ELCA

We make it work.

WHITE PAPER

What it takes to Refactor a Legacy Monolith

This white paper describes the restructuring of a large legacy code base from the initial situation and pain points to the resulting architecture and benefits.

It is the story of how we approached the refactoring of a complex three-tier application presenting issues with performance, regressions and slow release cycles, towards a modular and scalable architecture, with clear APIs and automated tests.

It is the first step, often the most difficult one, to go distributed and fully enjoy the benefits of a web-scale microservice architecture.

Tags: monolith, microservices, modules, domain-driven design, software industrialization, automated testing.

ELCA
We make it work.

Table of Contents

THE PAIN POINTS OF A MONOLITH

The story begins with findings that will be familiar to most development teams. A flagship application developed in house to meet the most challenging demands of the business, had grown to an all-in-one monolithic solution that was difficult to maintain and evolve. The three-tier architecture model promised a clear separation of concerns between the user interface, logic and persistence. At best, it succeeded to preserve a clear distinction between these layers (at least at code level), but it failed to address modularity within each layer, leading to the infamous big ball of mud.

Under business pressure, unregulated growth and the on-boarding of new team members, the original application had derived to code that no one in the team fully understood anymore. Tracking data dependencies and assessing the impact of a change in reasonable time was so difficult that, along with the lack of automated tests, no one dared changing code deeply. This in turn led to code that is less and less intelligible.

The flexibility and agility that were some of the principal advantages of an in-house development were offset by time-consuming code archaeology and tedious manual testing. The team got bogged down in regressions and hot fixes on every release, to the point where delivering value to the business was becoming increasingly difficult.

In addition to obstacles in the development of new features, the monolithic application failed to scale to higher load and business volumes. Scaling out with an increasing number of copies of the application behind a load balancer was costly (due to additional resources and JEE server licenses) or limited by the increasing data volume to handle in some parts of the application. Indeed, different components with different resource requirements were all deployed together. It forced operations to scale up all the nodes to unreasonable amounts of memory heap, and led to other problems such as unpredictable performance during automatic memory management by the system (full GC pauses).

The flagship application had all the symptoms of a monolithic system that had grown to an ill-designed system that was difficult to maintain and evolve:

- Large intimidating code base,
- Tangled dependencies in the database and all the way up to the UI layer,
- Frequent regressions on new releases,
- Slower development cycles,
- Performance and stability issues.

Hence, our client decided to split their monolith and decouple the entire system.

This white paper explains how we are:

- Refactoring the system to a new server architecture that is both scalable and modular;
- Industrializing the development process with a new team organization and a strong set of automated tests.

REFACTOR OR REWRITE?

Such an endeavour always starts with the same question: is it best to refactor the existing system or rewrite it from scratch?

Refactoring is an incremental activity that can be done along business-as-usual activities, while rewriting is a big-bang approach. Rewriting the whole system is often not an option, as the old application is expected to live during the rework. Maintaining the old system and rewriting a new one from scratch at the same time would be an overly complex task and a drain of resources.

Nonetheless, even in the absence of operational constraints, rewriting the system is not necessarily desirable, as it may present more risks than the refactoring approach. Although it looks faster to just ignore all the inherited code and intricate dependencies, what are the guarantees that the new system will not present some of the same defects as the original one, or lack some of the original features?

In the end, each approach takes time, requires management and users buy-in, and comes with its own risks. Therefore, the best approach very much depends on the context.

Reengineering risks	*Rewrite*	*Refactor*
Feature freeze	×	
Integration risk	×	
Loss of business logic	×	
Lack of resources	×	×
Obsolete technology		×
Intractable dependencies		×
Long stabilization period	×	

In our context, sponsors of the project insisted on continuous improvements with no interruption of service. In addition, the backend in Java was to stay in the same technology. Hence we advised to refactor the system incrementally to minimize reengineering risks and spot potential functional or non-functional regressions early. The refactoring effort is synchronized with the business-as-usual evolutions, and planned to limit merge conflicts whenever possible.

