

Whitepaper  
Cloud Transformation  
and IT Modernization

CGI



# Application Complexity Demystified

# Abstract

Those in charge of large applications often can't escape the feeling that they are far too complex to easily manage. Among the telltale symptoms: it's difficult and/or costs a bundle to make changes, and it seems to take forever to get new users up to speed. Sound familiar? There are lots of legacy systems out there that fit this description: they've reached the point where it has become difficult or impossible to make required changes and tweaks quickly enough. Excessively complex applications inevitably suffer from poor performance and flexibility, but there is no established definition of this syndrome and very little agreement in the literature on what "software complexity" actually is.

This paper fills the gap. It introduces a methodology for defining, quantifying and measuring application complexity. In it, we discuss various symptoms of complexity and show how they interfere with cloud transformation and IT modernization. We also explain how complexity shifts when migrating a legacy application to a microservice architecture.



# Introduction

Lehman's Second Law of Software Evolution (Lehmann, 1997) states that the complexity of applications increases over time:

*"Increasing complexity – a software system will become progressively more complex over time, unless explicit work is done to reduce its complexity."*

Increasing complexity leads to:

- More work to explain how a system works and teach new staff to use it
- Greater development and maintenance costs
- Loss of flexibility and speed when implementing new functionality

We therefore see a considerable need to offset the tendency for applications to increase in complexity. When the perceived complexity of a monolithic<sup>1</sup> application or set of applications threatens to reach the breaking point, the way out of this trap can be to implement a cloud transformation and IT modernization program for converting them into microservices. Typically, the first step is to analyze the application portfolio. The results should include some quantifiable statements about the level of complexity of the overall portfolio and of each individual application. The term "complexity" is not well-defined in an application context. Individual observers use the term very subjectively to mean quite different things:

| Stakeholder             | Subjective perception of complexity  |
|-------------------------|--|
| Head software developer | Number of applications<br>Number of interfaces<br>Number of development and or product lines                               |
| Software developer      | Number of frameworks and libraries<br>Number of dependencies between code elements<br>Lines of code                        |
| CIO                     | Number of services<br>Number of vendors relationships / suppliers<br>Degree of standardization and IT harmonization        |
| Enterprise architect    | Number of applications per capability or business domain<br>Number of platforms<br>Number of interfaces, APIs and services |
| Head of operations      | Number of platforms<br>Number of dependencies (of components in terms of ITIL)<br>Number of configuration items (CMDB)     |
| Line manager            | Number of products and product configurations<br>Number of process varieties   |

<sup>1</sup> monolithic application or software system is one that has to be deployed in its entirety, see also: <https://microservices.io/patterns/monolithic.html>.

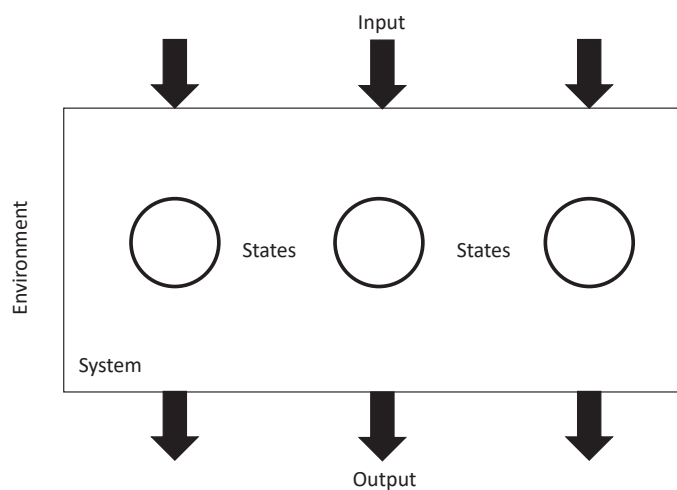
# System Perspective

To arrive at a common understanding of complexity, let's start by explaining some of the terminology used in systems theory. Then we'll develop a more precise definition of application complexity.

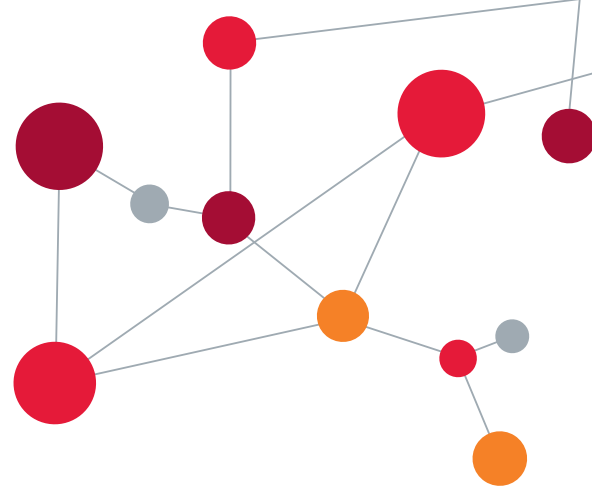
According to (ISO/IEC 15288:2008, n.d.), a system is a combination of interacting elements organized to achieve one or more stated purposes.

