DDD CQRS & ES

– LEE HAMBLEY

DOMAIN DRIVEN DESIGN,
COMMAND QUERY
RESPONSIBILITY SEGREGATION
AND EVENT SOURCING.
AN APPROACH FOR ROBUST
SCALABLE BUSINESS APPLICATIONS.

Table of Contents

Prologue

Writing a technical book is a massive undertaking, why bother when there's already more good reading material around than one could possibly hope to consume in a lifetime? What makes me, and my opinions so important that I put pen to paper and write a book trying to tell you how to do things? Am I mad? Maybe. At least I've worked in this field for a long time, work a variety of hats and have my own perspective to share on how we might better serve the needs of our customers and improve our quality of life. If through my work I might help people better communicate their thoughts and build bridges of understanding, then I consider this endeavour worth while.

About the author

I've been writing code for as long as I can remember, I got my start cheating at video games, downloading pre-made shareware tools from the internet, and decompiling Lua scripts for the behaviour of units in an online RTS game circa 1997, from there it spiralled. I did my first bit of *programming* for the web to set up a website to share my mods, and slowly but surely developed a solid skill set that saw my landing my first real programming job migrating a travel application from Perl to Ruby on Rails in my early 20s around 2005.

I came to programming without a formal education and have occasionally dreamt of going to University one day to fill in the blanks, but I wonder if the "classical" comp sci way is serving our industry well at all - probably the answer is "it's complicated" and we need people with academic and real world educations and to stop pretending that one is better than the other.

Through a short-lived career working on other people's projects, in 2012 I started my own agency and we continued to work on other people's things for a number of years whilst bootstrapping a SaaS startup, my role in the agency spanned project management, product ownership and technical endeavours, and starting our own product on the side put me firmly in the "product" camp for a time, though I was still writing code nearly daily.

Throughout it all I've maintained Capistrano, a Ruby language deployment tool that was, and continues to be heavily used by the Rails community to do the heavy lifting and coordination of servers and resources during deployments. With the ebullient adoption of containers growing seemingly unrestrained, Capistrano is gradually fading into historical obscurity. The change of approach weighs heavily on my heart, as I've always had a soft spot for the "classical" hardware side of operations, having spent many a fond day locked in an aggressively air conditioned data centre battling errant firewalls and hardware problems.

The opportunity to see things from the nuts-and-bolts day to day implementation at the hardware or software level, through to the challenges of raising money and touring Silicon Valley speed dating investors and negotiating contract terms and international law, via the often overlooked "management" layer in the middle has given me a fairly unique perspective on how we could maybe simplify things and approach building software products with a little better understanding and a lot more happiness.

Introduction

This book will first attempt to outline the perceived problem in the approach to building modern software products and will touch on some of the key things that fail.

The benefits of LSAA are numerous, and I will attempt to constrain this first section of the book into a very high level economic view of the biggest wins, the smaller wins (easier testing, more rapid prototyping, etc) will be addressed throughout the book.

Nihil sub sole novum

"There is nothing new under the sun."

There's nothing new in this book, no new programming language, no new formal mathematical proofs, nothing original.

What is in this book is a "batteries included" world view and a bit of framework code to help cut down on boilerplate code in your own application.

LSAA is a re-imagining CQRS & Event Sourcing, two techniques that go hand in hand to solve a decent chunk of the problems I have endured with modern application development.

CQRS and Event Sourcing are often championed by Greg Young, philosopher and programmer who has great insights on the "hows" and "whys" of developer team performance, moral and more, I humbly submit that my work "completes" his.

Greg's company produces a database tailored for storing event sourced data, but using that database effectively requires a certain amount of buy-in, and a leap of faith - my company tried and failed to adopt CQRS a couple of times before coming to a pitfall-free solution (or, at least we better understood the economic impact of the pitfalls, and deemed them acceptable).

