The Importance of Structure in Software

Jordan Colgan
www.codurance.com

Is writing ‘good’ code, *'clean'* code ever enough in itself? What makes code good? You may value the ability to read through the code and understand it, without having to compile and run it. Others may value that, and also have the code prove its correctness through tests.

In the Software Industry, it is tediously repeated that programmers should value disciplines and have pride that their code is self-documenting and self-proving [1]. These are not recent discoveries or revelations either. Some of the first literature about software describe approaches to proving then building computer programs, early predecessors to the practice we know today as test-driven development.

During World War II, John Mauchly and J. Presper Eckert set out to develop America’s first programmable computer, Electronic Numerical Integrator and Computer (ENIAC), with the initial goal to automate artillery ballistic computations [2]. These ballistic tables were originally being produced by humans, nearly 200 female clerks, who had been calculating the tables with mechanical desk calculators prior to the development of ENIAC. Some of these women were selected to program the machine that would replace them and their colleagues. These clerks, turned programmers, used their practice of calculating the tables manually first to drive the development of ENIAC, and also to narrow down bugs due to failures within its physical components [3].

Ideas about manually checking the expected output of computer programs continued, with a notable mention in “Digital Computer Programming” [4], from 1957:

> The first attack on the checkout problem may be made before coding is begun. In order to fully ascertain the accuracy of the answers, it is necessary to have a hand-calculated check case with which to compare the answers which will later be calculated by the machine. This means that stored program machines are never used for a true one-shot problem. There must always be an element of iteration to make it pay. The hand calculations can be done at any point during programming. Frequently, however, computers are operated by computing experts to prepare the problems as a service for engineers or scientists. In these cases it is highly desirable that the "customer" prepare the check case, largely because logical errors and misunderstandings between the programmer and customer may be pointed out by such procedure. If the customer is to prepare the test solution, it is best for him to start well in advance of actual checkout, since for any sizable problem it will take several days or weeks to hand-calculate the test.

This sounds like the beginnings of acceptance test-driven development (ATDD[1]), where the "customer" or someone representing the customer (business analyst, product manager, etc) works together with a quality assurance engineer to produce a set of executable specifications of how they expect the program to work. Prior to any feature development having begun [5]. These executable specifications are then implemented by the developers, as they build out the feature and use their passing state to mark it as 'done'. ATDD is of course a branch of TDD, a practice used by developers to drive out their own tests, separate from the acceptance tests provided by the business and quality assurance.

Another notable mention of test based development is from NASA's Project Mercury in the 1960s [6]. Independent test engineers wrote testing procedures, from the hardware requirements provided, before the developers wrote the software to integrate with the hardware. This allowed them to shorten the overall development time and increase the parallelity of quality assurance. Compared to other software projects at the time, which were beginning to be affected by the looming 'Software Crisis'.

The exact discipline of TDD first started to form in 1993 [7], as 'test-first programming', which then later became the more refined test-driven development we are familiar with today. Kent Beck is credited with inventing TDD, but he refers to having rediscovered the discipline. We can see why he would think this way, with valuing proof before implementation documented throughout the history of software development.

TDD is just one part of the Extreme programming (XP) methodology. Conceived with the intention to improve software quality and respond to the ever growing needs of business agility [8]. Despite XP practices being an adopted toolset of many modern programmers, the majority are yet to accept it as their own. This attitude against XP feels almost antagonistic, considering that we are said to be in a 'Software Crisis'.

The term 'Software Crisis' arose during the first NATO Software Engineering Conference in 1968 at Garmisch, Germany [9]. Edsger Dijkstra also made a reference to the same problem in his 1972 Turing Award lecture [10]. The main outlines related to the software aspect are low quality, inefficiency, and not meeting the desired requirements. In terms of the overall project delivery, software projects typically exhibited problems of over-budget, over-time, and low developer retention due to the software quality.

Although the software crisis was identified long ago, software built today still has the same issues. It seems like a difficult argument to still call it a crisis, considering it has been going on for so long. Why does software continue to fail and fail again? To answer that question, we first need to understand the values of software.

