Complexity: Let’s not make this complicated

By Abram Hindle <hindle1@ualberta.ca>

Introduction

“Keep it simple,” is a phrase I like to say when I teach my introduction to software engineering course. “Keeping it simple is easier said than done,” is another phrase I also like to say in the course. It’s funny how keeping it simple in software development can often mean revising and refactoring an existing system until it elegant enough to afford adaptation and change. Simplicity and elegance are goals of many developers when they're developing software. Developers often view complexity as the opposite of simplicity but I argue that complexity is not the right word. I think complicated software is really what people are worried about. In other fields, such as physics or even education [Doll 1993], complexity refers to how agents, individuals, and entities interact with each other via a small set of rules or processes to produce intricate and interesting behaviors much birds and fish flocking and swarming together in complex patterns without collisions. Analogously, developers want their code to compose a solution both clearly and elegantly, allowing for dynamism and adaptability. What developers are really worried about is that their software is burdened by too many modules affected by too many features with cross-cutting concerns. They are concerned that their software will be fragile and hard to change. They are concerned about software that lacks the elegance or dynamism to enable customisation and afford future changes. This is what complicated software is. The systems that we seek to build that exhibit elegance and simplicity are complex systems that through a set of rules or contracts that are intentionally or naturally kept small one can customize and extend such a system with ease. We Fear complication because it leads to brittle designs that are hard to change. Complication means we have to juggle too many competing concerns when we maintain our particular module.

In this article I’ll discuss simplicity in agile software, the relationship between architectural patterns and complexity, the value of simplicity in software engineering research, and why we should refer to the formerly perceived complexity in software as complicated software.

Agile and Simplicity

Agile software development processes and guidelines, consultants, and agile practitioners argue that you should keep it simple [Beck 2004]. This view was born out of experience with developing software systems where functionality and features were created that were not asked for but were perhaps expected, implied, or seemed like a natural necessity at the time. These extra features often caused maintainability problems later. The agile view of simplicity is much

like the systems view of complication and complexity. A simple system in agile is not complicated, it implements the requirements and the user stories and not much more. To add extra features or functionality was to waste time on what was not asked for, as well could complicate future changes. The more responsibilities you gave a module the more you would have to maintain later. So to Agile, keep it simple it also meant don't do what isn't asked for. The agile solution to not addressing potential future requirements was that refactoring, supported by unit testing, was always an option---refactoring which was easier with simple uncomplicated modules, rather than those complicated by too many responsibilities. The unit testing was a feedback mechanism in the process of agile software development, effectively causing Agile systems to exhibit complex behaviour through “simple” rules. In Open-source software Jingwei Wu confirmed that self-organizing and complex behaviours were being exhibited in open source communities [Wu 2007].

Regarding Complexity, it’s Complicated?

Complex software in software engineering typically refers to complicated code. Most measures of complexity are measures of information content in the code, whether it is McCabe’s Cyclomatic Complexity measuring branching, or Halstead’s Volume measuring the information within a block of code---Halstead’s Volume is very similar to entropy of tokens multiplied by number of tokens in a code block. Thus when I refer to complexity I refer to systems and modules with spartan rule sets, and complicated systems and modules are those with lots of concerns and requirements. I argue we should consider changing terminology as complexity is often used to enable elegant systems that are extensible and work well and scale and this is by making a small set of rules or behaviors that a single modules expected to fulfill thus allowing a composition of these sub-modules into an interesting and often complicated looking Software System.

What we really fear in software development regarding complexity is actually complication. If we look to the behavioral theorists, educators interested in self organization, chaos theorists, and some physicists, complex systems are those systems with simple rules that produce elegant and complex behaviors or complicated behaviors---this is referred to as complexity. Software engineers really seek to build systems that are complex and they seek to avoid building systems that are complicated. One example of complexity, as opposed to complicatedness, in software engineering is the architectural pattern of Model View Controller (MVC). The role of model and view in model-view-controller are those of the modules that represent (model objects) versus the modules that present (view objects). MVC allows us to build systems that produce very Dynamic Behavior that respond to changes in the environment quickly that synchronize and do not require a lot of code to keep views synchronized. If one doesn't use a model like MVC and design patterns like the observer pattern it is often hard to update all the relevant GUI components that present the data stored within a model. MVC provides a runtime performance trade-off for design time performance in terms of lack of complication in design and perhaps better maintainability. MVC can produce very elegant systems composed of components that

follow a very small set of rules rules and contracts that allows systems the dynamic behavior that we expect of a high quality applications utilizing modern GUI systems.