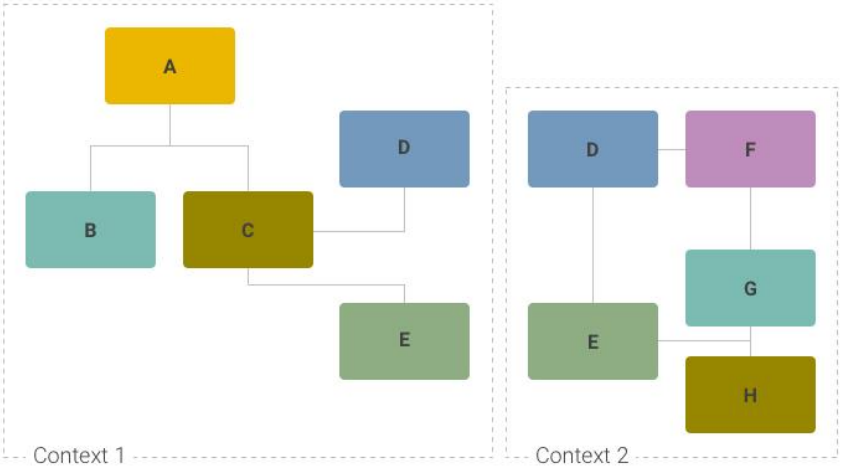
TURNING BUSINESS DOMAINS INTO MODULES

Modular programming is vastly covered in the computer science literature and its benefits well understood. The tighter the coupling is between software components, the more difficult and expensive it is to change things. Conversely, the looser the coupling, the easier it is to change one part without impacting the whole system.

In practice, however, it is very frequent that coupling grows uncontrolled, especially in monolithic applications where nothing prevents it. Most of the time, this is due to what Fred Brooks coined in the 80s[1] as *accidental complexity*. Brooks distinguishes the *essential complexity* due to the inherent complexity of the problem at hand, from the *accidental complexity* due to the lack of appropriate tools, modelling and knowledge of the underlying domain. Although there is no silver bullet to drastically reduce the complexity of software, there are methodologies to avoid bad design, and in particular to minimize coupling.

One such method is SIP for *Single Iterative Partitions,* introduced by Roger Sessions[2]. SIP provides a formal framework to decompose an IT system into smaller simpler partitions. It is based on component synergies from a business perspective and uses functional equivalence classes to reason about which features to group together.

This method shares similarities with modelling practices of *Domain-Driven Design*[3]. DDD also focuses on business first rather than on IT-driven considerations, and recommends splitting big models into smaller *bounded contexts* to confine complexity. As illustrated below, the same concept may end up in different bounded contexts, as it may have slightly different meanings in each context. The concept will be modelled differently in the different contexts, with a partial mapping between the two models for interoperability.



[1] Fred Brooks, No Silver Bullet – Essence and Accidents of Software Engineering, Computer 20, 1987
[2] Roger Session, The Mathematics of IT Simplification, 2011
[3] Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003

In both SIP and DDD, business domains are at the heart of the decomposition. And this is essentially where we started from to decouple the system into functional modules.

As with many systems that grew organically, there was hardly any documentation. Therefore we conducted a top-down analysis of the system to gather features, typical use case scenarios, and clarified some apparently similar concepts across different domains. We proceeded as follows:

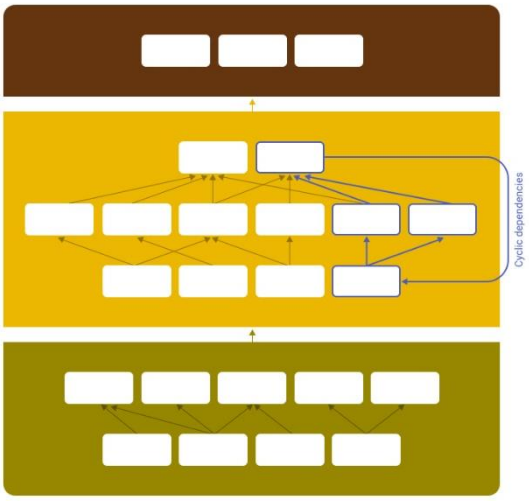
1. Make the inventory of all features via interviews and reverse engineering
2. Classify all features per business domain, operation modality (who uses each feature and in which context it is triggered), and main data that is manipulated.
3. Draw the high-level functional architecture, free from any constraints.
4. Incorporate functional and organizational constraints and refine the high-level decomposition into an exhaustive list of modules and sub-modules covering the entire set of features.
5. Take into account additional technical constraints, for instance related to the nature of data or to deployment, to propose the new technical decomposition.

For the last steps, we complemented our initial analysis with a bottom-up approach, taking advantage of the existing code base to better understand inherent dependencies, important sequence calls, key database accesses, and performance bottlenecks. This allowed us to confront our initial decomposition with technical considerations, and additionally propose the re-architecture of some core functionalities.

THE RIGHT TOOLS FOR THE JOB

We had to assess the effort to refactor to the new design, and understand how tangled the dependencies between the functions of the new modules were in the existing code. Indeed, although code untangling has been researched to assist refactoring[4], automating it at the scale of a large code base is simply not yet feasible.

There are some fantastic tools to help in this task, tools that show you the links between the different components of the model, and allow you to drill down into fine-grained dependencies.



