Domain-Driven Design on an Agile project: How it helped to implement a loyalty program

KACPER GUNIA, Domain Centric

In 2016 a global retail company decided to replace its off the shelf loyalty software with a bespoke solution. This report describes their journey and outlines how Kacper Gunia helped with adoption of Domain-Driven Design practices. These deliberate design activities helped them to implement and release to production a large-scale microservice system.

1. INTRODUCTION

About 3 years ago, a global retail company decided to replace its package-based loyalty system with a bespoke solution. It was critical to do this so that the company could provide a unified experience to all their clients, and enable the rapid delivery of business value that the business and members have been asking for. In order to succeed initially they had hired a big consultancy, which failed to deliver working software. In order to address the lack of progress, the consultancy was replaced with an in house development team of which I was a part. During the first 6 months we managed to rescue the failing project and release it in the first market. This gave us a mandate to start work on a global loyalty platform which was released to production 2 years later.

My name is Kacper Gunia and my role on the project was to lead and facilitate the implementation of Domain-Driven Design [Evans] practices. I have been training people and using these practices as an independent consultant for 5 years. In January 2016, I founded a DDD London Meetup where together, with hundreds of other practitioners, we regularly meet to share and learn from each other. In this report I share how DDD helped the company to design a fit-for-purpose system capable of handling hundreds of transactions per second for millions of members. I'm going to mention a number of techniques and tools in this report. Unfortunately I will not be able to explain all of them in detail, so I would like to encourage you to follow the links and learn more about them.

2. BACKGROUND

In the fall of 2016 the company started working on a new digital platform called Galileo. The first project— a loyalty platform for a small market—was an MVP (minimum viable product) for the organisation meant to prove that the company was capable of building bespoke business software. This was an interesting challenge, as not only the business requirements had to be translated into working product, but also the teams had to be formed and trained to use the new technology. That meant that the team had to work out their way of establishing a set of software development practices.

The director of engineering at the time realised that in order to deliver the mission, the teams needed some way of getting a common understanding of the software that was going to be designed and built. Event Storming [Brand] was one of the techniques used by the Domain-Driven Design community that was getting traction at that time. Event Storming is a collaborative modelling technique that allows people to explore the problem space using sticky notes, markers, and a long wall.

3. MY STORY

3.1 Fall 2016: Joining the Galileo Platform and building the Loyalty MVP

The director understood the value of this approach and started running sessions with his teams so that they could gain that common understanding of the platform they were going to build. It was about the same time as I was looking for a new project. Given my previous experience with Domain-Driven Design and Event Storming I was hired to help the team with this task.

Author's address: Kacper Gunia, Domain Centric Ltd, UK ; email: kacper@domaincentric.net

One of my first assignments after joining the project was to help people facilitate these already ongoing sessions. This was when we faced our first challenges. Event Storming can work well when attendees are in the same room and can explore the problem space in parallel. The moment some people are remote the benefits of the approach are gone. It is important to mention that at that time most of the people were based at the company headquarters, but there was also a large group of people working on the project that were based in North America and Europe.

We tried making people remotely join the sessions happening in the home office, but it was hard to contribute that way. I remember the struggle of sitting in my home office in the UK and trying to see what people were writing on the sticky notes. I was asking colleagues to read the events to me or put my ideas on the wall. Overall my impression was that the sessions were helpful, but we did not realise all the benefits of Event Storming because we did not have all the people in the same room.

Another approach I tried was to run a remote session that was focused on one specific area, defining and configuring program rules. This session was only joined by the only fully remote team at the time. We used an online whiteboard to collaborate on the events and a video call to be able to have a conversation at the same time. The challenge this approach presented was that an online chat is very much single threaded so usually only one person can talk at a time. Also it is hard to see how the board changes and impossible to pick up nonverbal signals from the participants. It was the only time we ever tried a fully remote session like this.

The outcome of the modelling sessions was then used to divide and assign work to the teams. The people involved in the project had a good common understanding of a problem space so the next step was to choose who should work on which part of the project. The primary heuristic we used was to align teams with business capabilities. These included redemptions, accruals, changing levels, expiration, and so forth. Although aligning with capabilities seems reasonable, focusing on that alone resulted in ignoring some other important factors such as aligning software with teams to minimise handovers or consistency requirements, to name a few.