Simplistic Structures of Software

Perhaps it is complexity, via simplicity, that makes software work. Tim Menzies has argued with me that an interesting aspect of software is how complicated we think it is yet how often stable it is. The software is being evaluated many times per second and for the most part is is quite stable. Most of the programs you use do not crash every second. Many of them will eventually crash. Many of them do crash. We complain about those who crash but frankly the norm is that software doesn't actually crash on us frequently. Most of the software we use actually does its job and actually works. So how is it we are suffering from complicated software when these software systems falling apart as much as some would have us believe. In Ubuntu, Campbell et al.[Campbell 2016] found that most projects do not have more than 1 crash report causes, while some have many different crash report causes and crash reports. Furthermore, It turns out that the causes of many software crashes are quite predictable, many crashes are caused by a small set of API functions such as strlen, free, and pthread_mutex_lock, with many of these common crashing functions producing the same signal (SEGV or ABRT). Crashes occur commonly in the same contexts with the same functions.

Perhaps our expression of software is more complex than it is complicated? Other researchers have focused on social dynamics and shown this to be case [Wu 2007]. But existing source code is typically full of repetitive and that uninteresting patterns are repeated often to produce software systems. Through our study of software where we treated source code to natural language processing techniques as if source code was a natural utterance [Hindle 2012], we found that the information content of software was quite low compared to English language text. This means that the language we use to define software is quite simple when compared to English text, but it also means it is more repetitive. That is natural language text is far more information dense then software source code. Thus less information is being transmitted per token or word in source code than in English. Now they aren't equivalent, the vocabularies of software source code are often quite large and project specific whereas the vocabulary of a language or writing for a particular language like English is often still large but general and does not change much across documents. This difference in vocabulary might explain why even though software is low in information, the broad vocabulary enables representation of problems via identifier naming. Programs are coded in common patterns.

Simplicity in research

As a software engineering researcher I have to deal with complicated research all the time. My experience is that papers that push for complicated methods typically are harder to replicate. There seems to be more chance of error in communication or replication or reimplementation

and the costs of the new specific complications whether it be algorithmic, features, data-sources, or other dependencies. The benefit to a researcher to build a complicated system is that they've done lots of work to get a system that performs well. But this comes at a cost beyond just the difficulty of replication. For instance if the data required is too expensive to gather or not available it often hinders applying the techniques. The pile-up of additional steps causes a problem where it is hard to replicate the proposed work and the proposed work cannot be used as a baseline unless the source code is actually shared and others can actually replicate the work. That level of sharing is actually a quite high bar where is if a system is kept simple or clearly defined then it enables more reimplantation more replication.

Research that is left uncomplicated enables better analysis of *why* a technique or where a technique would work. Research that is uncomplicated and simple has a higher chance of providing some level of explainability of results through posing relatively simple theories. Furthermore keeping a proposed technique uncomplicated means that errors in methodology and measurement, and other threats to validity can be further minimized. The promotion of simple research faces a barrier that performance of a simple technique might be explainable but it's performance might lag behind more complicated specialized results---this could be a hard sell for some program committees. Probably the largest benefit of simplicity in research is to the researcher themselves, as they allow their work to be impactful through its replication whether as a baseline or a contender.

Conclusion

Keeping it simple is easier said than done, but software faces a lot of factors that promote this keeping it simple ethos: complication is rarely requested, simplicity promotes complex systems that exhibit adaptability and sometimes elegance, software and its failure are often repetitive and predictable. All of these factors provide evidence that software is quite complex [Wu 2007] and perhaps it is simplicity at the heart of software that enables these complex systems that are not overly burdened by complication.

Researchers should consider the value to stakeholders such as developers if methods are kept simple and fundamentally replicable.

[Kent 2004] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional. 2004

[Campbell 2016] JC Campbell, EA Santos, A. Hindle. Anatomy of a crash repository. PeerJ Preprints 4:e2601v1 https://doi.org/10.7287/peerj.preprints.2601v1 2016

[Doll 1993] William E. Doll, A Post-modern Perspective on Curriculum, Teachers College Press, 1993

[Hindle 2012] Abram Hindle, Earl T. Barr, Zhendong Su, Premkumar T. Devanbu, and Mark Gabel **On the Naturalness of Software** International Conference on Software Engineering (ICSE-2012) Zurich, Switzerland 2012 pp. 837--847

[Wu 2007] J. Wu, R. C. Holt and A. E. Hassan, "Empirical Evidence for SOC Dynamics in Software Evolution," *2007 IEEE International Conference on Software Maintenance*, Paris, 2007, pp. 244-254.