0cx Documentation

Release 0.8

MaikelM

May 03, 2018

Contents

1 Abstract **1**

2 Status of This Memo **3**

3 License **5**

4 Language **7**

5 Content **9**
- 5.1 Goals . . . 9
- 5.2 Introduction . . . 9
- 5.3 No Complexity design principles . . . 10
- 5.4 Rationale and implications of the zero (0) complexity principles . . . 10
- 5.5 Contributors . . . 16

6 Indices and tables **17**

CHAPTER 1

Abstract

Business IT Complexity prevention is a real must for keeping your company agile and adaptive. Stop wasting time and money on complexity. Effective complexity prevention prevents hidden cost, improves quality and enforces an in-depth understanding of problems and customer needs. This 0CX (zero complexity) standard defines architecture and design principles to prevent complexity for new products. A high level of complexity often means serious risks and cost for systems. Especially for systems where humans, processes, software and new technology play a major role.

CHAPTER 2

Status of This Memo

This document specifies principles to prevent complexity when designing large scale business IT systems. Discussion and suggestions for improvements of this memo are welcome and encouraged. Just create an issue or pull request at https://github.com/nocomplexity/0complexity. The latest version is always published at https://readthedocs.org. Distribution of this memo in any form is encouraged, so please share the link https://0complexity.readthedocs.io/en/latest/ !

CHAPTER 3

License

CHAPTER 4

Language

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

CHAPTER 5

Content

5.1 Goals

This 0CX RFC is meant to provide a reusable set of principles for business IT software projects. It has the following specific goals:

- Offer a small but crucial list of principles to prevent complexity.
- Create a living document that outlines key business and IT knowledge available today to create large systems without negative complexity effects.
- Offer a short but crucial list of key principles that can be used in every business IT design project.
- Improve safety, security and privacy of systems, since dependency of humans on software technology is still continuously growing. We humans depend more and more on software driven products.
- Create a better world by preventing that our valuable time, money and resources are not wasted on solving complex business IT problems. We SHOULD use our time working on creating a better and healthier world for us all. So on problems that really matter and not on solving complex business IT problems that could have been prevented.
- Collect and maintain crucial universal but practical principles to avoid business IT complexity.
- All principles stated SHOULD provide input to steer developments within all types of business and for a wide range of IT architectures and design projects.
- The stated principles SHOULD be practicable usable over multiple disciplines and domains.

We know we aim high, but life is too short for wasting time on designing over engineered systems and we all SHOULD do better things than simplifying complex systems.

5.2 Introduction

Our ever-increasing complex world requires smart principles to prevent useless grow of business it complexity. Complexity problems are biased. This is because complexity has turned out to be very difficult to define. But despite a clear definition: Too much complexity is never good. So you SHOULD strive to avoid complexity from the start when initiating business IT projects for building new systems.

In basic business IT complexity does not have to be negative. We love complex technology as long as we have enough trust in the systems created. However we SHOULD be able to control all risks and be able to manage the created systems in an easy way.

In general to reduce cost and risks complexity MUST be avoided. Complex systems SHOULD be as simple as possible. Making changes on systems SHOULD not be a risk full and complex gamble. For simple systems maintenance cost are lower and quality is better predictable.

Most of the time you are not aware of the root cause that causes complexity in systems. When you are confronted with complexity you experience strong negative effects. IT complexity can have very serious effects. Besides high cost for maintenance and change, complexity can lead to severe risks that can impact security, privacy and safety for humans.

Nowadays any serious business is impossible without IT (Information Technology). So you SHOULD define general business IT design principles to prevent complexity in the systems you design. Since autonomous IT systems driven by machine learning software (like autonomous cars) SHOULD NOT become unneeded complex and high risk by default. We need solid architecture and design principles to mitigate risks when designing new systems.

This zero (0) complexity (0CX) business design principles RFC (Request for Comments) memo is meant to provide a reusable collection of principles to avoid and prevent complexity in new created large scale IT systems.

5.3 No Complexity design principles

This section gives an overview of principles that SHOULD be used when designing systems. If you are short on time and do not like reading: You MUST use the principles below to avoid adding complexity.

Summary of the zero complexity architecture and design principles:

1. Put People first!
2. Only use what you understand.
3. Define specific criteria that are tangible to measure complexity.
4. Create a model of your solution
5. Separation of concerns
6. Reduce all waste.
7. Problems should be fixed through simple solutions.
8. Design for change.
9. Make sure you can manage IT!
10. Privacy by design.
11. Never over engineer

In the next section, all principles are explained more in depth. It is RECOMMENDED to read the rationale for every principle stated.