A couple of months after I joined the project, not only was it going through challenges of creating a design for a software system, but also through challenges related to structure of the teams and their geographical distribution. The team I was working with was built out of remote consultants hired to boost the inhouse's teams' knowledge. My team was part of a larger group of 4 similarly sized teams, where 3 others were based in the main office. We were 8 people from all over the world working together as one team, but unfortunately it was not an effective way of sharing our skills with others. Because our team was working remotely, we became quite isolated as we had few interactions with other people.

The third challenge of our remote team design was the fact that the whole team was built of strong personalities and we were all trying to figure out how to work with each other. Forming a new team can be a frustrating experience as everyone works on establishing communication paths and finding their position in the group. That included me. Some of my ideas were not taken seriously because I did not have an established position in the group.

A few weeks into the project, the leadership decided to change how teams were structured. Each new team had a mix of people working on site and remotely together. This new structure helped to create more effective communication paths and enabled distributing the knowledge and skills so that more experienced people were mixed with people ramping up their knowledge. The 4 Scrum delivery teams (each with up to 6 developers, a product owner, and an architect), 1 team helping with adoption of technology, and another separate platform team ended up to be the combination that delivered the platform for the MVP market.

3.2 Spring 2017: Kicking off the Global Loyalty

Even though the implementation and testing of the Loyalty MVP platform was in progress, I was asked if I would be able to come to HQ for a couple of weeks to lead the domain discovery. The company was already preparing for the next project—replacing an off the shelf, monolithic solution with a scalable, microservice platform. It was a rare situation where almost the same group of people was asked to build very similar software for the second time. The learnings from building the MVP platform for one small market were the foundation for building an even more ambitious platform to run a couple of the biggest markets.

Some might ask why a full rewrite and the big-bang release were on the table. After all, it seems a very risky proposition. Let me explain. The "legacy" platform originally used by the company was a packaged solution that was not able to provide new capabilities the business wanted to implement. It was a 3rd party solution that the company used and moving to another provider was not a viable option especially since the loyalty program had become very core for the company. Because the existing solution was a close-sourced software we did not have the ability to gradually migrate or extend it - we only had access to exported data. Trying to work around its limitations did not seem to be worth the investment. Instead a radical decision to replace the whole platform was made.

The work on the rewrite of Global Loyalty started a couple of months earlier, late spring as the product team was already gathering requirements and analysing the legacy, packaged solution. This allowed us to kick off the project with another Event Storming session. This time I led the session with a different intention. Rather than trying to get both common understanding and clues on how to design our software, we focused on making it a true Big Picture session. This exercise allowed us to see the scope of the challenge at a quite high level. One of the traits of the Event Storming workshop that is both a blessing and a curse is the fact that the view of the analysed problem space will be constrained by the physical space (length of the wall). Because of that, it is important to make sure that there is enough space to explore the important topics.

During the one-day Big Picture session we had around 20 people in the room. We ended the day with a rough idea of the subsystems we were going to implement. In hindsight, I think we would have benefited from spending more time modelling the domain more thoroughly, as there were some parts of the model that were quite superficial.

One of the benefits of understanding the big picture was that we were able to figure out how concepts belonged together. In Domain-Driven Design we call these groups subdomains—parts of the whole domain (problem space). Based on this understanding, we were then able to choose how to model software around these subdomains, and know where the boundaries should be. I am fairly sure that without the holistic view that the Event Storming session gave us we would not have been able to figure out the boundaries so well.

We faced a number of challenges during the EventStorming session. We spent time coming up with a design and exploring subdomains that were never implemented, because some people insisted on exploring them. On the other hand, there were some other areas that we unintentionally ignored, which eventually turned out to be a large and important part of the solution. From my perspective, next time I would make sure to invite even more domain experts to the session and also facilitate the sessions better. It might be challenging to deal with vocal individuals, but letting quiet people speak is very important. Often they have insights that should be discussed with the group.

Having the shared understanding on what the problem space was, we then had to decide how to approach the delivery so that we could provide value to the business and prove that the platform we were working on would be fit for purpose. In order to make the decision and prioritise the overwhelming backlog (it was a like-for-like rewrite, so our scope was known and large) I ran a prioritisation exercise. We've our case we decided to look at our goal from the perspective of core capabilities that a new system will have to handle and that helped to frame the conversation.