One tool proved to be particularly effective: Structure101[5] that lets development teams visualize and organise their code base by moving chunks of code around to improve modularity.

Structure101 shows dependencies between each module in a layered representation. Modules are laid out into rows, or levels, so that as far as possible every item depends on at least one item on the level immediately below it - items in the same row do not depend on each other, and items on the lowest level do not depend on any other items at the same scope. In addition, strongly connected components are highlighted to quickly visualize cyclic dependencies.

This representation is much easier to interact with than the more common Design Structure Matrix (DSM) used by many code quality tools. In particular, Structure101 allows us to:

- organize classes into our own hierarchical model;
- use complexity measures to guide remodeling;

SIMULATE REFACTORING AND IMMEDIATELY SEE THE IMPACT OF CHANGES ON THE MODEL.

We used the tool extensively to explore, simulate and validate the feasibility of each refactoring. This way, we have minimized the risk of starting a refactoring directly on the code and getting stuck in intricate dependencies.

[4] Ran Ettinger, Refactoring via Program Slicing and Sliding, 2006
[5] http://structure101.com/

PRINCIPLES OF THE NEW ARCHITECTURE

The decoupling aimed of course at introducing clear and manageable module boundaries, but also at introducing strong design principles enforced through coaching and reviews:

- Strive for simplicity and write simple code to increase both maintainability and performance.
- Define clear, easy-to-use APIs between business services, libraries and UI components. At the same time, hide internal implementation details to reduce unwanted coupling.
- Avoid chatty communications, reduce payload and calls between the different layers from the clients to the databases, and between business services.
- Split shared databases and encapsulate data into their corresponding services.
- Favor asynchronous calls to blocking synchronous calls both in UI and in business services.
- Separate commands from queries whenever possible, as advocated in the CQRS pattern, and introduce a distributed data grid (here we recommended Hazelcast[6]) to maintain views of the data that are frequently requested.

Perhaps the most challenging principle to apply was to split shared databases. This is a major shift compared to monolithic systems that use shared databases for integration. For that purpose, prior to refactoring, we introduce a new database user per module to refactor, and temporarily grant appropriate privileges to both owned and accessed objects, even those in different bounded contexts. Then, during the refactoring, we progressively alter ownership to make sure (as much as reasonable) that objects owned by a technical module are accessed through that module. The goal is to limit cross-module accesses in the end. This is crucial to clean up dependencies and gain in the ability to freely evolve a module later on without impacting others in the development process.

INDUSTRIALIZATION OF THE DEVELOPMENT PROCESS

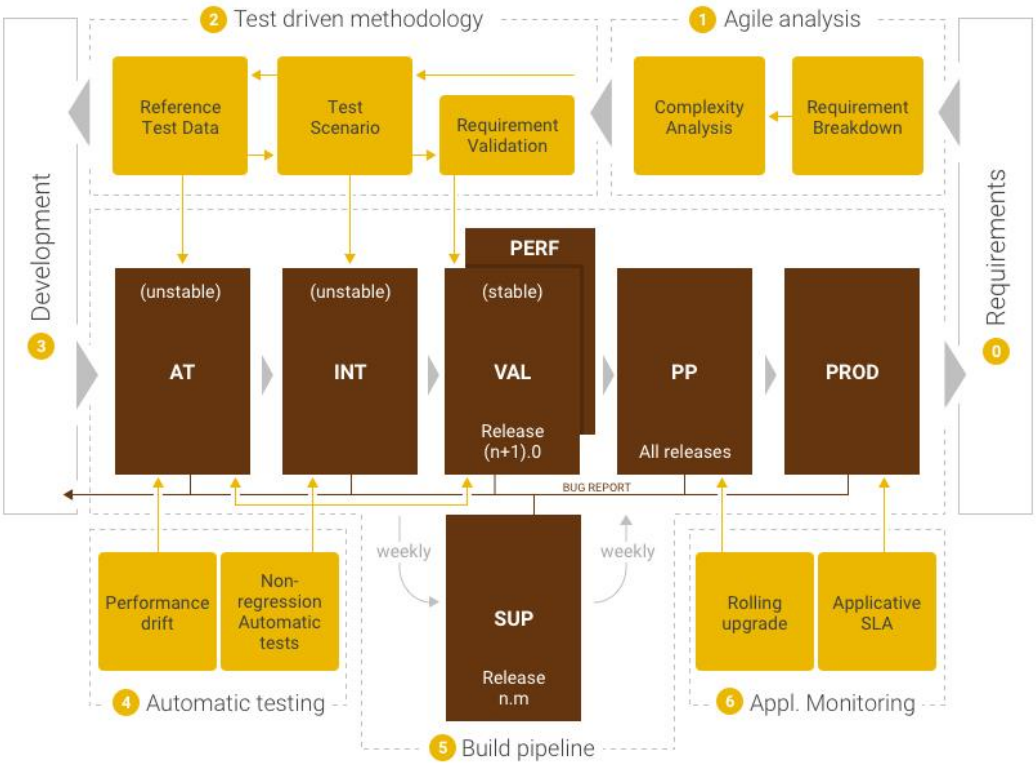
Originally, the testing strategy mostly consisted in conducting test campaigns prior to any release: a large set of manual tests was run by all team members during a week at least. This implies that tests were only performed before a release, but never before a patch deployment. This approach was obviously a major risk for the refactoring, as it prevented us from catching regressions early. Therefore we proposed to invest in automated testing, specifically to:

- Prevent regressions due to the refactoring;
- Ensure that the new architecture will be stronger against regressions in the future;
- Prove the improvement in terms of volume handling, and reduction in response times.

[6] https://hazelcast.com/

The new testing strategy is based on more unit tests and on UI automated testing to perform typical business scenarios, and call through it most relevant backend services in their real-life usage. However, automated UI tests are often seen as brittle when code changes. This is why we recommend an approach based on the *Page Object* pattern, and not on recording. The design principle of this pattern is to write a small, human readable API to manipulate the key features of the application. From these functions, it is then possible to build high level scenarios of testing by describing them naturally. Although recording seems to provide a quick-gain, the generated scenarios are typically mixing business and technicalities, making it hard to maintain and adapt in case of a change. It would therefore be cheaper immediately, but would incur a long term maintenance cost.

Another issue with front-to-back tests is that they usually take considerably more time to run than unit tests. So we proposed to introduce rationalized smoke tests to precede intensive regression tests. Concretely, this means we want to stage automated tests in the development process, and according to test results, push code versions further through the rest of the build pipeline. The pipeline, made up of a series of environments, aims at eliminating tedious and error-prone manual deployments across the different environments.



ELCA has a vast expertise on the provision and operation of such a *Software Factory*, with automated testing, configuration management and monitoring, be it on premise or in the cloud as a service. It provides a crucial competitive edge as it considerably reduces the amount of time between user demands and the rollout of corresponding features.

IMPACTS ON THE TEAM

The initial team was not organised to ensure the coherence of the system. The team grew from 4 developers to over 20 with no ownership for business modules. Anyone could decide to add a function, duplicate a piece of code, or access data in the shared databases if it was deemed convenient. Conway's popular adage proved to be very true; the code grew out of control.

> *"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"*
>
> -- Melvin Conway, 1968

To ensure that the refactored system stays healthy, the team was reorganized.

Following agile best practices, we recommended the introduction of smaller teams, each owning a number of business modules, and transverse roles, including:

- Product owner: manages the product backlog and ensures the functional coherence of the system.
- Architecture owner: Chooses technologies and ensures the technical coherence.
- Database manager: databases configuration, including their versioning, expansion/contraction for upgrades.
- Test manager: test automation and test campaigns.
- Release manager: responsible for the entire build chain and releases.
- Team leads: ensure the delivery and quality of the modules under their lead.

BENEFITS AND NEXT STEPS

The target backend architecture and organization present many benefits:

- Smaller components with well-defined responsibilities are easier to understand and maintain.
- Modules can be tested in insolation from the rest of the system.
- New features are delivered faster with fewer regressions.
- Business modules can evolve separately with independent releases. In particular, it is now possible to replace one component with a possibly better off-the-shelf solution in the future.
- Modules can be deployed on different nodes with different configurations, which makes it easier to scale out the entire solution.
- Morale and confidence of the team increase as they feel more productive.

Furthermore, the decoupling provides strong future-proof foundations for:

- Rapid innovation, with the ability to quickly change one part of the system.
- Reduced long-term commitment to a single technology stack, as each module can evolve to a new technology incrementally.
- Fast and distributed deployments, by embedding each module to its own container.

Such decoupling is therefore a major step forward to go distributed and fully enjoy the benefits of a web-scale microservice architecture.

ABOUT ELCA

ELCA is one of the biggest independent Swiss full-service providers for business and technology solutions, and a leader in the fields of IT business consulting, software development and maintenance, and IT systems integration. ELCA solutions reduce complexity and increase innovation cycles, improve business outcomes and customer satisfaction. The privately owned company, with more than 750 experts, has branches in Lausanne, Geneva, Berne, Zurich, Paris, Madrid and Ho Chi Minh City (offshore development), all operating according to a common process framework.

ELCA Informatique SA
Lausanne 021 613 21 11
Genève 022 307 15 11

ELCA Informatik AG
Zürich 044 456 32 11
Bern 031 556 63 11

www.elca.ch