How to Avoid Becoming an Agile Victim

David Argent

Salesforce

Agenda

Introduction

- Who am I?
  I'm a veteran of Microsoft and Amazon, with more than two decades of experience in online service delivery, primarily as an SRE and TPM. I'm also a battle-scarred veteran of Agile gone wrong, and I'm eager to help others avoid being victimized.

- A Brief War Story – The Control Plane That Didn't

A (Very) Brief Review of Agile

What's All the Fuss About Anyway?

What Agile Isn't

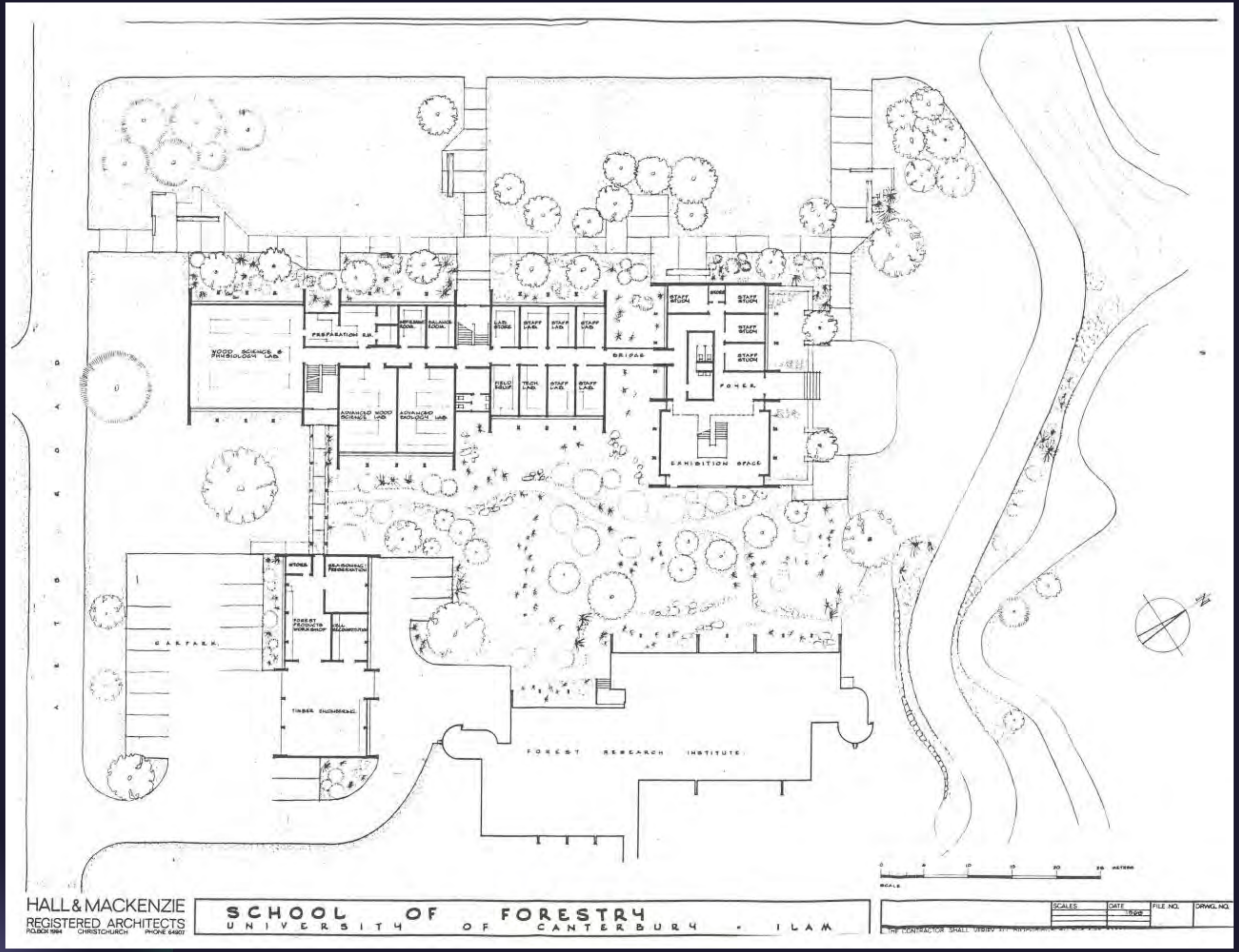
- A Framework
- A Methodology
- A Process
- A Set of Rules
- Prescriptive