The three most important capabilities for us were enrollment of a new member, point accrual (from a single, most used transaction provider), and reward redemption. If we could deliver these three functionalities, it meant that we could support the majority of API calls being made to the system. This was very much in line with the Pareto rule that 20% of work delivers 80% of the results. The heuristic to implement most used capabilities first served us well, but it is not the only one that should be considered. By focusing on the usage of the capability we might focus on answering the easy questions, without addressing the difficult ones.

Having a direction set, we then decided to agree on what our "walking skeleton" version was going to be. We came up with a persona—Bobby—and talked about his journey of enrolling in the loyalty program, earning points, and then redeeming these points for a reward. This was our first milestone and we called it "skinny jeans." Approaching it this way helped us to identify which APIs and integrations we would have to support in order to deliver this first milestone. It also provided us with a very clear acceptance criteria and easy way for validating the work for the product team.

With a shared understanding of "what" needed to be built, we now focused on "how" we were going to build it. In order to get a better understanding of a problems space we ran a series of deep dive sessions, where we explored the model (events, commands, policies and aggregates) in detail. These "Design Level" Event Storming sessions helped us to explore in detail various scenarios and cases. Sessions like these were much more focused on exploring one specific sub-domain; we knew that getting the accrual and redemption model was very core for us. If we got it wrong members were going to be very vocal about it, which could damage our reputation. Each deep dive session like this took about two hours and during these sessions I facilitated the discussion between the product and technical team members.

Given our learning from the previous project we implemented for the MVP market, we decided to try out Event Sourcing [Fowl] as our persistence model (only for one sub-system). In this new architecture, the current state is always derived from (and persisted as) a history of events. Because our Design-Level Event

Storming modelling was done using events as well, there was little to no translation needed between the events we discussed with the business and the events that were implemented in software. In fact, the very events (such as "Points Accrued") that we modelled back in May 2017 are used in production today.

This decision allowed me to prototype and try out the newly discussed model very quickly. The discovery process was kicked off at the beginning of the week. At the beginning of the following week I already had a working prototype proving the model we came up with. The model consisted of a main Aggregate, multiple Commands, Domain Events, and also some Read Models.

In Domain-Driven Design an Aggregate is a pattern that encapsulates state and makes sure changes to this state are always consistent—we call it a consistency boundary. In our case the aggregate was modelling a single member account, similar to a bank account. The aggregate is then capable of handling a number of Commands such as Accrue or Redeem points. These commands are requests to change the state of the aggregate and can fail, e.g. we do not allow points to be redeemed if the balance is insufficient. Any successful change is then persisted as one or more Domain Events which are facts describing what has happened in the system. For example events like Points Accrued or Points Redeemed once saved are then used to update Read Models such as Current Balance which can be queried by the clients.

We also made sure that our new model would help us to mitigate the problems we had with the platform developed earlier that year for the MVP market. From my perspective, this was very empowering and allowed us to show that even with a very new architectural approach we were able to deliver value quickly. And we did not have to spend a lot of time on big upfront design or setup activities.

Beginning with a prototype and walking skeleton was a great way to start shaping our new system. I am a strong believer that the Agile approach to software development and focusing on validation and delivering value incrementally is the right thing to do. Unfortunately, many times it happens at the expense of the software design. I think it is because people skip this design step arguing that "big, upfront design" is the thing that waterfall practitioners do.

Paul Rayner, one of the Domain-Driven Design thought leaders in his "How Agile Can Cripple Effective Design Work" [Rayn] talk quotes Douglas Martin who says: "Questions about whether design is necessary or affordable are quite beside the point: design is inevitable. The alternative to good design is bad design, not no design at all." [Mart]. I could not agree with this quote more. A lot of times we hear that a project needs to be rewritten after some time because its design is rigid, not maintainable, or simply not fit for purpose. Frankly this is what happened to us too. The first system we put together for the MVP market was not feasible to be used globally. So this time we had to design it properly. The problem is that "properly" is not something we can easily measure or quantify, so finding balance between "no design" and "big, upfront design" is a fine art.

From my experience, for a medium size project like this, a 2-week effort to understand, design and implement a prototype is a worthy investment. During a project that took nearly two years to implement, 10 days of modelling was a fairly small amount of time to invest. In return it gave us a solid foundation and a common understanding of the problem space. To make this happen we had to invite business, product and technical leadership into the same room.