5.4 Rationale and implications of the zero (0) complexity principles

This section outlines for every 0CX principle the rationale and outlines some implications. A good principle SHOULD cause some sort of pain when applying.

5.4.1 Put People first!

Statement : Put people first! Or never ever forget the importance of humans and the human factor.

Of course, we known that most systems SHOULD serve humans and also maintenance is done by humans. But looking at digital systems sometimes we wonder for what kind of humans the systems were created.

Rationale : Only humans experience the complexity. Humans are in charge for creating and maintaining your business IT landscape. Most humans like to be happy. Humans use technology. Explicit or implicit. Humans know what real value is. Human emotions and fears SHOULD NOT be neglected. E.g. are you willing to put your life in hand of a machine without any safety guarantee for your health? So whatever you are designing, never forget it's all about humans.

To prevent complexity in your product development you MUST put people before everything. If people are happy, healthy, feel safe and are really motivated to work together only then it is possible to work on solving complex problems without ending with complex products. People are more important than what they produce. Good positive people will learn. Make sure learning is allowed, so create a safe environment for learning.

Creating great software solutions will never make every person happy. But if no real harm is done, a little discomfort to change the world SHOULD be acceptable. You cannot make everyone happy. Creating simpler and better solutions means dealing with the human factor that comes with a change process, e.g. friction, discussions and emotions. So listen, be open, be good, and then doing it all over again. Iterate. Remember: All people matter.

Implications :

- Make sure all environmental factors are suited for people to work on good solutions. Organization culture SHOULD put people working on solutions above all. E.g. short or long time profit.
- Make sure that your business or system is not depended on the same small group of people.
- Put effort in a process for seamless transfers of activities from one resource to another if needed. What is easy for you can be hard and complex for others. How long does it take before some other new group can maintain your business and IT systems?
- Humans do have a very bad intuition for nonlinear systems. Unfortunately, the whole world and all complex systems are characterised by nonlinear dynamic behaviour.
- Most humans are very bad at predicting the future. So focus on a clear time scale that you can oversee. If your time scale is too far in the future, many unforeseen factors will impact the creation of your business or product. So keep complexity low by limiting the time scale.
- Safety first. Strive to avoid disasters rather than to attain an optimum for a simple solution. Lives of people matter.

5.4.2 Only use what you understand

Statement : Only use new technologies, methods or concepts when you fully understand the why, what and how of them.

Companies and humans will never create new systems with old ancient technologies. So we tend to have a weak for all kind of new fancy IT innovations. E.g. not many people have the skills to fully understand blockchain cryptographic mathematics, but this does not mean people often pretend to comprise what they are using.

Rationale : You SHOULD embrace and be open for new technologies and management methods. Only you MUST understand the short and long term risks. Beware that silver bullets for hard problems never come overnight. But IT technologies and new management methods do too often over promise and under deliver. Only result is too often an even more complex environment without the promised advantages.

So first focus on the problem you want to solve and not on so called tools that seems to provide a perfect solution. Don't follow the herd and the hypes. Given a hard complex problem, keep drilling until the problem is absolutely clear to you and then start working on the solution.

Implications :

- Experience, test on small scale what the new innovative method or technology really is. Do a small production pilot, with real customers. See if the new software technology is really x-times faster and cheaper for delivering functionality.
- Develop a structural way for introducing new innovative techniques in your organization.
- You MUST take risks, but you SHOULD have the capacity to handle large failures and disasters on this road. Only over several years you see effects of your risky decisions. Changes need time to have effect. You can not expect to see direct benefits from day zero. Often complexity will increase more when starting to resolve it. This since you can not stop your current complex business IT landscape overnight.

5.4.3 Create a model of your solution

Statement : To solve problems and to prevent complexity you MUST make a model before starting building your solution.

Rationale : Models are possible in many ways. But you SHOULD choose to create a model that not only you but also your stakeholders understand. Create a model that simulates all the effects that you think are important. E.g. create an architecture model, a security view, a privacy model, a threat model, a process model, a data model, an interaction model, a governance model, a cost model or any type of model that steers development in the desired direction.