In other words, a system is a set of (possibly heterogeneous) objects along with the relationships among these. According to Ropohl (Ropohl, 2009), there are three main ways to view a system:

- 1 Functionally, focusing on how the system behaves. It is modeled as a black box, meaning that we are ignorant of its inner workings. We can know (or observe) a system's input and output parameters and possible states, but we don't know how it actually works or converts inputs into outputs.



*Figure 1: Functional concept of a system*



- 2 Structurally, focusing on a system's elements and the relationships among them. From this perspective, different combinations and correlations between elements produce a variety of system properties.

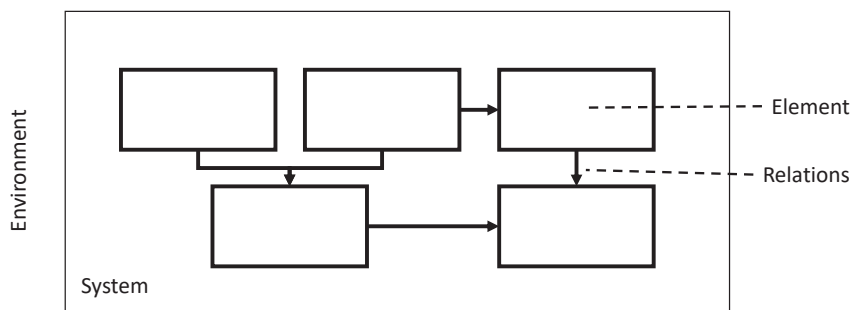


Figure 2: Structural concept of a system

- 3 Hierarchically, organizing a system into levels with supersystems and subsystems, all of which are systems in their own right.

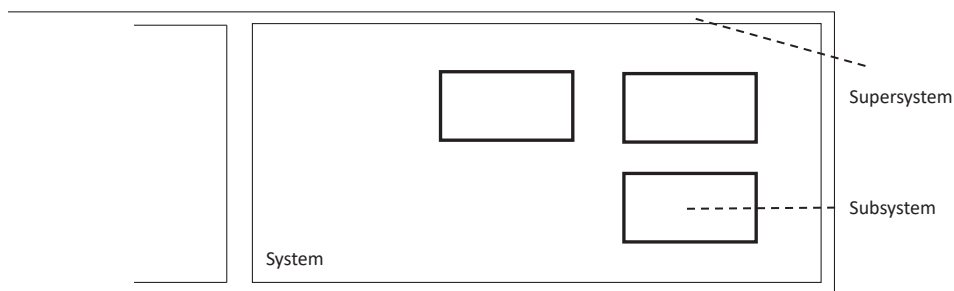


Figure 3: Hierarchical concept of a system

For our purposes, the second and third approaches are most important. An application is a software system. If we look at a portfolio or set of applications as a system, every application is a subsystem with an architecture corresponding to the system's overall structure.

# Complexity

Now we're ready to take the first step in defining complexity. We could distinguish between structural and behavioral complexity, but for our purposes only structural complexity needs to be considered.

Definitions:

A system's structural complexity is determined by the number of elements it contains, the number of relationships among these elements, and the degree of heterogeneity of the elements and their relationships.

A system's behavioral complexity is the degree of uncertainty with which its behavior can be predicted.

The simplest way to measure structural complexity is by counting the system's elements and their relationships on different hierarchical levels. In order to do this, we need a way to tell which elements we should count and which not. The C4 model for visualizing software architecture is ideal for this purpose.

# C4 + 1 Model

The C4 model developed by Simon Brown (Brown, n.d.) provides a static view of a software system, which can also be extended to a collection of software systems (as a system landscape). It gives us a tangible interpretation of the structure of a software system, defined in terms of containers and components.

In the C4 model, a software system is defined as a hierarchy of simple building blocks as follows: “A software system is made up of one or more containers, each of which contains one or more components, which in turn are implemented by one or more classes. A software system is the highest level of abstraction and represents something that delivers value to the users, whether they are humans or not. A container<sup>2</sup> is something that hosts code or data, like an application or database. A container is something that needs to be running in order for the overall software system to work. A component is something grouping functionality implemented using one or more classes (or just code).”