The Two Values of Software

What exactly is Software? "Britannica" offers the following definition [11]:

[1] ATDD is also Behaviour Driven Design (BDD). BDD is often mistaken to mean using the Gherkin syntax (Given, When, Then), but this is not true.

> Software, instructions that tell a computer what to do. Software comprises the entire set of programs, procedures, and routines associated with the operation of a computer system. The term was coined to differentiate these instructions from hardware—i.e., the physical components of a computer system. A set of instructions that directs a computer's hardware to perform a task is called a program, or software program.

Software was invented to change the behaviour of machines, the hardware. The reason as to why is held in the word "hardware", meaning hard to change. It takes enormous effort and resources to design and build hardware, therefore we cannot spend anymore time trying to change it for various other reasons. Software promised the ability to easily change the output for hardware it ran on. From this, we can conclude that a value of software is its "behaviour".

What would the other value of software be then? That would be how we share the instructions the software performs, the architecture. Architecture in the software world is difficult to provide a definition for. Quite often it is associated with just the high level overview of how components interact with each other, but it is much more than that [12]. It is everything, from the high-level decisions to the low-level details. Architecture is the continuous growing shape of the system. Therefore, the second value of software is its "structure".

Which is more important? Is it the behaviour, the structure, or are they both equally important? If you were to ask developers which were more important to them, they would most likely answer behaviour. Programmers are hired to change the way machines behave, in order to make or save money for their stakeholders. Therefore, many programmers assume their default priority is behaviour, as it can seem like the entirety of their job. It is their responsibility to implement the requirements and fix any mistakes in the softwares functionality.

If you were to ask the stakeholders, managers, and anyone on the 'business' side what the value of software is to them, they of course would answer behaviour. To them, the process looks like providing the developers with functional specifications and requirement documentation, to which the developers somehow work the machines into what they asked for, eventually. Without their requests, the programmers would not have a job to do.

So far, it seems obvious that it is more important for the system to work than it is for the system to have sound structure. However, consider again that the whole reason software was invented was to *easily* change the behaviour of the hardware. The purpose of software is to be soft, easy to change. Software that is easy to change the structure of can easily be made to work. That is, software that is easy to change can keep up with the ever changing requirements from the business.

Software that works, but suffers bad structure that is hard to change, results in an unwanted situation in which the software struggles to meet requirements. When the business wants to add new features or extend existing ones, it takes too much time to rework the system to meet the new business requirements. Even worse, software that cannot be extended due to the tangled mess is seen as useless and loses its value to the business. The business only

sees programmers as being valuable due to their ability to make the machines do as they ask. When programmers lose control of the software, then they too lose their value.

You, as a developer, may realise the importance of structure by now, but how do you convince your manager or the business that structure is more important than behaviour? Consider the business and their competitors. The software you write for the business has better features than their competitors. You are able to get these features out quicker, but only because you cut corners with the quality of the software. Of course, you say to yourself, you will go back and make it better. Your competitors however do care about quality and they insist on making the software the best structure they can as they go along, but this makes them slower.

What happens when you cut corners on the software quality, and promise to go back and clean it? You convince yourself that someday you will make it better, someday you will fix the hacks made. That someday will never come [13]. Even if you do find yourself with the opportunity to present the idea of going back to rectify the mistakes, you will find that the priorities have moved away from the area. Of course, this is the nature of software and the world of ever changing requirements.

Your software that once had the features that made you the market preference, has now been overtaken by your competitors. Their structure allowed them to catch up to your features, and it even encouraged the change required to go beyond what you had.

Programmers face a dilemma. The architecture of their system is more important than the urgency of the functionality required of it. For without the ability to respond easily to change, they can't consistently deliver and sustain business needs. The business does not have the knowledge to understand why architecture matters and why time must be spent caring for it. Therefore, it is up to the programmers to learn and master the techniques required of them to maintain sound structure.

Bridging the Gap between the Code and Business

Architecture of a system supports the intent of the system. Although the architecture is more important than the behaviour for agility purposes, it's the behaviour of the system that influences the shape of the system. An ecommerce platform will have an entirely different architecture to that of a social network.