It is easy to say now that after years of working on the two systems we now know what the right design is. So how can we then maximise the return on investment and get most out of the limited time we have during the discovery phase? Based on my experience working on the two projects for the company discussed in this report and number of others I have consulted on in the meantime, I can recommend following process:

1. Discover and learn about the domain
2. Choose and validate boundaries of subsystems
3. Categorise subsystems and decide on levels of investment required
4. Design each subsystem in detail and decide on how it will integrate with others

Spending at least a couple of days on the first three steps will greatly improve chances of creating loosely coupled and evolvable software. Also it is not like we find the perfect boundaries for our system the first time we design it. Instead we should focus on choosing boundaries that can help us solve our current problems and make the system flexible enough so that we are not afraid to change them if we learn that some other choice fits our needs better. In my experience, flexibility is enabled by choosing system and module boundaries in line with expected rate of change, certainty, functional and non-functional requirements, and also with minimal dependencies. That gives us a better chance to add new capabilities without affecting existing ones, or to replace them if it turns out the implementation was not fit for purpose.

The fourth step, designing a subsystem in detail, is an iterative process during which we leverage both "Design Level" Event Storming and "Model Exploration Whirlpool" [Evanw]:

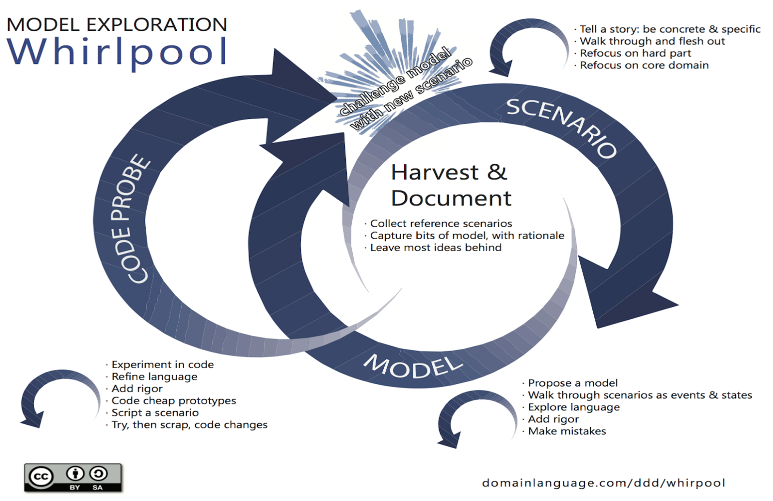
MODEL EXPLORATION
Whirlpool
challenge model with new scenario
SCENARIO
· Tell a story: be concrete & specific
· Walk through and flesh out
· Refocus on hard part
· Refocus on core domain
Harvest & Document
· Collect reference scenarios
· Capture bits of model, with rationale
· Leave most ideas behind
CODE PROBE
· Experiment in code
· Refine language
· Add rigor
· Code cheap prototypes
· Script a scenario
· Try, then scrap, code changes
MODEL
· Propose a model
· Walk through scenarios as events & states
· Explore language
· Add rigor
· Make mistakes
domainlanguage.com/ddd/whirpool

Figure 1. Model Exploration Whirlpool

The aim of this activity is to design and validate models quickly using real life scenarios. This is what served us well during the build of the platform for the global market. The initial model and prototype worked for the walking skeleton version, but did not support more complex requirements. So it had to be evolved. Each subsequent change in the model was done only to the extent needed to support the new use case. Because of that we did not create a big upfront design that would try to solve all possible problems.

In the end we ended up with a very rich and fit-for-purpose model, but this did not happen overnight. The initial model was created in two weeks, but it evolved, matured and stabilised over the course of a year. We continue doing the modelling whirlpool today and most changes that affect this model are prototyped first, then the design is presented to a wider audience and finally implemented and documented.

3.3 Summer 2017 - Summer 2018: Global Loyalty team transitions

Some time after we began working on the new platform for the global market, even more of the people were asked to leave their old teams and form a new one, Team Lightning. Lessons learnt from the project for MVP market made us focus on being a "remote first" team. We wanted to avoid making the same mistake of isolating people not based in the main office. It all started well, and as more people were finishing their MVP duties they were joining a new team working on the global system. What we failed to react to, is that the team grew too big. Within a year we grew to a team with nearly 35 people working on a number of different codebases, all with different sets of requirements but joining the same stand-ups and other scrum meetings. On top of that, some of the people that joined the team were from an outsourcing company based in yet another time zone. Because of the size of the team, the number of communication paths had grown to an unmanageable number. This led to frustration and people naturally clustering with peers they knew best or sat closely with.

