly, the account manager will be in charge of the estimations throughout the tender or bidding process so the team will inherit an overall time for stages 1 and 2. It is imperative that the account manager consult senior team members to gain advice on how big the project should be so there is enough time to ensure a high quality project.

- 8 to 12 weeks for stage 1
- 3 to 8 weeks for each milestone in stage 2
- 15 to 40 weeks for 5 milestones over stage 2

Principles

Last, and definitely not least, we can now unveil the principles that will be used to guide Jidoka projects:

1. Augment the intelligence of a centaur software team with a codebot
2. Treat everything as a model and do useful things with them at scale
3. Knowledge is understanding and must be recorded in its simplist form in a code repository
4. Pipelines lead to quality, increased automation, and so many other benefits
5. Address high risk issues early and take on the most challenging first
6. The admin associated with something is propotional to it’s risk
7. Ignore non-functional requirements at your peril
8. Avoid changes to the organisations business processes while modernising, leave it to optimising later
9. Live the company manifesto!
10. Find balance and only use these principles where practical…