By providing a "batteries included" "framework" my hope is to cultivate the kind of environment that Ruby on Rails enjoyed circa 2007 when a complete solution with a philosophy, a strong leadership position, opinionated, but well formed arguments and a healthy dose of code generation and best practices helped propel Rails to it's global popularity and advanced the field of web application programming in ways that we still profit from today. Just as Rails was extracted retroactively from a successful product, so are large parts of retro, our LSAA framework.

It's a lofty goal, but I believe to my very core that this approach is worthwhile and produces better software for less cost which better serves it's business.

Most of the techniques we'll be drawing upon date back twenty years or more, they are concepts which you may have crossed paths with and never explored, it is my intention to show that when used together they deserve a place in most, if not all modern applications of *virtually* any size.

~~This book will walk through building a relatively simple application, just enough to grok the three techniques mentioned above without getting bogged down in the contrivances of the hypothetical domain I selected as a demonstration project.~~

What problem do we intend to solve?

Software product development practices can't be disussed without can wandering into the territory of cooperation modes, "Agile" vs. "Waterfall", etc, and it's here I'd like to start describing the bigger issue I've always faced working with software products, and that is that according to virtually every description of agile processes I've ever seen "Feedback" is as critical as planning, building and launching, and it's nearly always completely left out of the discussion, with a 3rd party solution such as Google Analytics, or KISS Metrics, or Segment.io bolted on as an after thought, and rarely, if ever tested for.

I've worked in many businesses that treat engineering as a simple technical translation of the requirements, as written in a ticket into executable code; I think this is a mistake, it puts the people with the biggest potential to have an economic impact at a disadvantage and in a weak position.

Giving engineers insight into what they are building, and why, and how the business is expecting the new widget to be used equips them with an understanding to create an implementation faithful to the intent, not dogmatically adhereing to the description, as a result, I would love to see most companies having productive discussions about KPIs and the adoption and impact of changes to how a platform behaves - this goal gives us a global objective "have enough data to fuel these conversations".

This book will talk about one way to achieve that, by completely re-imagining the way we build appliations...as a side effect, if we get it right, we'll have blazingly fast applications which give us a wealth of business intelligence of that our competitors can only dream about, and an architecture that will give us incredibly low running costs whilst allowing us extraordinary hortizontal scaling avenues.

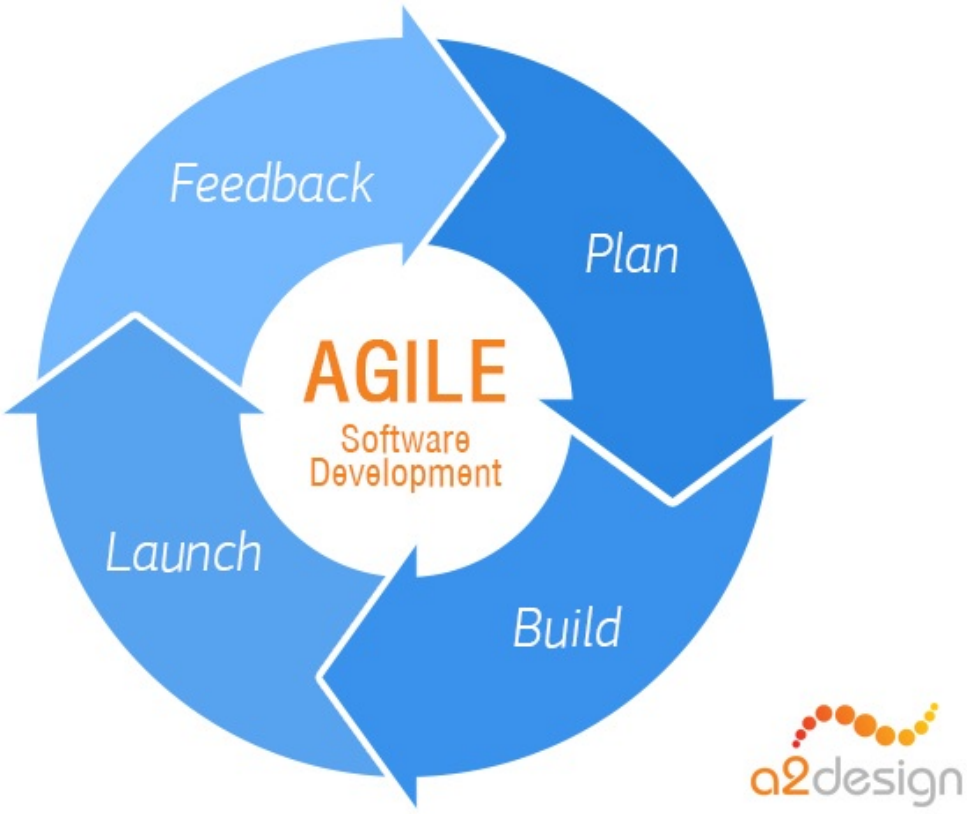
It was whilst working for a German client who explicitly demanded we not connect ***any*** 3rd party tracking software to their system (a misinterpretation of the german data protection laws, as it happened) that sent me on a journey for how to close the feedback "loop" without 3rd party tools.