So we had to change the structure again. This was not an easy decision to make, as the product managers were rightfully worried about impacting the deadlines and disrupting productivity. Fortunately the principles we followed were similar to our last transformation: reduce the number of people in each team to minimise communication overhead; ensure that each team is working on services that are aligned with the business domains; ensure that teams are not overloaded with work; and distribute the team members, so that each team has people from each location to enforce the communication and distribution of knowledge

We ended up with three development teams of 5-9 developers and testers, an enabling team, and also still relying on the platform team to make sure we did not have to worry about infrastructure concerns. All of that seemed to work. Similar (but downsized) three teams are still working on new features of the loyalty platform nearly a year after going into production.

3.4 Spring 2018 - Spring 2019: Global Loyalty data migration and changes in requirements

One of the major challenges we faced during the development of the new platform was how to migrate the transaction data from the old system to the new one. The old system was very generic and modelled the point accrual and redemption in a CRUD way using two entities that were serving many purposes using flags and statuses. On the other hand, the new model we came up with was much richer and represented these concepts using 6 different immutable events such as “Points Accrued,” “Points Redeemed,” or “Points Expired.”

There are a number of reasons why we decided to model these concepts explicitly. First, it makes the model easier to understand, test, and reason about. Second, the event based model enables us to provide true audit capabilities which are one of the key requirements for accounting systems. Third, it enables us to develop new features (such as dynamic expiration dates) that the business stakeholders were asking for.

Translating the old model to the new one was not trivial, as they did not align. Some entities from the old legacy system were impossible to migrate to events as they did not contain all required information. For that reason, we also had to introduce some events in the new system that existed only so that we could migrate data that did not fit into our new model. An example was a new event called “Tier Migrated,” which we had to introduce because we were not able to create normal tier events using available data.

One of the challenges with using an Event-Sourced system is that we cannot simply write data to tables that will be queried by services and APIs. Instead, as we migrate the data, the events have to be written to the Event Store and then propagated through handlers to read models that are used by the query side of the Command-Query Responsibility Segregation architecture. In our case we had about 25 different read models created from the event data that support various APIs, policies, and scheduled processes. The projection logic (translating the stream of events into a desired state required by a certain capability) was already written to be used for normal production operation. We were able to leverage this by implementing a new command called Migrate. This command translated entities from the old system to events in the new one (one member at a time).

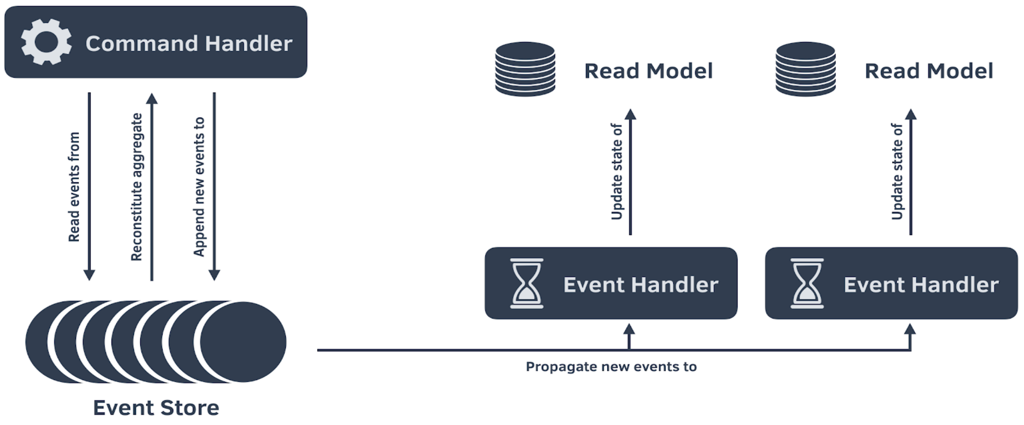

Figure 2. Event Sourced Architecture Diagram

After all the data was moved, we were then able to compare the results returned from old and new APIs and see if they matched. This allowed us to find and fix a large number of problems that would otherwise have surfaced after going live. We were also able to prove that either the data was migrated correctly or in cases where that was not possible, that we correctly compensated our members so that they were not worse off.