The following diagram shows an excerpt from the meta-model:

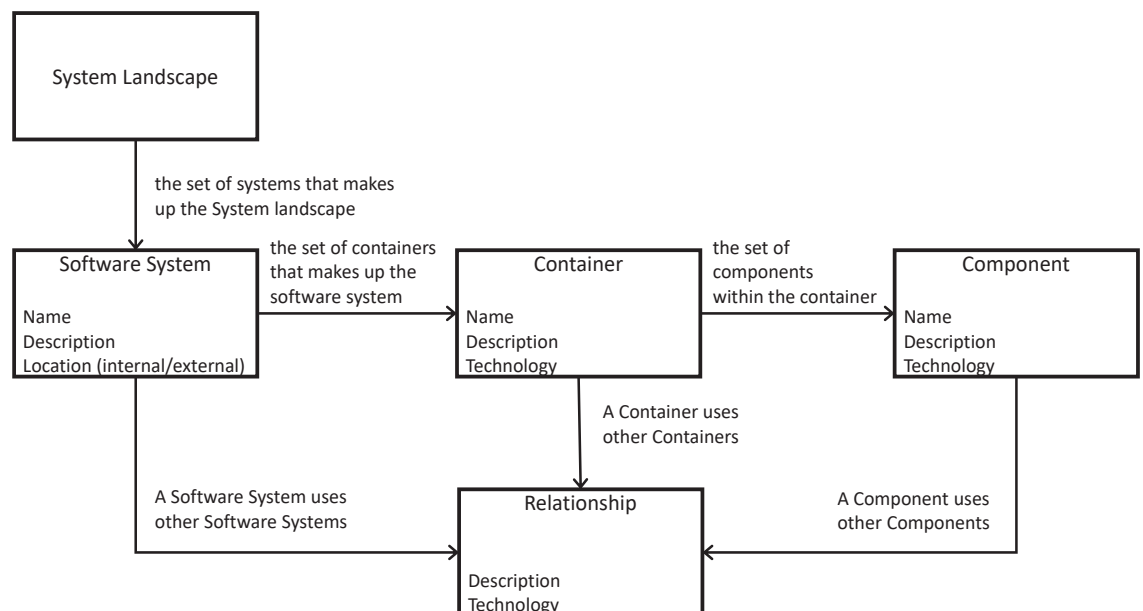


Figure 4 - Excerpt from the C4 meta model

<sup>2</sup> In the C4 model, a container is neither a Docker container nor a J2EE container. It is actually an application running inside containers of those kinds. Here we're not interested in infrastructure. Good examples of containers include server-side web applications, client-side desktop applications, mobile applications, and databases. Every container can be deployed onto a separate piece of infrastructure that can be physical, containerized, or virtual.





A good example is a description of a monolithic software system before and after it is migrated to a microservices architecture.

- 1 The monolithic software system is modeled as a single software system. It usually consists of a bunch of containers, each of which holds a lot of components. This perceived complexity makes it hard to understand the system as a whole and make changes to it.
- 2 Instead, we model each microservice as an independent software system with one or two containers and just a few components. Now we have a system of systems. Each individual microservice is fairly straightforward and easy to understand, even if the system landscape as a whole is much more complex.