What Agile Is

A set of principles which can help guide software development:

- Working Software over Comprehensive Documentation
  - The Built Software stands in as your "spec" rather than trying to document every little nuance
- Individuals and interactions over processes and tools
  - Self-defined teams with frequent face-to-face communication
- Customer collaboration over contract negotiation
  - Customer needs can change, and we need to flexibly adjust to that reality
- Responding to change over following a plan
  - Iterative – build small chunks of deliverable functionality at a time
  - Learn from each iteration and change course

But I Just Said Responding to Change over Following a Plan...



How To Fail

Just because you have priorities in your principles doesn’t mean other elements can be safely ignored

Failure 101

No Definition of Success

- How do you know if your product is successful?
  - If you can't measure it, how can you determine success?
- What is going to make people use your product?
  - Features, reliability, performance, security, privacy, etc.
- Why would people NOT use your product?
  - Lack of: features, reliability, performance, security, privacy, etc.
  - The very same things which can make people use your product, done badly, can convince them not to use it.
- What would cause you to stop developing this product?
  - Too costly to deliver the product
  - Too difficult to maintain
  - Too small a user base

Failure 201

Didn't Plan for Success

- Functionality is only **<u>one</u>** part of a successful product
- Few incremental features exist in a vacuum
  - Complex interplay between features, and they need to play nicely together
  - Overarching requirements to meet your definition of success
- "Invisible" items needed to deliver functionality and delight your users
  - Code testing (unit, functional, UI, etc.)
  - Data integrity, availability, and security
  - Availability / reliability
  - Limited downtime for maintenance or deployment of new code
  - Performance / scalability
  - Monitoring
  - Incident Management / Documentation
  - Disaster recovery / georedundancy

Failure 301

Didn’t Plan for Murphy

- THINGS WILL GO WRONG
- THERE IS NO SUCH THING AS A SAFE CHANGE
  - Diagnose failures quickly, automate responses to bad deployments
- Everything must accommodate failure
- This especially includes failures you have limited control over:
  - Partners
  - Network providers
  - Hardware
  - Datacenter Infrastructure (power, cooling, etc.)
  - Bad Actors
- Not enough to plan for failure, but how to recover from a complete outage
  - Coming up cold is often not the same as recovering from a partial outage
  - Determining responses to huge failures must happen before, not during an outage

Think Before You Code

The cheapest place to make changes is when you're designing and haven't yet written a line of code

Design 101

Anticipate Failures

- The only truly reliable thing in online services is that they fail – it's like death and taxes rolled up into one
- Understand and define what is 'acceptable'
  - Don't design for greater reliability than needed
  - Don't design for greater performance than needed
  - Both of the above add large costs and limited benefit to your customers
- Design for fast recovery from failures
  - Monitor and measure
  - Automate responses where possible
  - Include being 'hard down 100%' in your recovery scenarios
- Build degraded modes of operation for when you or dependencies fail
  - Support some user scenarios even in the absence of full functionality
- Multi-layered security
  - Anything is a potential single point of failure – make it harder on bad actors than that

Design 201

Anticipate Success

- The most dangerous time for an online service is when it's embarrassingly successful
- Architect for hyper-scale – You might need it!
- Avoid monolithic structures
  - Monoliths often cannot scale important subsystems independently, forcing you to overscale to compensate
  - Microservices and similar architectures allow components to scale more efficiently
- Avoid processes that scale linearly with people
  - For example, if customer onboarding requires manual steps, this can become a bottleneck if you are successful, since while your service may be able to scale, your staff may not
- Protect your service from excess traffic
  - Serving up some of your traffic is >>>> serving up none of it
  - You can't control client behavior or bad actors throwing excessive load at you
  - Identify your 'high value' traffic and service that when resources are strained

Design 301

Anticipate Change

- Nobody can accurately tell the future
  - Leave as many possible futures open as is reasonable
- Design for flexibility and possibility
  - You don't always know what the customer will want or need next year
  - Design with APIs rather than direct calls, enabling you to change underlying business logic without rewriting every component that needs it
- Zero-downtime software upgrades
  - Not all services will truly need this, but designing from the perspective that it must be possible safeguards your future
- Learn from experience and let it inform you
  - Change includes changing direction or plan in response to new information
- Couple loosely
  - Loosely coupled systems are generally easier to change and generally more resilient

