Constructal Law & Second Law Conference 2015 CLC 2015, Parma (Italy), 18 - 19 May

Applying the constructal law to restrict complexity growth in software architectures

Alexandre A. de Oliveira

146 Pinheiro Machado street, Pedro Osório, RS, 96360-000, Brazil, chavedomundo@gmail.com

Abstract

Techniques employed to restrict complexity in software architectures usually rely on empirical knowledge. The most known and used design solutions in the industry are not supported by scientific evidence, making their replication in different applications a challenge, often times resulting in hard to fix defects due to unpredictable disorder. In this context, the Constructal Law becomes a prominent point of reference for new studies in the area of software design. By deliberately applying the concepts of the Constructal Law in the design of messages between components and the position of such components, developers are able to engineer simpler distributed systems and internal functions in a natural way, keeping the growth of messaging complexity, previously exponential, under an acceptable threshold, proportional to the growth of the system itself. By relying on the Constructal Law, engineers are able to replicate the same proven design in different software environments, resulting in lower, predictable costs for building new features and maintaining big systems.

Keywords: constructal law, software design, systems architecture, tree-like design.

1. Introduction

From websites to hospital systems, from controlled rockets and weather simulators to stock markets and the vehicles we drive, software is at the core of modern civilization.

Yet, the results evidence that not enough effort is being put into engineering better architectures to generate flexible, easy to modify systems that would enable businesses and organizations to evolve in an agile manner, therefore adapting to the ever changing environment. Newer systems are not solving the inflexibility problem of old systems (Fauscette, 2013).

In that scenario, the *Constructal Law* has an important role in the development of ideas to remove the blockers that prevent organizations from changing their software systems. Bejan & Zane (2012) define the Constructal Law as follows:

“for a finite-size flow system to persist in time (to live), its configuration must evolve in such a way that provides easier access to the currents that flow through it”. This governs both the animate and inanimate.

Change itself is the evolution of one form into another, therefore change is itself a process that flows in time. By studying the constructal law and deliberately designing software in particular ways, as will be explained further in this text, engineers are able to generate simpler systems that are easier to change. Furthermore, the Constructal Law provides us the tools to predict the future of software architectures, therefore allowing us to formalize new concepts ahead of time.

Before demonstrating how the Constructal Law helps engineers design less complex systems, we’ll analyze what are the efforts required to modify software systems. Let’s start by introducing some of the reasons why changing systems is so costly.

2. The cost of changes

In software, a component is a unit that executes some specified logic to achieve a particular goal (Heineman & Councill, 2013). The way components are laid out will impact on how complex a system is. When a modification has to be undertaken, it takes place in a component or in group of them.

2.1 Possible configurations

Fakkink (2013) states that the number of possible configurations in a system grows exponentially in respect to the number of nodes. On figure 1, we demonstrate this behavior by showing how components’ connections could evolve as a particular system grows.

Fig. 1. Systems of components with different sizes.

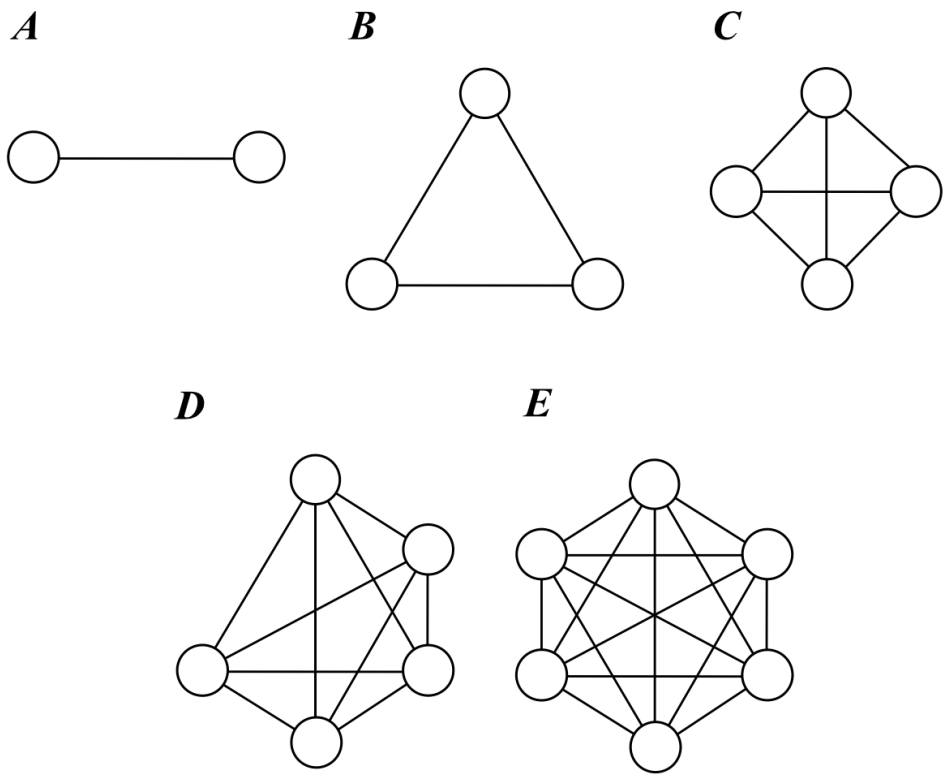


Table 1 compares the different configurations.