The act of translating the requirements into code is known as domain modelling [14]. Domain models act as the link between the ubiquitous language forming the requirements in the code and the executable binary the machine processes. Even if the domain is a real world tangible concept, the model is still our artificial representation in the world of data. It comprises the unique abstractions, knowledge, and processes the business utilises within the machine.

Domain models are often where software structure collapses. Programmers work in details, low-level details. They care about frameworks, interfaces, concurrency, optimisation, just to

name a few. While these are important, they are not things the business understands or cares about. When programmers try to build domain models without guiding principles that ensure inversion from low-levels[2], they often mix in their concerns with the businesses.

It just isn't the programmer level where modelling fails, many high-level actors and operations can impact decisions that result in poor domain modelling. Multiple models will exist in organisations with large codebases, from monoliths to scattered repositories that connect to each other. Different models will have different contexts, but some of these models can be the same yet will be independently worked on. Miscommunication between the teams involved will result in subtle interpretations within models, meaning those models can't be shared. Duplication is the enemy of software and difficult to fix.

Is unreliable, convoluted, and duplicated software just the nature of large codebases with many teams or are there ways to rectify? Models are applied in contexts, the context being an area of the codebase or the features being worked on by a team. Model harmony can exist and scale with developer or business operations. It requires intercommunication, not only amongst the teams but also in the codebase. Model scopes are defined as bounded parts of the software system, representing the limits of where the model will apply and its ownership.

The product of defining the boundaries within the organisation and teams, and specifying usages within the system, are known as 'bounded contexts'. Bounded contexts provide teams clarity and encourage shared understanding of what needs to be cohesive and the potential coupling by other contexts. There is a better understanding of when to use the same model and when not to. It also brings awareness of trade-offs of sharing, which helps during the processes of team collaboration. Team synergy will most likely fail if everyone does not understand where the bounds of the model contexts lay.

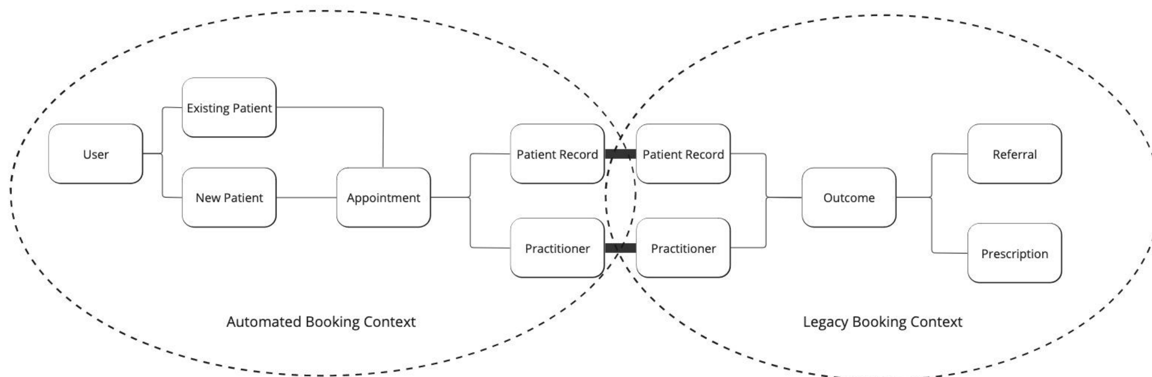
How do you manage bounded contexts? There are many nuances around different types of software systems, but consider a medical institution who initiates an internal project for appointment bookings. This project seeks to automate the process of booking with practitioners, rather than patients having to manually organise them through receptionists.

What would the bounded contexts look like for the model of this application? There is already an existing model with the legacy booking system used by receptionists. This system is maintained by a team, who have been directly influencing the model. Another team is required for the new user facing booking application. Their expected model will have similarities to the existing model, but there are some discrepancies based on use cases they have discovered.