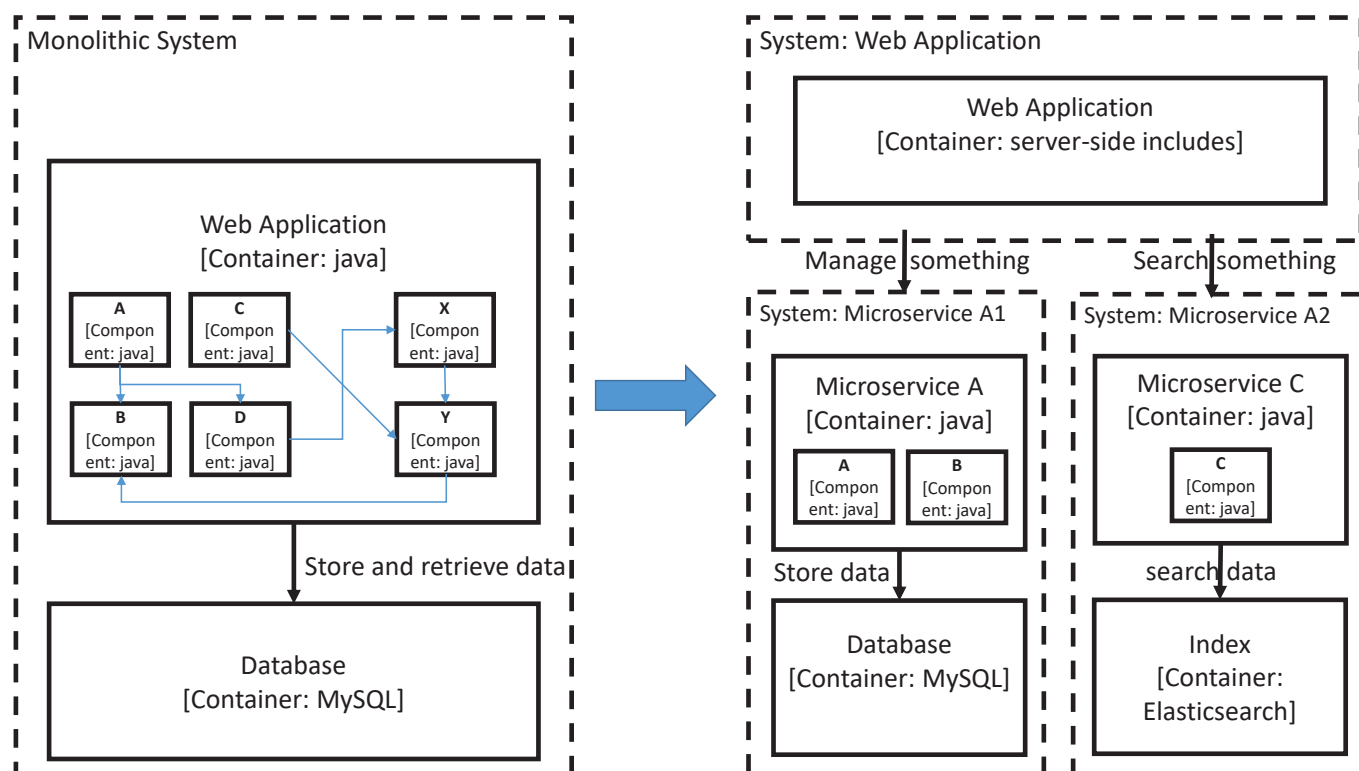


Figure 5 Complexity on different levels

This figure illustrates the benefits of defining structural complexity at different levels.



# Structural Complexity

Using the C4 model, we can define a metric for quantifying the structural complexity of


1 a software system by counting:

- the number of containers plus the numbers of components in them,
- the number of relationships between the containers plus the number of relationships between them, and
- the heterogeneity of components and their relationships; and

2 the system landscape by counting:

- the number of software systems,
- the number of relationships between the software systems, and
- the heterogeneity of the software systems and their relationships.

These techniques compensate for the fact that we humans aren't very good at multitasking, i.e. managing many different things at the same time. This is also why cognitive<sup>3</sup> complexity is a function of structural complexity.



*Systems get harder to understand with increasing complexity.*

<sup>3</sup> See also Appendix B on cognitive complexity.

# The Complexity Cross

In their article “The Complexity Cross”, Ephraim R. McLean and Scott L. Schneberger (McLean & Schneberger, 2003) examined the structural complexity of information systems. After identifying a relationship between the complexity of an entire system and the simplicity of its elements<sup>4</sup>, they established that transforming a centralized mainframe system into a distributed client-server system doesn’t always reduce complexity. (This result is analogous to the example above.) They call this effect the “Complexity Cross”. Their insights have been very helpful for our own investigation.

They came up with a very similar definition of an information system’s complexity:

*“Computing complexity can be characterized by the number of and variety of components and their interactions.”*

They then added the rate of change<sup>5</sup> in time and space to derive a less static approach.

*“The faster the computing system changes, the harder it is for the system to adjust and for users to understand the new system.”*

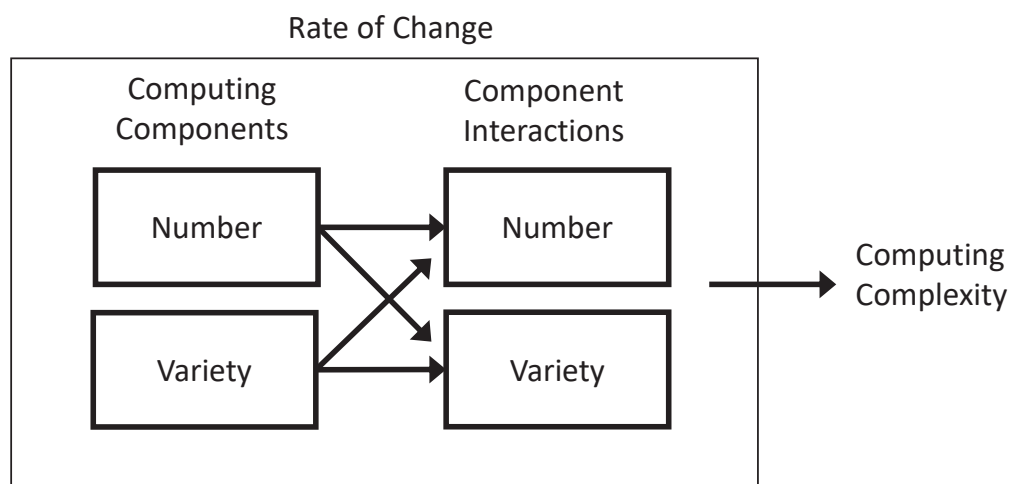


Figure 6 – Complexity factors

<sup>4</sup> They call the elements in their work “components” and refer to structural complexity as “computing complexity”.

<sup>5</sup> See Appendix A.

They derived three main conclusions:

- 1 The difficulty of software maintenance is directly related to the complexity of the software system itself.
- 2 Some complexity factors appear to have a greater impact than others:
  - a The heterogeneity of components and interactions plays a greater role than the number of components and their interactions.
  - b The variety of components is more important than the variety of interactions.
  - c The rate of system change has a greater effect than either the variety of components or the variety of interactions.
- 3 The Complexity Cross: as the complexity of the components decreases, the complexity of the overall system accelerates.