Table 1. Complexity evolution.

Configuration *X*	Total components (*T*)	Possible configurations (*k*)
A	2	1
B	3	3
C	4	6
D	5	10
E	6	15

Complexity tends to grow exponentially. The lack of adequate efforts to keep a system under an acceptable level of complexity will yield softwares that will, ultimately, become impossible to be modified without unwanted side-effects, such

as the introduction of bugs and inconsistencies.

2.2 Component dependencies

When a component *A* requires a component *B* in order to work, it means that *B is a dependency of A* (Martin, 1994). During the lifetime of the software, whenever component *B* has to change, component *A* is forced to be changed altogether. Therefore, for any change *m*, the total changes (*t*) required will be

$$t = m + d \tag{1}$$

where *d* is the total number of components that depend on *m*.

With this, the natural conclusion is that the more dependencies a software system's components have, the more likely it is that changing one component will consume additional time.

Given that it's possible to mitigate this problem by designing the system architecture in specific ways, as will be shown further in this paper, the additional time required to make changes in a software component should be seen as *lost productivity*. As a result, environments that depend on fast paced changes are blocked by systems that are not flexible nor easily adaptable.

That can be verified in the Enterprise Resource Planning (ERP) market, where Fauscette (2013) did an important discovery: meeting changing business requirements involves moderate to substantial underlying ERP system changes. The following was also discovered:

- The cost of system modifications is direct and includes a substantial amount of lost productivity.
- 15% of the surveys companies were forced to reimplement the entire ERP system. Of those, 60% had initially spent over $2.5 million.
- The frequency of change and the subsequent investment to make the ERP system support the business have become so common that many businesses simply accept that there is no better way to operate the business.

This lost productivity takes place because of fragile architectures. Martin (1994) defines fragility as the "tendency of a program to break in many places when a single change is made". Such fragility greatly decreases the ability of a company to make strategic moves that depend on information technologies.

These facts acknowledge that businesses, unaware of better ways of designing software architecture, are losing money and time to market due to ineffective engineering solutions.

3. Design of components

Bejan and Zane (2012) state that given freedom, flow systems will generate better and better configurations to flow more easily. Observing nature, where freedom is abundant, the *tree-like shape* is easily recognized as a very prominent pattern. It's present in the air passage of lungs, the capillaries and in the dendrites of neurons in our brains. It's present in rivers, lightnings and snowflakes. That it the shape nature uses to reduce resistance to flows.

To exemplify how that applies to software, we will look into three examples of architectures, the first two presenting unwanted side-effects during the modification process, albeit commonly used in practice, and the third one with a tree-like shape, applying our acquired knowledge from the Constructal Law.

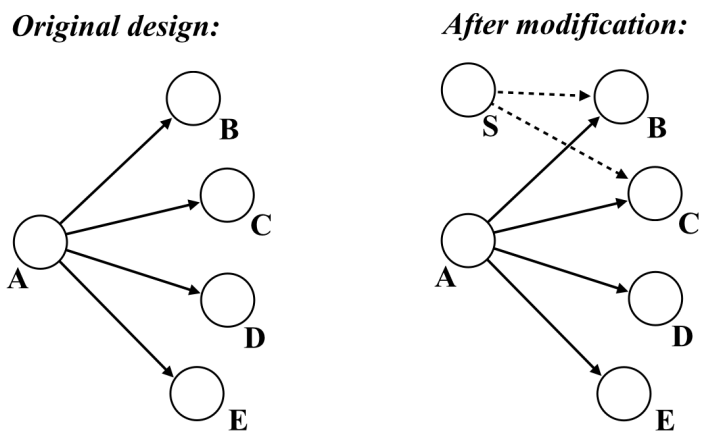
3.1 Star-like design

In the star-like design, a component *A* depends directly on other components without the use of well-defined modularity (Parnas, 1972), interfaces (Martin, 1994) or mechanisms to decrease complexity.

Figure 2 shows the flow of change where components are represented by circles. Before the modification, the system has 5 components and 4 dependencies (represented by lines between components). Component *A* depends on components *B, C, D* and *E*, which are executed in this respective order.