Implications:

- Creating good dynamic models for complex problems is hard. Since complexity is never only about software code, you MUST create a model that gives insights in the elements that matter for preventing complexity.
- You SHOULD use good (FOSS) software to model your solution and run some typical use cases. Software simulation is easy, predictable and good models can be created very fast. See [ ] for a list of good FOSS simulation tools.
- Never ever fall in love with your model. No model is perfect. So there is only one way to know if your model works: Try it out for real! With software construction, this is easy: No expensive material cost are involved in creating software products. Just throw the software away if it didn't work out as expected. So take advantage of this great property of software products. This is also the reason that many complex electronic products are created in software first: It's far faster and cheaper to see if it works!

5.4.4 Define specific criteria that are tangible to measure complexity

Statement : Define specific criteria that are tangible, realistic and measurable to check if your system is not getting too complex.

Feedback based on facts is valuable. This accounts also for managing complexity within your systems. Make the complexity data hard. How many components are involved? How many interfaces are involved? How many concurrent users request will be served? Too often hard tangible criteria to evaluate the complexity of a system are missing. Too often all there is are conflicting opinions and perceptions.

Rationale : Since complexity SHOULD be defined differently depending on the specific context, you MUST define context specific rules to steer your design process in order to avoid complexity for your situation.

Implications :

- Define criteria like change cost, change time, repair time (MTTR), disaster recovery time needed, or number and competencies of resources required.

- Create a process to prevent complexity growth due to adding new user requests.
- A checklist for controlling complexity SHOULD be constructed using clear and simple explicit rules. Avoid opinion based and implicit arguments on your checklist whenever possible.

5.4.5 Separation of concerns

Statement : Separation of concerns (SoC).

Rationale : Problem solving is hard. Even harder is to solve problems in a simple way. Too often we end up with very complex solutions for a simple problem.

By using the separation of concern principle it is easier to understand how a system works and where improvements can be done. Separation of concerns means that sometimes you SHOULD strive to create autonomous SBB's (Solution Building Blocks). Call it working towards a micro-services style for your software building blocks. But the separation of concern is also applicable to business domains. E.g. do not mix external regulator processes with internal customer value processes. When applying SoC separation of work for all kind of agile work methodologies is possible.

Implications :

- Keep clear boundaries. Be very strict on the logical layers involved when solving a problem. Respect domains. This accounts for business domains and IT domains. Especially for software: you SHOULD NOT mix business logic with infrastructure logic. You SHOULD NOT expose business process logic in your IT components. Else easy reuse or creating changes becomes hard.
- Use loosely coupled APIs. Validate why the principles of the Twelve-Factor-App SHOULD NOT be applicable for you.
- Solve problems on the level where there problem is caused. So if your business process sucks, you can never add enough components and nice looking user interfaces to make users happy.
- Components can be replaced if better components are developed. So make sure that over time system building blocks will be replaced in your system.

Using this principle MUST BE done with care: From system science we know that we SHOULD NOT neglect the crucial system thinking principles for a system. So separating a system in subsystems (decomposition) can have drawbacks. This since systems are formed from their emergent properties, what means that the properties of the complete system is more than the properties of the separate parts.

5.4.6 Reduce all waste

Statement : Reduce all waste upfront in your design.

Rationale : Remove all activities, documents, software code and even people that do not deliver value. Call it zero waste. Spending too much time, money or resources on hypothetic discussions will not give you a working product faster. Start building. Go for execution. Start a real life-pilot as fast as possible. Do not design everything in front. But pay attention to on what functionality is minimally needed and define what is maximally needed for maintenance and changes. Kill the rest to prevent getting dead weight in your product and organization. To use a buzz term, call it a minimal viable product (MVP).

Overhead in organizations lead to Conway's law. The more people involved directly in a design phase, the harder it gets to deliver a simple system without all kind of strange concessions.

Especially for software design you SHOULD strive for KISS (Keep It Simple, Stupid). Simple is hard enough already. Another trap with too much stakeholders involved is the YAGNI trap. YAGNI means You aren't going to Need it. Implement only functionality that have a direct obvious purpose. Do not create functionality or integration logic for later. Later will not come. Requirements for later will almost never fit in your architecture without adjusting anyway. Creating functionality for later use will often lead to unneeded features and over engineered systems.

Of course you MUST make your architecture and design adaptive for change. But this SHOULD NOT mean to create functionality or abstractions that are not used directly.

Implications :

- Keep your organization and processes simple to avoid the risks of implementing needless complexity in software.
- Only spend time on creating functionality when it is clear who will use it.

5.4.7 Problems should be fixed through simple solutions

Statement : Problems SHOULD be fixed through simple solutions.