The bookings from the new application need to be passed to the legacy booking system. An agreement was made between the teams that a new model would be formed from the legacy model. Therefore the legacy system is outside the boundary. Translations will be required between this new model and the old. The responsibility of these translations falls upon the legacy team, since contexts are traversing outside the agreed upon boundary.

---

[2] Also known as the "Dependency Inversion Principle". Mistakenly thought to only apply to Object-Oriented design, but it represents a much higher level design principle.



In our example, the teams decide on a monolithic codebase to house both of their implementations. Does this mean bounded contexts are modules? It could be interpreted that the bounded contexts are the same as modules, since each team has their own modules and sub-modules, but bounded contexts and modules have different motivations. Modules are programming language paradigms, and bounded contexts must be agnostic to such paradigms. The purpose of modules is to organise different elements within a model. They don't always communicate clearly the intention of separated contexts.

Bounded contexts are just a part of the steps towards better team topology and code cohesion. There are other strategic design patterns along with bounded contexts, forming what we know as Domain Driven Design (DDD) [15].

The Constant Pursuit of Quality

Good structure enables good behaviour, throughout both the software and the people. Bad structure obstructs good behaviour. Flexibility for ease of change must be in from the very beginning of system implementation. Without keeping the structure clean, we are setting ourselves up to never follow the path of sustainable development.

How good is good enough? There are many metrics we can use to measure code quality, cohesiveness, and agility. As much as software has the property of change, so do our tactics and approaches to its development and maintenance. The best attitude that you can have is to always be in the constant pursuit of quality. From the Programmers Oath [16], the second promise is:

> The code that I produce will always be my best work. I will not knowingly allow code that is defective either in behaviour or structure to accumulate

Improving software structure is hard. It requires many years of intricate knowledge, spanning many different fields in computing. Not only is deep theoretical knowledge essential, but vast practical experience is crucial. As one must pursue a constant high quality software structure, so should they also pursue a high quality in their own skills and knowledge.

Professional developers must place great importance on the structure of code over the behaviour. Software has been, and will probably continue to be, a fast accelerated realm in our world. Demands for developers are high, and demands for developers to be quick is higher. Not every developer has the privilege to learn that the only way to go fast is to go well. Therefore, it is our duty as those who know to guide those who do not.

> The strong should aid and protect the weak. Then, the weak will become strong and they, in turn, will aid and protect those weaker than them. That is the law of nature.
>
> -Tanjiro Kamado

References

1. Martin, Robert C (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. ISBN 9780136083238.
2. Light, Jennifer S (1999). "When Computers Were Women". ISSN 0040-165X.
3. Fritz, Barkley W (1996). "The Women of ENIAC". IEEE Annals of the History of Computing. doi:10.1109/85.511940.
4. McCracken D D (1957). Digital Computer Programming. John Wiley & Sons. ISBN 9780471582458.
5. Pugh, Ken (2011). Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration. Addison-Wesley. ISBN 978-0321714084.
6. Larman, C and Basili, VR (2003). "Iterative and incremental developments. a brief history" (PDF). Computer. doi:10.1109/MC.2003.1204375.
7. Beck, Kent (2002). Test-Driven Development by Example. Vaseem: Addison Wesley. ISBN 9780321146533.
8. Beck, Kent (1999). Extreme Programming Explained. Addison-Wesley Professional. ISBN 9780201616415.
9. Naur, P and Randell, B (1969). “Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968”. Scientific Affairs Division, NATO.
10. Dijkstra, Edsger W (1972). “The humble programmer”. Communications of the ACM 15.10.
11. Britannica, The Editors of Encyclopaedia (2023). "software". Encyclopedia Britannica.
12. Martin, Robert C (2017). Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Pearson. ISBN 9780134494272.
13. Kondo, Marie (2014). The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing. Clarkson Potter/Ten Speed. ISBN 9781607747307.
14. Evans, Eric (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. ISBN 0321125215.
15. Fowler, Martin (2014). “BoundedContext”. Martin Fowler.
16. Martin, Robert C (2015). “The Programmer's Oath”. The Clean Code Blog.