Consider that a new component *S* has to be created, perhaps due to a new business requirement, and it needs to execute *B* and *C* only. For the sake of understanding, let's consider that *B* is an *email delivery* mechanism, whereas *C* is a *SMS delivery* mechanism, reason why both are called together. In that situation, the system size is now 6 components, but dependencies increase from 4 to 6.

Fig. 2. Modifying a star-like architecture



In this type of architecture, complexity tends to grow faster than the size of the system. Any change to *B* will impact *S* and *A*. Any change to *C*, it will impact *S* and *A*. This shows that any change to a component X will potentially consume additional time, yielding a system that is less flexible and harder to adapt.

3.2 Pipeline-like design

In the pipeline-like design, a component *A* depends on *B*, which depends on *C*, which depends on *D*, and so on subsequently. Each component is executed by the one that comes before it.

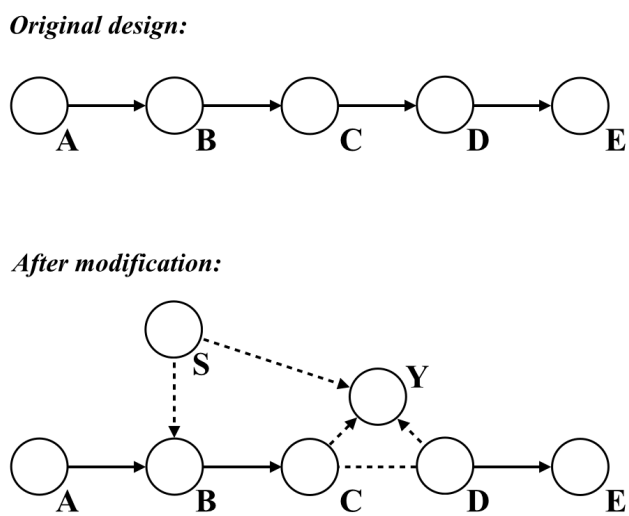
Figure 3 shows the flow of change in such architecture. Before the modification, the system size is 5 components and 4 dependencies.

Consider that a new component *S* has to be created and it needs to execute *B* and *C* only. This poses a serious problem because when the execution is started by *S*, *C* should not execute *D*. For that to be possible, we need to introduce yet a new unit of logic, a component *Y*, which will serve as a switch between *C* and *D*. *Y* depends on *S, C* and *D*.

Another inconvenient side-effect after this modification is that the original dependency between *C* and *D* has to be altered to support the addition of *Y*.

In that situation, the system size increased from 5 to 7 components, but there were 5 side-effects, moving the dependencies count from 4 to 7.

Fig. 3. Modifying a pipeline-like architecture



Any change to *B* will impact *S* and *A*. Any change to *S* will impact *Y*, which will impact *C* and *D*. This domino-effect makes it considerably harder to make any change to the system.

3.3 Tree-like design

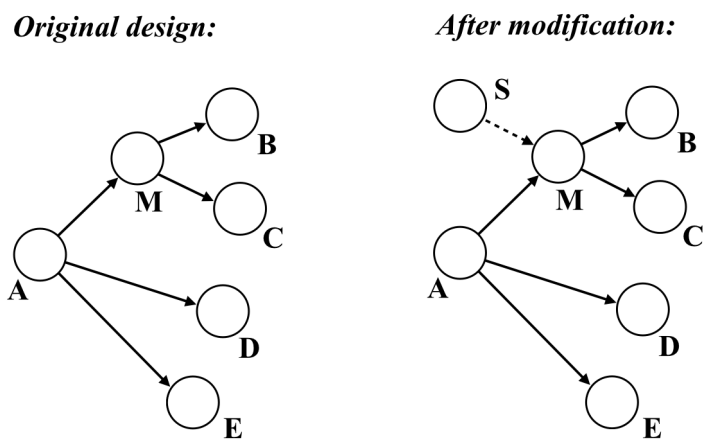
Figure 4 shows our example system in a tree-like design. We introduce a new component to the original design, resulting in a bigger system initially. Albeit counterintuitive, this bigger system will give ways to increase its size at a smaller complexity growth rate.

Before the modification, the system has 6 components and 5 dependencies. Given that *B* (email delivery mechanism) and *C* (SMS delivery mechanism) are likely to be used together, we consider them as part of the same module (Parnas, 1972). The new component *M* goes front of the module, as a node in a tree.

Component *A* now depends on *M, D* and *E*, which are executed in this respective order.

Consider that a new component *S* has to be introduced due to business requirements and it needs to execute *B* and *C* only. In that situation, the system size is now 7 components, but dependencies increased linearly, not exponentially, from 5 to 6.

Fig. 4. Modifying a tree-like architecture



In this example architecture, complexity and possible side-effects due to dependencies were linearly proportional to the growth of the system.