Rationale : Use simple solutions over complex solutions. Unfortunately designing a simple solution is much harder than creating a complex solution. Senior engineers and experts in a domain with years of experience have developed a troubled sight on what is simple and what is complex. To be clear: A simple solution is a solution where no configuration, maintenance or control is needed.

Elegantly simple designs don't happen by chance. They're the result of difficult decisions and discussion. Whether in the ideation, designing, or the testing phases of projects, all participant play a critical role in restraining the feature sets to reduce the complexity of the resulting product.

Simple solutions don't focus on features only, they focus also on clarity.

Be aware that all problems can be solved in multiple ways. There is never one a best solution. But there are bad solutions. So strive to find multiple simple solutions and choose the one that fits you. Good solutions needs to be so simple and transparent you forget how hard it was to get it simple.

For serious problem solving it is common to separate the problem in various domains. Things go wrong when you are creating a solution that you think is best for all domains involved.

Over engineering often occurs when for every sub process or sub function the perfect solutions is implemented. Perfect solutions however do not exist. Every solution fits on a typical use case and every solution has negative side effects.

Implications :

- Never stop directly when a solutions seems appropriate. Creating a simple solution takes far more time than a complex solution.
- Be open for other solutions that also match, but have other strings attached.
- Always evaluate if an available FOSS (Free and open-source software) solutions fits on your use case and within your context.

5.4.8 Design for change

Statement : Design for change.

Rationale : The only constant for every business is change. Dealing with change SHOULD not be delayed by IT. So whatever the business process is your IT systems must support: Make sure you can implement changes with minimal effort and minimal cost.

Implications :

- Keep your business process simple, so changes can be easily incorporated.
- Minimize the number of dependencies. This to minimize testing efforts and to minimize business risks when introducing new functionality.

- Make use of open standards supported by FOSS software implementations. This so you do not have an vendor lock in when you do want it.
- All basic tests should be automated when possible. Nowadays every serious business IT system must meet an enormous amount of explicit and implicit standards and requirements. E.g. for safety, security, privacy and of course error free is always nice to have. Implementing changes is not always hard. But validating that changes have no negative impact on other system part is. So to minimize business and human risks you SHOULD make a difference by automating testing whenever possible.

5.4.9 Make sure you can manage IT!

Statement : The management capacity MUST be able to manage the systems that MUST be controlled.

Rationale : Somewhere there seems to be a magical number of people, systems, application, or software (micro)services that can be managed. Management is control, change and maintenance of systems, software, people or even customers. Management can be done automated or by human activities. But it is obvious that when the number of objects that must be managed by one human is automation is the only way.

Implications:

- If management software, e.g. automatic CI/CD (Continuous Integration / Continuous Deployment) software is used, this software MUST be: Scalable, error-free and SHOULD be able to work on task in parallel.
- Enough resources MUST be available to prevent chaos.

5.4.10 Privacy by design

Statement : Privacy by design

Rationale : Do not store private data if it is not needed and remove private data when as soon as possible.

When using this principle you have less challenges to comply with legal regulations (local, global). E.g. to comply with the EU General Data Protection Regulation (GDPR). Limiting data collection prevents risks on data leakage.

Implications :

- Visibility and transparency and traceability of data in your systems.
- Respect for User Privacy.
- Data Minimization.Collection of personally identifiable information should be kept to a strict minimum.

5.4.11 Never over engineer

Statement : Never over engineer.

Rationale : We all like simple and easy solutions. However when decomposing a problem we often fall in the trap to solve all sub defined problems and then stop when we have a working system. However real simplicity is hard to create. It takes time, iterations and crucial feedback. The risk is we over engineer. The balance between working on a simple solution and over engineering is hard to find. In general when your system will not get any simpler stop engineering. Maybe later you see a way or opportunity for simplifying.

Over engineering often occurs during optimization. Optimization SHOULD always be considered harmful: In particular, optimization often introduces complexity, and as well as introducing tighter coupling between components, layers and a tight coupling between business processes and IT systems. So stop engineering when it works. But never stop making it simpler, but mind the trap of optimization.

Implications :

- Prototype before polishing. First get it working, then simply, simplify and simplify.

5.5 Contributors

The following people have contributed to this RFC document of principles:

[name] [OPTIONAL email] [Optional Organisation name]

If you like your name stated here: This book is open source. Issues and pull requests are welcome. All contributors will be added to this list.

So Get involved in the discussion to make it better!

If you wish to make comments regarding this document, please raise them as GitHub issues. Only send comments by email if you are unable to raise issues on GitHub. All comments are welcome!

CHAPTER 6

Indices and tables

- genindex
- modindex
- search