Balancing Tactics and Strategy

How to concentrate on short-term deliverables while not sacrificing long-term vision

Tactics 101

Delay non-Critical Decisions as Long as Possible

- Cold feet are an asset
- Committing to a path and discovering you're wrong is expensive
  - Allow yourself to learn more about your problem space before making non-critical decisions you can't easily walk back
  - Leverage Agile principles and learn from experience
- Get the right people in the design process
  - Coders are very good at writing code
  - Architects are good at designing services
  - SREs are usually more familiar with the intricacies of running online services in the real world
  - Product Owners usually are 'the voice of the customer' and help define the requirements

Tactics 201

Try to Get the Critical Architecture Right the First Time

- Architecture decisions are often very expensive or nearly impossible to fix later
    - Do your research, understand the requirements, plan these out first
    - Everything else will usually align around the expensive architectural decisions with greater flexibility
- Get scalability, availability, performance, security, and data integrity requirements nailed down
    - Without at least most of these, you don't have a service
- Design initially to your non-negotiable requirements
- Adapt features and customer scenarios to your architecture
    - If it's obvious your architecture can't support your features and customer scenarios reasonably, redesign the architecture until it can

Strategy 101

Two-way Doors

- Two-way doors are decisions which can be easily reverted
- One-way doors, put you on a set path with no easy way to revert
- Sometimes, you need to take a step back to take three steps forward
  - This is ok! You're leveraging one of the strengths of Agile, where you learn from each iteration
  - Try to make your decisions reversable, so that taking a step back is easy and you aren't blocked from doing so
- As much as you need to design for flexibility, you also need to plan flexibly
  - Odds are, you're going to wake up smarter one day than you are today
  - Listen to and be able to implement the ideas of that 'smarter you'
  - Scrap and rework as needed without embarrassment – you want to get to the right endpoint, sometimes the path to get there is a little crooked

Strategy 201
How Much Planning Is Enough?

- Planning is necessary, but this isn't waterfall, folks!
- Don't plan every little thing – allow yourself to learn from experience as you go, what the 'right' things to do are
  - You will probably change significant elements over time based on what you learn. Good.
- Understand what you need for long-term success, and don't compromise on delivering it
- Know the shape of what you want to deliver, how you're going to deliver it, and how you're going to support it during its lifetime
- Realize that you're probably wrong about ALL THESE THINGS and be prepared to adapt to real world circumstances

Code Is Not Your Only Deliverable

Online services are more than just code, they include data, monitoring, testing, documentation, redundancy, availability, performance, budgeting…

Deliverables 101

Service code Is the Easy Part

- Service Code
- Not Service Code
  - Monitoring / alerting / incident response
  - Internal documentation / runbooks
  - External-facing documentation
  - Network
  - Security
  - Testing & Test Framework
  - Deployment tools
  - SLA / OLA / SLO
  - Administrative Tools
  - Reporting

Notice how there are so many more things which aren't service code, necessary to running an online service well?

Deliverables 201

Integrating non-Code Deliverables into Planning and Execution

- Add non-Code items to your backlog
- Understand what you need to successfully release
  - What is a release blocker?
  - What is my non-code collateral?
- Create a release sprint
  - Some things you only need to do if you're releasing – this is much of your non-code collateral
  - Processes often need to change around major releases, since a much larger percentage of time is spent doing bug fix work from newly discovered bugs not in the backlog
  - Allocate time differently
    - Ignore non-blocking code items from the backlog, concentrate on bug fixes and non-code deliverables
    - Test, test, test – this includes game days and breaking non-Production to ensure your runbooks, monitoring, and incident response are solid
    - Test some more, doing deployments so you know what to expect before and after
    - Train your operational staff and vet the documentation

Summary

Even with the principles of Agile, you need to do some planning to ensure the long-term success of your product.

Architectural defects and deficits in redundancy, availability, monitoring, and performance can destroy trust in your product weeks, months, or years from now and can be difficult/expensive to fix.

Thank You

David Argent

dargent@gmail.com (Personal)

dargent@salesforce.com (Work)