4. Constructal Law in action

One of the tenets of the Constructal Law is that freedom must be present in order for the flow system to evolve into a better design (Bejan & Zane, 2012). One such evidence that the Constructal Law guided the development of software theories that seek to facilitate the flow of change can be seen in the various theories and concepts that emerged in the systems architecture field since its beginning, as a result of researches by companies, academia and other individuals.

Many software concepts were brought to life in the attempt to diminish resistance to change, often times culminating as motivators for the tree-like architectures presented in *3.3 Tree-like design* to be implemented, even if indirectly. There are yet countless opportunities for new ideas and concepts to emerge, as Beck (2010) states, "we're just beginning to understand software evolution […] How can I maintain focus as I make larger changes in small, safe steps over a long period of time, interleaved with feature development?"

Freedom gave opportunity for Parnas (1972) to specify methods for modularization, which is the idea of segregating groups of components by determined criterias. This could lead to architectures similar to the one presented in *3.3 Tree-like design*, where a whole group of software parts aren't accessible directly, but encapsulated behind a coordination component, therefore decreasing the chance of side-effects during modifications.

Freedom gave opportunity for Martin (1994) to define the principle of dependency inversion, which could be represented as the component *M* in *3.3 Tree-like design*. Even if the original paper does not look for a tree-like design, it naturally leads into that direction.

Meyer (1997) defined the *Command Query Separation* (CQS) principle, which states that "functions should not produce abstract side effects". After Meyer's new ideas, there's a clear separation of what will produce side-effects and what will not. The application of this concept will culminate in systems that are easier to change, therefore having more flexibility and being more adaptable. In architectures similar to the one presented in *3.3 Tree-like design*, CQS fills a very important role in decreasing what can unexpectedly change.

Young (2010) takes CQS a step forward and defines the *Command Query Responsibility Segregation* principle (CQRS), which difference is that in CQRS objects (components) are split into two components, one containing the commands and one containing the queries.

Martin (2006) defined the *Single Responsibility* principle, states that "a class should only have one reason to change" for the exact same reason that *3.3 Tree-like design* was a better architecture, to avoid that "when the requirements change,

that change will be manifest through a change in responsibility amongst the classes". The author uses the term classes as a unit of logic, which translates perfectly to components as described in this paper.

In all of these manifestations of knowledge, as well as many others (this paper's goal is not to cite all of them), the objective was the same, to allow software projects to be designed in such a way that less side-effects are present, presenting less resistance to modifications, therefore facilitating the flow of changes in systems. That is evidence that the Constructal Law also governs the development of the software systems.

5. Conclusion

We have gathered evidence that the Constructal Law also governs the evolution of principles in systems' architectures. The principles referenced in this paper all attempt to present ways of decreasing side-effects, therefore decreasing the additional time required for a certain modification to be made.

With that in mind, we are able to design and verify better architectures that have less resistance to change. We can also predict that the future of software architectures, according to the Constructal Law, is one which presents less resistance to the flow of design change, benefitting institutions that live in environment that are constantly evolving.

We conclude that to follow the path of least resistance in the evolution of software design, engineers should foster the development of ideas and principles that drive systems to have less side-effects and less resistance to change.

References

[1] Fauscette, M., Maintaining ERP systems: the cost of change, IDC, city: Framingham, 2013.

[2] Bejan, A., Zane, J.P., Introduction, *Design in Nature: How the Constructal Law Governs Evolution in Biology, Physics, Technology, and Social Organization*, ISBN:978-0-385-53462-8, 1st ed., New York, NY: Doubleday, 2012.

[3] Heineman, G. T., Councill, W. T., Definition of a software component and its elements, *Component-based software engineering: putting the pieces together*, ISBN:0-201-70485-4, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[4] Fokkink, W., Introduction, *Distributed algorithms, an intuitive approach*, ISBN:978-0-262-02677-2, Cambridge, MA: The MIT Press, 2013.

[5] Martin, R.C., Object oriented design quality metrics: an analysis of dependencies, *C++ Report*, 1994.

[6] Parnas, D.L., On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15 (12), 1053-1058, 1972.

[7] Beck, K., The inevitability of evolution, software evolution, *IEEE Software*, 27 (4), 29-38, 2010.

[8] Meyer, B., Principles of class design, *Object-Oriented Software Construction*, 2st ed., 747-764, Upper Saddle River, NJ: Prentice Hall, 1997.

[9] Young, G., Command Query Responsibility Segregation, *CQRS Documents*, retrieved from https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf, 2010.

[10] Martin, R.C., Martin, M., SRP: The single responsibility principle, *Agile Principles, Patterns, and Practices in C#*, 1st ed., 116-120, Upper Saddle River, NJ: Prentice Hall, 2006.