Conclusion (3) reflects the shift from mainframe to distributed systems. A mainframe is a very simple system with complex components, and a distributed system is assumed to be a complex system with simple components.

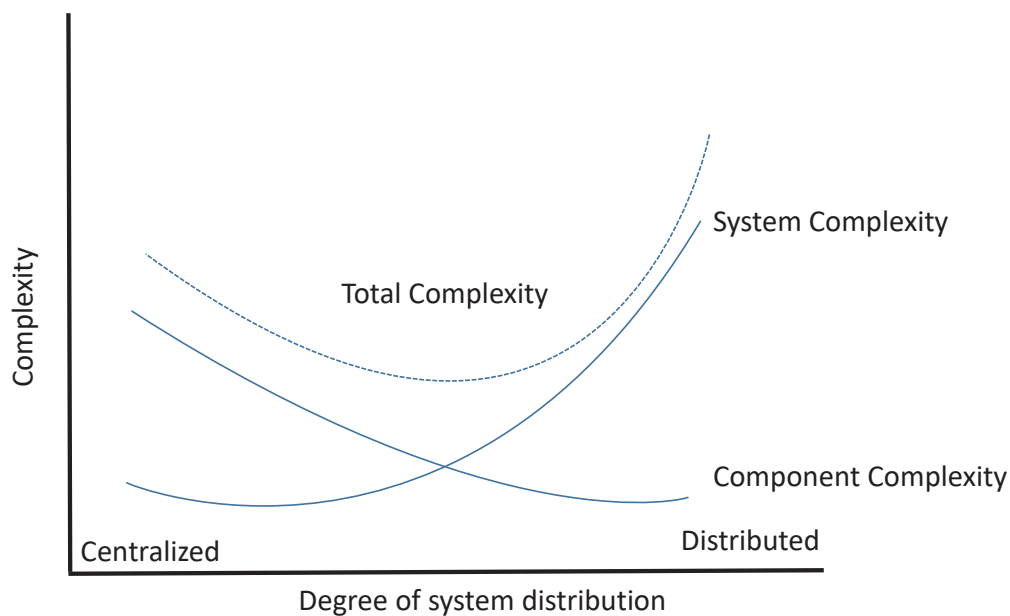


Figure 7: The Complexity Cross

One fundamental question now is whether these insights will continue to hold true as distributed systems based on client-server technology evolve into distributed systems based on cloud-native technology. Research on the relationship between software modularity and costs suggests that they will:

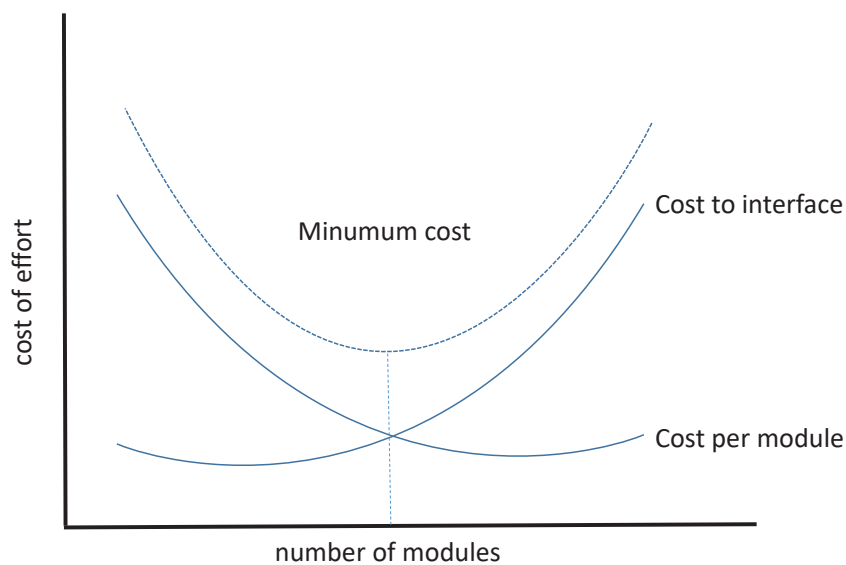


Figure 8: Modularize at the minimum cost point

*“The logic behind breaking a large program into smaller pieces is to ‘divide & conquer’, as a complex problem is easier to solve when broken into simpler pieces. But the greater the number of modules, the greater the number of interfaces among them. Eventually, the overall cost of dealing with a large number of modules overpowers the savings from working with smaller program pieces. The trick is to modularize at the minimum cost point.”*

The Complexity Cross implies that, in order to reduce overall system complexity, it’s necessary to reduce the complexity of both the system as a whole and its individual parts by:

- lowering the component complexity curve by simplifying individual components;
- lowering the system complexity curve by reducing the variability of components, standardizing components, and reducing the interdependencies between components; and
- shifting the intersection of these curves by finding a good compromise between centralization and distribution.