I was glad of the excuse to finally invest in figuring out how to close the loop without relying on 3rd party integrations after many years of retroactively bolting on tracking after-the-fact, or finding ways to shoehorn audit trails and customer behaviour tracking into an app only to watch it break down and stop tracking, or never start tracking *properly* in the first place.

I've spent more time than I feel comfortably admitting bolting on well-funded SV SaaS "reporting" and "insights" tools, only to observe that they don't tell us enough, in enough breadth or depth about what people are doing with a piece of software

Incorrect, or incomplete data is probably worse than *no data*, and I've seen far too often companies reporting their "insights" up the chain of command that has a limited connection with the "facts on the ground".

Above all all "after the fact" tracking mechanisms that are bolted on the side, shoehorned in on top, or underneath or otherwise hot-glued onto an already running platform are rarely built with a holistic schema or plan in mind, leaving them fractured and incomplete, not only in breadth, but in their inability to be compared to themselves before they were implemented, that leaves a "lead time" after you implement a new tracking mechanism before you can profit from the data, and by extension leaves us with an incomplete circle, at least some of the time.



The product owner/manger in me *longs* for a day that I'm informed by default, in a way that enables me to make better decisions day to day, without tasking already overloaded developers with building infrastructure to let me answer my own questions about how things are working and being used.

The developer in me wishes that I'd be able to build more things that land in user hands, that are more performant and predictable, and mean that when I'm wearing the virtual pager, I can probably still get a decent night's sleep.

The ops guy in me wonders if it's possible to have nearly perfectly predictable (high) performance, and reduce the amount of infrastructure we need to manage, perhaps to nearly zero. Paradigms such as FaaS and "Serverless" are ripe for exploitation if we can architect our applications in a way that lends itself to the task without regressing to the dark ages of ops as it existed in the late 90s and early 00s.

Not only for web applications

This book will focus on modelling *business rules. A* project domain without tying itself to build a *web* application. Most of my relevant experience stems from the world of web applications, but increasingly the traditional "web" is becoming a 2nd class concern. We'll be building "for the web" though, whatever that means in your industry.

This position may seem strange when you look around. I certainly feel like we are enjoying a renaissance of *web* technologies, with an explosion in popularity of back- and front-frameworks.

We are however also in the age of chat-bots, and AI-assistant systems which have non-traditional interfaces. We also can't escape the ever accelerating rise of big data, machine learning and autonomous behaviour classification, by making behaviour our schema and building applications that derive their current state from a list of historical facts, we're setting ourselves up for success here too.

With that in mind, the approach we'll take is simply to model *behaviour* and *state* for a start, and then to talk in the second half of the book about specific interfaces and how they relate to the various mediums through which people consume your application.

For ease and familiarity, as well as practical reasons (I expect nearly everyone reading this book to be building software for the web, in one form or another) we will build a web application.