Another challenge we faced and had to solve was the introduction of the offers engine. Offers allow our members to earn extra points or get other benefits if certain conditions are met. Initially, the scope of this part of the system was quite small and we simply started building it into our solution. Since the project was under time pressure, going for the simplest working solution seemed to be a pragmatic choice.

In retrospect, I noticed that the new offers capabilities started polluting our core point calculation logic. At that point it seemed obvious that this subdomain should have been cleanly separated from other subdomains, as the initial approach resulted in accidental complexity and coupling. Because of the deadline on the horizon, I did not have the guts to ask the leadership to step back and change the design of the system.

The lesson and realisation I have now is that in order to avoid a situation like this we should have gone through our modelling process again. I know getting everyone into the same room is usually not a straightforward thing to do, but periodic review of the architecture and course correcting is something I would now recommend doing on a regular basis.

At that point, the leadership decided that the company would be better off integrating with an off the shelf solution. Because of that choice, we had an opportunity to clean up our model and ended up with a simpler solution. The third party solution integrated with our platform using public APIs. This allowed us to focus on the core capabilities.

The importance of aligning the business with software was also apparent on a number of other occasions. There was a situation that I remember very distinctly. Some time before going live the business decided to rename Tiers into Statuses. It seems like a trivial change but this had large impacts on the team. Throughout our codebase we had hundreds of references to the word Tier. The appetite to refactor it and change it to Status was small. No one wanted to introduce extra risk right at the moment we were getting ready to go live. Some time after we went live, a new developer joined the team. He was asked to make a change to how Statuses work. He looked at the codebase and was very confused as there was not a single mention of Status!

After explaining the situation and learning how the system differed from the business domain, he was able to make the change he was asked to do. Unfortunately it was the lack of "Ubiquitous Language" alignment that caused his confusion. If we do not pay attention to changes like this, the system and business will slowly drift apart. Luckily the aforementioned developer was determined to make it right. Some time later he made a change aligning the code with the language of the business, and by doing so set a very good example.

3.5 Spring 2019: Global Loyalty go live

Going live with such a big system that was 1:1 replacing an existing solution was a very complex operation. There were tens of teams involved across the whole company and our go live sequence had hundreds of steps, including a point of no return. The risk of something going wrong was very high. How did we make it work?

One of the best ways of making sure that a go live sequence goes as planned is to practice it. For an extended period of time all teams involved practiced their respective steps during "dress rehearsals." That helped a lot and gave confidence that friction points were identified and any missing steps were added to the sequence. The second way of reducing the risk was to run the new system in a validation mode handling live traffic, but without being the system of record. This helped us to make sure the new platform implemented functionality in the same way as the old did. We were also able to observe how the system behaved under load and do necessary adjustments. Once everything was ready, we started migrating the records from the old system. Due to the very large volume of data we had to do this in multiple stages. The first initial import took a couple of weeks. During that time the legacy system was still running, so we kept accumulating even more data. Because of that, a number of incremental imports were required to catch up. After both old and new systems were close enough to one another, the actual go live sequence was started. Finally during the summer of 2019 our new system was handling live traffic in production.

We had a working system that was providing value to our members and I am sure everyone was as thrilled as I was. I would be lying though if I said that everything went so smoothly and that we did not have any hiccups! It turned out that some internal functionality that was not thoroughly verified during data migration and parallel validation had bugs that we discovered only after going live. One of the problems that was caught within minutes after flipping the switch was the incorrect migration of some parts of the member data. A simple validation check would have highlighted inconsistencies, but it was not performed. Fortunately, because some of the problems were with logic that was translating events into final state, we were able to fix those bugs and make sure everything worked as expected. After that code was fixed, we replayed all historical events and recreated the state. This was done without downtime or any impact on our members or business. This replay ability was used a number of times and was well received by the business stakeholders. This would not be possible if we did not treat the events as the source of truth in the first place.

4. WHAT WE LEARNED

When I joined 3 years ago, I had no idea of the size and scope of the project I was signing up for. After a lot of sweat and hard work we are in production with a great system that is serving many millions of members and handling hundreds of transactions per second. But it would not be possible without very smart and open-minded people working on it. First, and for me personally, a very important learning is the fact that any agile project big or small should include deliberate and lightweight design activities (both at the beginning and during the project). If we do not do this we risk ending up with arbitrary boundaries for our software systems, which in turn causes coupling and makes software rigid.