# Heterogeneity

According to the second conclusion drawn by McLean and Schneberger, heterogeneity is a much stronger predictor of a system's structural complexity than the numbers of elements and relationships it contains. It is a statistical value reflecting how widely the attributes of a system's elements vary. It can be measured by creating a diversity index (borrowing from biology and information theory) for the technological characteristics of components and their relationships in the C4 model:

Heterogeneity increases with the number of different technologies (e.g. vendors, programming languages etc.) that are used at the component level.

Heterogeneity is inversely proportional to the disparity between different technologies (used at the component level).

To shed light on various aspects of heterogeneity, Alexander Schütz (Schütz, 2016) chose three different indices.

- 1 Shannon entropy<sup>6</sup> EM: for a population of  $N$  elements belonging to  $S$  different

species, with  $n_i$  being the number of elements of species  $i$  and  $p_i = \frac{n_i}{N}$  their proportional relation, Shannon entropy is defined as:

$$EM = -\sum_{i=1}^S p_i \cdot \ln(p_i)$$

- 2 Richness  $n$ : this tells us how many different technologies are used in a software system (or in biology, the number of species in an environment, i.e. its biodiversity).
- 3 Evenness  $E$ : this tells us how evenly distributed technologies are across the elements of a software system.

$$E(N) := \frac{EM}{\ln N}$$

<sup>6</sup> In information theory, Shannon entropy is the average rate at which information is produced by a stochastic data source.

# A measure of complexity in a system landscape

Based on the previous discussion of heterogeneity, we can extend our measure of structural complexity:

- 1 to a software system by defining complexity  $C_T$  (of a system's parts) and complexity  $C_R$  (of the relationships among them):

$$C_T := (N_T, (EM_T, n_T, E_T))$$

$$C_R := (N_R, (EM_R, n_R, E_R))$$

where

- $N_T$  is the number of containers plus the number of components,
- $N_R$  is the number of relationships between containers plus the number of relationships among components,
- $EM_T, n_T, E_T$  are the diversity indices (Shannon entropy, richness and evenness) of the components, and
- $EM_R, n_R, E_R$  are the diversity indices (Shannon entropy, richness and evenness) of the relationships among the components,

- 2 and to a system landscape by measuring complexity  $\dot{C}_T$  influenced by the software systems and complexity  $\dot{C}_R$  influenced by the relationships among the software systems:

$$\dot{C}_T := (\dot{N}_T, (\dot{EM}_T, \dot{n}_T, \dot{E}_T))$$

$$\dot{C}_R := (\dot{N}_R, (\dot{EM}_R, \dot{n}_R, \dot{E}_R))$$

where

- $\dot{N}_T$  is the number of software systems,
- $\dot{N}_R$  is the number of relationships among the software systems,
- $\dot{EM}_T, \dot{n}_T, \dot{E}_T$  are the diversity indices (Shannon entropy, richness and evenness) of the components of the software systems, and
- $\dot{EM}_R, \dot{n}_R, \dot{E}_R$  are the diversity indices (Shannon entropy, richness and evenness) of the relationships among the components of the software systems.

These measures allow us to quantify complexity simply by counting the elements of the systems. In combination with heterogeneity, we thus obtain a much better metric for understanding complexity.

<sup>7</sup> We are using the same notation as in (Schütz, 2016).

# Conclusions

This paper introduces a methodology for defining, quantifying, and measuring the complexity of applications. In it, we discuss three of the four determining factors of structural complexity: the number of elements, their relationships, and the heterogeneity of the elements and their relationships. We have assumed that heterogeneity has a greater impact on complexity than the numbers of elements and relationships. We have defined ways to measure complexity by simply counting elements and relationships, and applied the well-known principle of Shannon entropy to measure heterogeneity. Where applications are concerned, we use an extended form of the C4 model of software architecture to measure the complexity of software systems and system landscapes.

We have not investigated the fourth factor, namely the rate of change. However, the cited book by Robert Martin includes principles<sup>8</sup> for designing systems that are made to change and metrics for determining how ready a software architecture is for change.

We have discussed complexity in the context of cloud transformation and IT modernization. In essence, we explain how complexity shifts when migrating a legacy application to a microservice architecture: a monolithic software system is transformed into a system landscape comprising multiple simple software systems based on the microservices architecture. In connection with converting a monolith to a microservices architecture, we have derived insights from the Complexity Cross and concluded that there may be an optimal degree of system distribution. It is very important to find the right balance between segregating and aggregating functionality.

In addition, looking at heterogeneity it is apparent that, in the context of a microservices architecture, using new technology for everything has no impact at the system level<sup>9</sup> but has a strong impact at the system landscape level.

We can now apply these insights to develop an application transformation methodology that will help us create an application landscape that is less complex, more flexible, and more efficient.

<sup>8</sup> See Appendix A.