~~The realisation that the **web** part of your application may not be as cast in stone as you thought will force us to consider whether traditional web-concerns (sessions, pagination, telemetry) are *actually* part of our business domain.~~

~~We'll also touch on the thought of *state boundaries*, and what exactly constitutes “your application”, when caching and considering what out of data information to update. This will help us, for example to build elegant, maintainable solutions to traditionally messy parts of an app, such as live-updates in a browser, if someone is using one to access our application.~~

Version Control Systems

Version Control Systems (VCS) are something that many of us take for granted, I don't expect there to be many people reading this book who haven't used at least one version control system.

All version control tools, in some form or another keep track of changes to source code, some behave differently to others, but fundamentally they are all systems which store changes over time, and can be used to perfectly reproduce state at any time in the history of the storage medium.

One thing they **all** have in common is that we can travel through time, and examine how our collection of tracked things (files) looked at any point of time. Many source control tools even allow us to insert checkpoints (Git names them tags) so that we can easily checkout a point in time that has a significance for us such as a known good copy, or a "release" that we packaged and shipped as such.

One can query Git or Subversion "give me all changes in that file up until a given time", this gives us incredible flexibility in figuring out who changed what and when, with what cadence we worked on things.

Use of version control systems is so pervasive that it has become the de-facto norm that the question is no longer whether one uses a VCS or not, but ***which*** one.

Great artists steal

We can build our applications in a way that will have the same capabilities, out of the box, forever, as a foundational fundamental principle. From a developer perspective this gets me excited, but more importantly as a product owner/manager being able to access to the following capabilities, by default feels like I just discovered *fire*.

- Generating correct invoices from 6 months ago reflecting the customer's effective billing address, even if they've updated it in the meantime.
- Sending daily summary emails of missed messages and interactions with other members on the platform to the user via their preferred channels (notifications, etc)
- Answer questions like:
    - How many people remove an item from their basket in the five minutes before they check out?
    - What percentage of users confirm their email address within a minute of signing

up?

When we bolt tracking mechanisms on after the fact, the maintainability suffers, many of us have lamented the tracking and monitoring for a given service breaking down because it's not essential to the function of that piece of software, and it has bit rotted.

There isn't much vocabulary we can borrow from version control systems, but as a crutch to understand the approach they are invaluable.

~~Version control systems also teach us something about CQRS, that is that you **command** a VCS to do certain things "add this file", "store a specific change", you write (commit) messages to justify your actions to your colleagues. The *query* side however has a totally different interface, you "query" a VCS by having it lay out files in a directory so that your editor and tools can work with them.~~

~~The domain model of the VCS is **changes**, but you're interested in **files**. The same can largely be said of business domains, the *business domain* may be interested in profiles, users, accounts, private messages, appointments and shared cat pictures; but the user is interested in seeing their news feed summary, and having some notifications to highlight the most interesting things they missed and keeping up with their friends, or having a very effective experience whilst shopping online.~~

Disappointing Depth

In discussions about what to change, I find it helpful to frame where we are currently. Let's look at this from the lifecycle of a typical user who enters something into their browser, and then peel back the layers to see what's really going on "under the hood".

Your visitor doesn't buy a widget...

1. Enters a search term into their browser address bar and whilst they type the browser's "omnibar" makes suggestions, the user might click one of the suggestions.
2. The user lands on a page of search results relating to their search term, and selects a site to visit, if you're lucky, it's yours
3. The user makes a request to your website, "HTTP 1/1 GET /offers/super/special"
4. The user is presented with a list of unbeatable deals and clicks through to your "Buy" page and … *poof*
5. .... what the hell happened?

This fictional user's journey intentionally starts outside our control, because they so, so often do on the web.

Steps 1&2 are outside our control but serve as a userful starting point. For the search provider knowing what a user typed and how good the search suggestions are is super, super important, so they almost certainly track what was typed, how quickly it was typed, how quickly their servers presented suggestions and what suggestion in what position was clicked, and, eventually whether or not this was the user's last search before clicking through to a real site. The search provider is really collecting a huge amount of data from something as innocuous as a search query *autocomplete*.