Another takeaway is that if we are migrating any system to a new implementation for whatever reason, we need to understand the business domain first. Then we can evaluate a number of alternatives on how the software can help us to address these challenges and decide which one makes sense in our particular context. Only then, when we know what good looks like, can we decide how to transition to that desired state.

To gain that understanding we can use a number of techniques starting from Big Picture Event Storming, through Business Model Canvas [Oster], choosing service boundaries, then going into design activities with the use of Design-Level Event Storming, Bounded Context Canvas [Tune], analysing how messages will flow through the system, Aggregate Design Canvas [Gunia], and applying any number of design heuristics. Not all of these techniques were explicitly discussed in this paper, as some of them I've started applying, teaching or formalised more recently. I believe that if we do all these things we will have a greater chance of succeeding. But even the greatest system design is not going to help if it is not aligned with the teams delivering it. Making sure that the number of handoffs between teams is reduced is critical to ensure team autonomy. But in order to do that, we need a good software design in the first place. The feedback loops coming both from teams and software design should be constantly evaluated and used to optimise the work. That also means that the organisation needs to create an environment where people feel safe to fail, learn from failures, and then adapt and change their processes. This take a great leadership team to achieve, but based on my personal experience I can truly say in the light of my own experiences that the company has created such a culture and I am grateful for being able to work as a part of that team.

5. ACKNOWLEDGEMENTS

I would like to thank all the people who have worked with me on this project and report. Firstly, the Director of Engineering who initially hired me because he realised the Domain-Driven Design is valuable and would enable the successful delivery of this project. Secondly, the senior Manager who was very open to collaboration and made sure engineers were included in the product discussion. Thirdly, our second Director of Engineering who encouraged continuous improvement and created a safe environment where people were not afraid to speak up. He also helped greatly with reviewing this report and making it happen.

Success of this project would not be possible without my colleagues with whom I have worked on a daily basis. I would like to especially thank all product managers who were part of the teams that I have contributed to. Last but not least, all engineers that I worked with and who helped to implement this microservice platform. I have learnt a ton from all of you and definitely I am a better consultant than I was 3 years ago.

I would like to also thank all people from the Domain-Driven Design community who shaped my thinking around solving business problems using software. Alberto Brandolini for coming up with Event Storming and Paul Rayner for his inspiring presentation at DDD EU 2016. Nick Tune for his thought-provoking questions, pushing the boundaries of DDD, and together with Kenny Bass-Schwegler suggesting I write this report.

Rebecca Wirfs-Brock, shepherd of this report, tirelessly gave me feedback, nudged me to make progress, and commented on my ideas. I really appreciate her contribution and kind attitude that helped me to overcome the obstacles I've encountered along the way. Thank you, Rebecca!

Lastly, I'd like to thank my wife Daria and son Teodor. Without their support I wouldn't be able to do so many great things on this project and finish this paper. I hope this report will help you understand what I'm doing at work.

REFERENCES

[Brand] Brandolini, Alberto. *Introducing EventStorming.* Leanpub, https://leanpub.com/introducing_eventstorming
[Evans] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison Wesley, 2003.
[Evanw] Evans, Eric. Whirlpool Process of Model Exploration. Web page: http://domainlanguage.com/ddd/whirlpool/
[Fowl] Fowler, Martin. Development of Further Patterns of Enterprise Application Architecture. Web page: https://martinfowler.com/eaaDev/EventSourcing.html
[Gunia] Gunia, Kacper. Modelling aggregates with 'Aggregate Design Canvas'. blog post: https://domaincentric.net/blog/modelling-aggregates-with-aggregate-design-canvas
[Mart] Martin, Douglas. *Book Design: A Practical Introduction.* Van Nostrand Reinhold, 1990.
Osterwalder, Alexander. *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers.* John Wiley and Sons, 2010.
[Rayn]Rayner,Paul. video: How Agile Can Cripple Effective Design Work (and what to do about it). https://www.youtube.com/watch?v=Bl30X0MvT0I
[Tune]Tune,Nick. Modelling Bounded Contexts with the Bounded Context Canvas: A Workshop Recipe. blog post: https://medium.com/nick-tune-tech-strategy-blog/modelling-bounded-contexts-with-the-bounded-context-design-canvas-a-workshop-recipe-1f123e592ab