<sup>9</sup> This is why we consider a microservice to be a system in itself.





## References

- Braun, S. (n.d.). The C4 model for software architecture. Retrieved from <https://c4model.com>
- ISO/IEC 15288:2008. (n.d.). Retrieved from Systems and software engineering — System life cycle processes: <https://www.iso.org/obp/ui/#iso:std:iso-iec:15288:ed-2:v1:en>
- Lehmann, M. (1997). Laws of Software Evolution Revisted. Retrieved from <https://www.rose-hulman.edu/class/cs/csse575/Resources/Lehman-LawsRevisited-96.pdf>.
- McLean, E. R., & Schneberger, S. L. (2003). The Complexity Cross – Implications for Practice. Communications of the ACM.
- Ropohl, G. (2009). Allgemeine Technologie: eine Systemtheorie der Technik. Retrieved from <https://books.openedition.org/ksp/pdf/3003>
- Schütz, A. (2016). Komplexität von IT-Architekturen. Springer Gabler Research.

## Appendix A

### Structural complexity as a measure of change.

Dependencies strongly impact change. If you want to change something that many other things depend on, you have to understand all of those things and probably also test and change them as well.

Robert Martin posited additional principles (beside SOLID) for designing packages: his cohesion and coupling principles. Based on CBO<sup>12</sup>, he distinguishes between afferent and efferent couplings for measuring the stability (in position) of a package by counting the dependencies that enter and leave the package:

- Afferent couplings (CA): the number of classes outside this package that depend on classes inside it.
- Efferent couplings (CE): the number of classes inside this package that depend on classes outside it.
- Instability (I):

$$I = CE / (CA + CE).$$

I=0 indicates a package with maximum stability. A package is stable if it is hard to change: in other words, if there are many more afferent couplings than efferent couplings. As a result of all the dependencies on other packages, you can't change anything without changing all the dependent packages too.

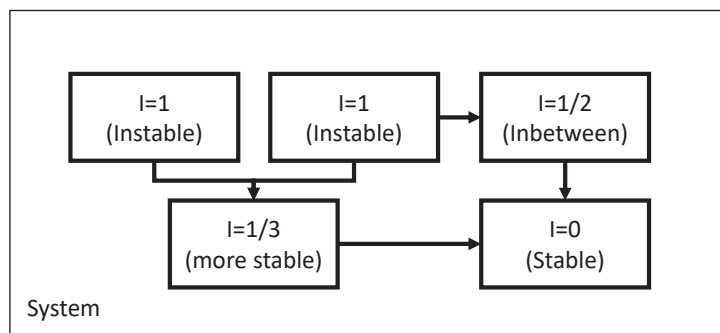


Figure 9: Stability

<sup>10</sup> Robert Martin: Agile Software Development. Principles, Patterns, and Practices, Prentice Hall (2002)

<sup>11</sup> CBO means "Coupling between object classes". This metric was introduced in 1994 by Chidamber and Kemerer in their paper "A metrics suite for object oriented design" (see <https://ieeexplore.ieee.org/abstract/document/295895>).

If all of the packages in the entire system have maximum stability, the system is impossible to change! Code that needs to be changed very often should therefore be put into unstable packages. A package's stability increases with its level of abstractness:

- NC: number of classes in the package
- NA: number of abstract classes in the package
- A: abstractness:

$$A = NA / NC.$$

A value of  $A = 1$  means that a package only contains abstract classes, and  $A = 0$  means that there are no abstract classes in it.

Now we can compute the abstractness and instability of each package and plot these two metrics in a diagram with x and y axes:

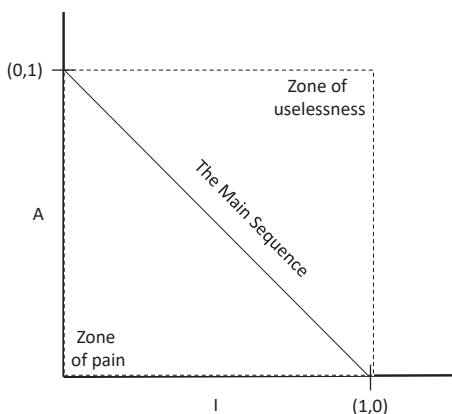


Figure 10: Robert Martin: Agile Software Development

The main sequence is ideal. This leads to the last metric. We can compute the distance  $D$  from the main sequence as follows:

$$D = |A + I - 1| / \text{SQRT}(2)$$

$$\text{or the normalized distance: } D' = |A + I - 1|.$$

Packages with a  $D$  close to zero are easier to maintain and less sensitive to change. Applications with a structure consisting of packages in which  $D$  is close to zero are therefore less complex.

This leads to the following basic principles formulated by Robert Martin:

*Acyclic Dependencies Principle (ADP): The dependency graph of packages must have no cycles.*

*Stable Dependencies Principle (SDP): The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.*

*Stable Abstractions Principle (SAP): Abstractness increases with stability.*

## Appendix B

### Cognitive complexity as a measure for understanding complex structures.

Cognitive complexity measures the effort a human being must make in order to understand a complex structure. If we compare two software systems A and B, then A is less complex than B if A is easier to understand than B.