Once the user lands on *our* site, a number of things happen, their browser will forward us any cookies they have for our domain, and a bunch of metadata about how they got here, usually in the form of headers with hints about where they came from (HTTP header "Referrer") or if not at least (usually) some "query string" parameters hung on the right hand side of the URL often giving a hint about where they came from.

In general, we as a collective industry ignore most of this data. We maybe take a peek into the cookies and check if we know this user's identity already and then we grab a handful of records out of a database and render them to HTML or JSON and send them down to the user's browser or device.

Usually something we sent to the client will give them enough data to make a follow-up request if they need more from us, the "next steps" we expect or hope the user will take, and if/when a user clicks through to the next page, the story starts again, we have a LOT of contextual data on how and where they came to that page, indicating for example in this case they they navigated first via our homepage rather than directly from the search engine to the particular offer, and we discard nearly all of it.

At the point the user abandons their journey with us, we know nothing, nothing useful at least, we see no more requests from a user in any form, and must assume that they've gone "cold" and left, we don't usually have a lot of data about what if anything they did or didn't see on the browser, and we're *usually* not even storing the number of people who *looked* at things in our database.

Barely scraping the surface

Even this technical explanation glosses over grossly complicated systems in the browser, operating system, network hardware of the machine, networking hard- and firm-ware of the network devices that stitch the user's favourite coffee shop to an internet backbone and out to the data centre hosting the application, and back again.

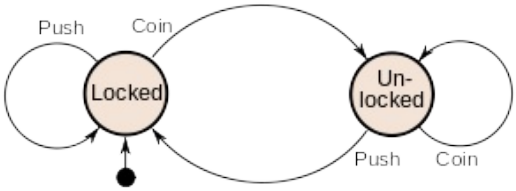
The specifics from the example above, of what we store and don't store aren't super important. I selected this example to show that there's a great deal of contextual data floating around, in the requests themselves of the spacing of them over time that other people (namely search providers who profit massively from this kind of data) are managing to extract much much more value out of.

At best most applications today store the "end results" and lose all the *context*, both breadth and depth.

The fact that internet hosted applications work as well as they do feel "real time" at all is nothing short of a miracle when one considers the sheer number of queues, buffers and proxies involved.

Welcome to the Machine

Consider a simple machine, anything, preferably something from the real world with moving parts and discrete states, how about a simple coin operated turn style for a fairground ride:



This machine has two states, to extend our analogy of the widget buying user a little, the"arrows" contain all the business value here (how hard do people push, how long does the gate stay open, how many people per hour travel through the gate, etc), whilst the circles let us know what paths it's possible to take.

Attaching labels to these things we can say that the arrows are **transitions** and the circles are **states**.

The turnstyle state machine is very limited, but it's as simplest one possible. One can push or pay, real state machines tend to be hierarchical and usually a bit more complex - you can imagine a toll bridge for cars, the toll bridge itself contains multiple such gates, and the coin receiver mechanism as a "sub" state machine that can be in the state of "nothing", "coin rolling", "coin falling", "coin approved", or "coin invalid", etc. Finding a level of granularity that allows you to have meaningful conversations about what's going on is easier than it may seem given that *everything is just state machines all the time.*

With these tools and visual aids, you can imagine dreaming up a state machine for a user, or product in a software system and then documenting the transitions from one state to another, think of a user being anonymous, signing in, being reminded that they have not accepted the latest Terms of Service, etc, products being "available", "shipping", etc.

I found when I started thinking of things as a series of discrete states, and transitions between them, I started to realise that everything that ever gave me a "wow" feeling using some software came from it reacting to what things were doing, not how they looked at any given period in time:

1. An email when someone adds me as a friend on a social networking platform.
2. A push notification when my item ships from an online retailer.
3. A reminder when the TOS has been updated on a site I frequently visit.
4. Changes in pricing for something that was in my basket on an e-commerce site.

This was a sample of the DDD-CQRS-ES Book by Lee Hambley via dddcqrsesbook.com. To get notified when this book finally publishes please visit dddcqrsesbook.com and join the mailing list. Follow @leehambley on Twitter for updates.