From cognitive psychology<sup>12</sup> we are familiar with three cognitive processes that human beings use to understand complex structures: chunking, formation of hierarchies, and schemata:

*“... the realization that people respond to complex structures by chunking, creating hierarchies, and building schemata. A software system whose architecture supports these processes has lower architectural complexity.”*

A software system that supports these processes will by nature have lower cognitive complexity, because it is structured and built in a way that humans can understand with less effort.

In the context of cognition, structural complexity is mapped onto modularity (chunking), pattern conformity, and ordering.

These three criteria apply to the Software Architecture Complexity Model (SACM)<sup>13</sup>.

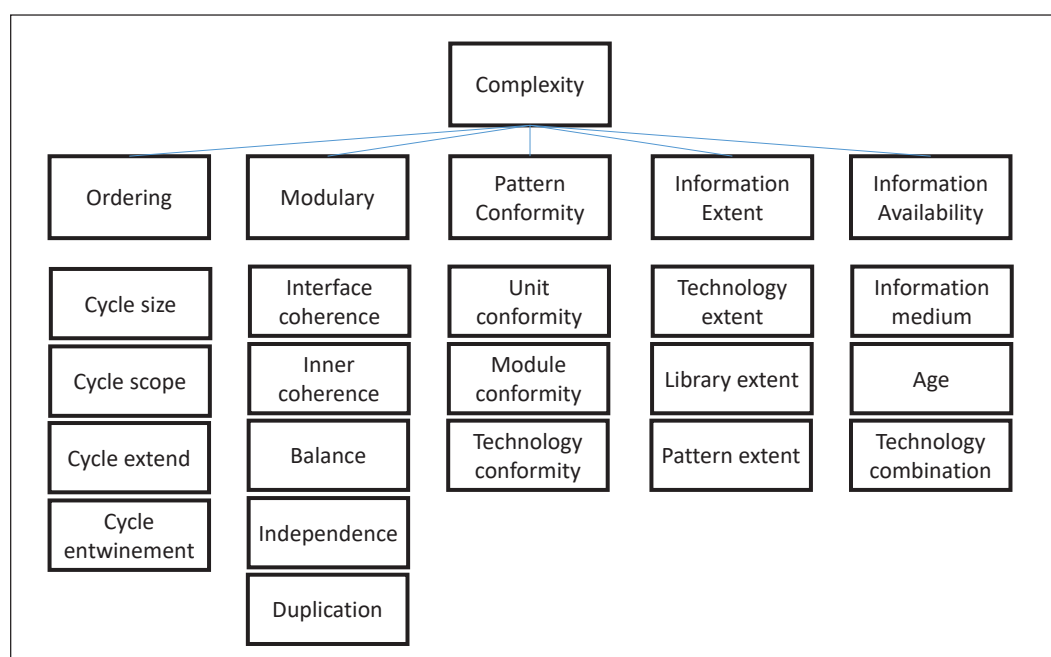


Figure 11: Overview of SACM from “A Cognitive Model for Software Architecture Complexity”

<sup>12</sup> Carola Lilienthal, Komplexität von Softwarearchitekturen, 2008

<sup>13</sup> E. Bouwers, C. Lilienthal & A. van Deursen, A Cognitive Model for Software Architecture Complexity, 2010

And in that model the criteria were extended to huge software systems, never can be understood by a single person because of its enormous complexity. In these cases, several people have to work together to achieve some kind of distributed cognition. The term “information” refers to every piece of meaningful data that is needed to understand a software system. In the case of “distributed cognition”, information has to be exchanged, and communication and team processes must be efficient. “Informational extent” refers to the total amount of information that needs to be understood, and “information availability” refers to the same information. The assumption is that heterogeneity increases with the information extent.

## **Appendix C**

### **Tools for quantifying complexity and analyzing qualitative aspects of complexity.**

Tools are available in the market for calculating and visualizing an application's complexity:

- JDepend (<http://mcs.une.edu.au/doc/jdepend/docs/JDepend.html>)
- SONARGRAPH (<https://www.hello2morrow.com/products/sonargraph/>)
- SOTOGRAPH (<https://www.hello2morrow.com/products/sotograph>)
- Dependometer (<https://www.heise.de/developer/artikel/Dependometer-228004.html>)
- SonarJ (SonarJ has been renamed Sonargraph)
- SonarQube (<https://www.sonarqube.org>)
- Structurizr (<https://structurizr.com>)
- Structure101 (<https://structure101.com>)
- STAN (<http://stan4j.com>)



**CGI Deutschland B.V. & Co. KG**

Leinfelder Straße 60

70771 Leinfelden-Echterdingen

[www.de.cgi.com](http://www.de.cgi.com)